# The Fortress Language Specification

Version 1.0

Eric Allen
David Chase
Joe Hallett
Victor Luchangco
Jan-Willem Maessen
Sukyoung Ryu
Guy L. Steele Jr.
Sam Tobin-Hochstadt

Additional contributors:
Joao Dias
Carl Eastlund
Christine Flood
Yossi Lev
Cheryl McCosh
Janus Dam Nielsen
Dan Smith

March 31, 2008

This release of the Fortress Language Specification is the first to be released in tandem with a compliant interpreter, available as open source and online at:

`http://projectfortress.sun.com`

Our tandem release of a specification and matching interpreter is a major milestone for the project; it is a goal we have been working toward for some time. All Fortress source code appearing in this specification has been tested by executing it with the open source Fortress implementation. Moreover, all code has been rendered automatically with the tool *Fortify*, also included with the standard Fortress distribution. Fortify is an open source tool for converting Fortress source code to LaTeX.
Our reference implementation has been evolving gradually, in parallel with the evolution of the language specification and the development of the core libraries. In order to synchronize the specification with the implementation, it was necessary both to add features to the implementation and to drop features from the specification. Most significantly, most static checks in the implementation are currently turned off, as we are in the process of completing the static type checker and the type inference engine. Static constraints are still included in the specification as documentation. Contrary to the Fortress Language Specification, Version $1.0\,\beta$, inference of static parameter instantiations is based on the runtime types of the arguments to a functional call. Support for syntactic abstraction is not included in this release. We do not yet support nontrivial distributions, nor parallel nested transactions. Moreover, many other minor language features defined in the Fortress Language Specification, Version $1.0\,\beta$ have been elided. All of these features require additional research before they can be implemented reliably; this research and development is a high priority for the Fortress team.

With this release, our goal in moving forward is to incrementally add back features taken out of the specification as they are implemented. In particular, all language features included in the Fortress Language Specification, Version $1.0\,\beta$ remain goals for eventual inclusion in the language (perhaps with additional modification and evolution of their design). By proceeding in this manner, we believe that our implementation will be useful for more tasks more quickly, as it will comply with the public specification. Moreover, the Fortress community will be better able to evaluate the design of new features, as users will be able to use them immediately, and developers will be able to contribute to the implementation effort more easily, as they will be able to build off of a relatively stable and well-specified base.

Moving forward with the implementation, in concert with our open source community, our goal is to build off of the infrastructure of our interpreter to construct the first optimizing Fortress compiler and to achieve our long-standing goal of constructing a new programming language with high performance and high programmer productivity, owned by the community that uses it, and able to grow gracefully with the tasks it is applied to.

# Contents

# Part I

# Preliminaries

# Chapter 1

# Introduction

The Fortress programming language is a general-purpose, statically typed, component-based programming language designed for producing robust high-performance software with high programmability.

In many ways, Fortress is intended to be a “growable language”, i.e., a language that can be gracefully extended and applied in new and unanticipated contexts. Fortress supports state-of-the-art compiler optimization techniques, scaling to unprecedented levels of parallelism and of addressable memory. Fortress also supports modular and extensible parsing, allowing new notations and static analyses to be added to the language.

The name “Fortress” is derived from the intent to produce a “secure Fortran”, i.e., a language for high-performance computation that provides abstraction and type safety on par with modern programming language principles. Despite this etymology, the language is a new language with little relation to Fortran other than its intended domain of application. No attempt has been made to support backward compatibility with existing versions of Fortran; indeed, many new language features were invented during the design of Fortress. Many aspects of Fortress were inspired by other object-oriented and functional programming languages, including the Java™ Programming Language [3], NextGen [4], Scala [18], Eiffel [14], Self [1], Standard ML [16], Objective Caml [12], Haskell [20], and Scheme [11]. The result is a language that employs cutting-edge features from the programming-language research community to achieve an unprecedented combination of performance and programmability.

Fortress is an open source project. An interpreter, implementing the language features presented in this specification, is available at the Fortress project website:

`http://projectfortress.sun.com`

There you will find source code, supporting documents, and access to discussion groups related to the Fortress project.

## 1.1 Fortress in a Nutshell

Two basic concepts in Fortress are that of *object* and of *trait*. An object consists of *fields* and *methods*. The fields of an object are specified in its definition. An object definition may also include method definitions.

Traits are named program constructs that declare sets of methods. They were introduced in the Self programming language, and their semantic properties (and advantages over conventional class inheritance) were analyzed by Ducasse, Nierstrasz, Schärli, Wuyts and Black [5]. In Fortress, a method declared by a trait may be either *abstract* or *concrete*: abstract methods have only *headers*; concrete methods also have *definitions*. A trait may *extend* other traits: it *inherits* the methods provided by the traits it extends. A trait provides the methods that it inherits as well as those explicitly declared in its declaration.

Every object extends a set of traits (its “supertraits”). An object inherits the concrete methods of its supertraits and must include a definition for every method declared but not defined by its supertraits.

Fortress allows the use of Unicode characters [22] in program identifiers, as well as subscripts and superscripts. Fortress also includes a set of standard formatting rules that follow the conventions of mathematical notation. For example, most variable references in Fortress programs are italicized. Moreover, multiplication can be expressed by simple juxtaposition. There is also support for operator overloading.

Although Fortress is statically and nominally typed, types are not specified for all fields, nor for all method parameters and return values. Instead, wherever possible, *type inference* is used to reconstruct types. In the examples throughout this specification, we often omit the types when they are clear from context. Additionally, types can be parametric with respect to other types and values (most notably natural numbers).

These design decisions are motivated in part by our goal of making the scientist/programmer’s life as easy as possible without compromising good software engineering. In particular, they allow us to write Fortress programs that preserve the look of standard mathematical notation.

In addition to objects and traits, Fortress allows the programmer to define top-level functions. Functions are first-class values: They can be passed to and returned from functions, and assigned as values to fields and variables. Functions and methods can be overloaded, with calls to overloading methods resolved by multiple dynamic dispatch similarly to the manner described in [15]. Variable size argument lists are also supported.

Fortress programs are organized into *components*, which export and import APIs. APIs describe the “shape” of a component, specifying the types in traits, objects and functions provided by a component. All external references within a component (i.e., references to traits, objects and functions implemented by other components) are to APIs imported by the component. We discuss components and APIs in detail in Chapter 20.

To address the needs of modern high-performance computation, Fortress also supports a rich set of operations for defining parallel execution of large data structures. This support is built into the core of the language. For example, `for` loops in Fortress are parallel by default.

## 1.2 Acknowledgments

With the help of many contributors and supporters, we are in the midst of building an open source interpreter for Fortress. Many people have contributed to the Fortress project with useful comments and suggestions for both the specification and implementation, code contributions, and supporting tools. Special thanks go to Bob Apthorpe, Mike Atkinson, Alex Battisti, David J. Biesack, Steve Brandt, Martin Buchholz, Zoran Budimlic, Bill Callahan, Corky Cartwright, Darren Dale, Joe Darcy, David Detlefs, Dave Dice, Kento Emoto, Matthias Felleisen, Robby Findler, Kathi Fisler, Richard Gabriel, Markus Gaisbauer, Alex Garthwaite, Jesse Glick, Brian Goetz, Robert Grimm, Yuto Hayamizu, Steve Heller, Maurice Herlihy, James Hughes, Zhenjiang Hu, Andy Johnson, Mackale Joyner, Miriam Kadansky, Kazuhiko Kakehi, Ken Kennedy, Scott Kilpatrick, Alex Kravets, Shriram Krishnamurthi, Eric Lavigne, Doug Lea, Vass Litvinov, Eugene Loh, Kiminori Matsuzaki, Daniel MD, John Mellor-Crummey, Daniel Mendes, Jim Mitchell, Mark Moir, Akhilesh Mritunjai, Dan Nussbaum, Andrew Pitonyak, Bill Pugh, Doron Rajwan, John Reppy, John Rose, Ulrich Schreiner, Nir Shavit, Michael Spiegel, Bob Sproull, Walid Taha, Masato Takeichi, Peter Vanderbilt, Michael Van de Vanter, Chris Vick, Larry Votta, Jim Waldo, Russel Winder, Kathy Yelick, and the many other participants in the Fortress community.

We would also like to thank the anonymous referees of peer-reviewed papers related to the Fortress project, who have influenced the language design in many ways, as well as the DARPA reviewers from the HPCS project for their thoughtful feedback throughout the design of the language.

## 1.3 Organization

This language specification is organized as follows. In Part II, the Fortress language features for application programmers are explained, including objects, types, and functions. Relevant parts of the concrete syntax are provided with many examples. The full concrete syntax of Fortress is described in Appendix D. Part III describes advanced Fortress language features for library writers. In Part IV, APIs and documentation of the Fortress Library are presented. Finally, in Part V, the Fortress calculi and the Fortress grammars are described.

# Chapter 2

# Overview

In this chapter, we provide a high-level overview of the entire Fortress language. We present most features in this chapter through the use of examples, which should be accessible to programmers of other languages. In this chapter, unlike the rest of the specification, no attempt is made to provide complete descriptions of the various language features presented. Instead, we intend this overview to provide useful context for reading other sections of this specification, which provide rigorous definitions for what is merely introduced here.

## 2.1 The Fortress Programming Environment

Although Fortress is independent of the properties of a particular platform on which it is implemented, it is helpful for concreteness to present the programming model used in the Fortress reference implementation. In this programming model, Fortress source code is stored in files and organized in directories, and there is a text-based shell from which we can store environment variables and issue commands to execute and compile programs.

A Fortress program can be processed in one of two ways:

- It can be *executed*. The Fortress program is stored in a file with the suffix "`.fss`" and executed directly from an underlying operating system shell by calling the command "`fortress run`" on it. For example, suppose we write the following "`Hello, world!`" program to a file "`HelloWorld.fss`":

  ```
  export Executable
  run(args) = print "Hello, world!"
  ```

  The first line is an *export statement*; we ignore it for the moment. The second line defines a function $run$, which takes a parameter named $args$ and prints the string "`Hello, world!`". Note that the parameter $args$ does not include a declaration of its type. In many cases, types can be elided in Fortress and inferred from context. (In this case, the type of $args$ is inferred based on the program's export statement, explained in Section 2.2.)

  We can execute this program by issuing the following command to the shell:

  ```
  fortress run HelloWorld.fss
  ```

  The instruction `fortress run` can be abbreviated as `fortress`:

  ```
  fortress HelloWorld.fss
  ```

- It can be *compiled*. In this case, the Fortress program is compiled into a *component*, which is stored in a hidden persistent cache maintained by the implementation. called a *fortress*. Typically, a single fortress holds all the components of a user, or group of users sharing programs and libraries. In our examples, we often refer to the fortress we are storing components in as *the resident fortress*.

For example, we could have written our "`Hello, world!`" program as follows:

```
component HelloWorld
  export Executable
  run(args) = print "Hello, world!"
end
```

We can compile this program, by issuing the command "`fortress compile`" on it:

```
fortress compile HelloWorld.fss
```

As a result of this command, a component named "HelloWorld" is stored in the resident fortress. The name of this component is provided by the enclosing component declaration surrounding the code. If there is no enclosing component declaration, then the contents of the file are understood to belong to a single component whose name is that of the file it is stored in, minus its suffix. For example, suppose we write the following program in a source file named "`HelloWorld2.fss`":

```
export Executable
run(args) = print "Hi, it's me again!"
```

When we compile this file:

```
fortress compile HelloWorld2.fss
```

the result is that a new component with the name HelloWorld2 is stored in the resident fortress. Once this component is compiled, we can execute it by issuing the following command:

```
fortress run HelloWorld2.fss
```

Compiling a source file allows us to catch static errors before running a program. It also allows the system to perform static optimizations on a program and use those optimizations when executing.

Once a program is compiled, the cached code will be used during execution unless modifications have been made to the source file since the most recent compilation. If the source file is newer, the program is recompiled before execution.

A source file must contain exactly one component declaration, and its name must match the file name.

## 2.2 Exports and Imports

When a component is defined, it can include *export statements*. For example, all of the components we have defined thus far have included the export statement "`export` Executable". Export statements list various *APIs* that a component implements. Unlike in other languages, APIs in Fortress are themselves program constructs; programmers can rely on standard APIs, and declare new ones. API declarations are sequences of declarations of variables, functions, and other program constructs, along with their types and other supporting declarations. For example, here is the definition of API Executable:

```
api Executable
  run(args: String...): ()
end
```

This API contains the declaration of a single function $run$, whose type is $(\mathrm{String}\ldots) \rightarrow ()$. This type is an *arrow type*; it declares the type of a function's parameter, and its return type. The function $run$ includes a single parameter;

the notion $(\mathrm{String}\ldots)$ indicates that it is a *varargs* parameter; the function $run$ can be called with an arbitrary number of string arguments. For example, here are valid calls to this function:

```
run("a simple", " example")
run("run(...)")
run("Nobody", "expects", "that")
```

The return type of $run$ is $()$, pronounced "void". Type $()$ may be used in Fortress as a return type for functions that have no meaningful return value. There is a single value with type $()$: the value $()$, also pronounced "void". References to value $()$ as opposed to type $()$ are resolved by context.

As with components, APIs can be defined in files and compiled. APIs must be defined in files with the suffix `.fsi`. An `.fsi` file contains source code for exactly one API, and its name must match the file name, minus the suffix. If there are no explicit "`api`" headers, the file is understood to define a single API whose name is the name of the containing file (minus its suffix).

An API is compiled with the shell command "`fortress compile`". When an API is compiled, it is installed in the resident fortress.

For example, if we store the following API in a file named "`Zeepf.fsi`":

```
api Zeepf
  foo: String → ()
  baz: String → String
end
```

then we can compile this API with the following shell command:

```
fortress compile Zeepf.fsi
```

This command compiles the API Zeepf and installs it in the resident fortress.

A component that exports an API must provide a definition for every program construct declared in the API. For example, because our component HelloWorld:

```
component HelloWorld
  export Executable
  run(args) = print "Hello, world!"
end
```

exports the API Executable, it must include a definition for the function $run$. The definition of $run$ in HelloWorld need not include declarations of the parameter type or return type of $run$, as these can be inferred from the definition of API Executable.

Components are also allowed to *import* APIs. A component that imports an API is allowed to use any of the program constructs declared in that API. For example, the following component imports the API Zeepf and calls the function *foo* declared in Zeepf:

```
component Blargh
  import Zeepf.{...}
  export Executable
  run(args) = foo("whatever")
end
```

The component Blargh imports declaration *foo* from the API Zeepf and exports the API Executable. Its $run$ function is defined by calling function *foo*, defined in Zeepf. In an import statement of the form:

```
import A.{S}
```

all names in the list of names $S$ are imported from API $A$, and can be referred to as unqualified names within the importing component. In the example above, the names we have imported consist of a single name: $\mathit{foo}$. If we had instead written:

$\mathtt{import}\ \mathrm{Zeepf}.\{\mathit{foo}, \mathit{baz}\}$

then we would have been able to refer to both $\mathit{foo}$ and $\mathit{baz}$ as unqualified names in $\mathrm{Blargh}$.

Note that no component refers directly to another component, or to constructs defined in another component. Instead, *all external references go through APIs*. This level of indirection provides us with significant power. It provides a way to hide implementation details in a component. Also, our intention is that in future implementations, it will be possible to make use of APIs to provide sophisticated link and upgrade operations for the purposes of building large programs.

Components that contain no import statements and export the API $\mathrm{Executable}$ are referred to as *executable components*. They can be compiled and executed directly as stand-alone components. All of our $\mathrm{HelloWorld}$ components are executable components. However, if a component imports one or more APIs, it cannot be executed as a stand-alone program. Instead, its imports are resolved to other components that export all of the APIs it imports, to form a new *compound* component. For example, we define the following component in a file named `Ralph.fss`:

$\mathtt{export}\ \mathrm{Zeepf}$
$\mathit{foo}(s) = ()$
$\mathit{baz}(s) = s$

We can now issue the following shell commands:

```
fortress compile Ralph.fss
fortress compile Blargh.fss
fortress run Blargh.fss
```

The first two commands compile files `Ralph.fss` and `Blargh.fss`, respectively, and install them in the resident fortress. The third command tells the resident fortress to run component $\mathrm{Blargh}$. References to API $\mathrm{Zeepf}$ in $\mathrm{Blargh}$ are resolved to their implementation in component $\mathrm{Ralph}$.

## 2.3 Rendering

One aspect of Fortress that is quite different from many other languages is that various program constructs are rendered in particular fonts, so as to emulate mathematical notation. For example, variable names are rendered in italic fonts. Many other program constructs have their own rendering rules. For example, the operator ˆ indicates superscripting in Fortress. A function definition consisting of the following ASCII characters:

```
f(x) = x^2 + sin x - cos 2 x
```

is rendered as follows:

$f(x) = x^2 + \sin x - \cos 2x$

Array indexing, written with brackets:

```
a[i]
```

is rendered as follows:

$a_i$

There are many other examples of special rendering conventions.

There are also ASCII abbreviations for writing down commonly used Fortress characters. For example, ASCII identifiers for all Greek letters are converted to Greek characters (e.g., "`lambda`" becomes $\lambda$ and "`LAMBDA`" becomes $\Lambda$). Here are some other common ASCII shorthands:

| | | | | | |
|---|---|---|---|---|---|
| `BY` | *becomes* | $\times$ | `TIMES` | *becomes* | $\times$ |
| `DOT` | *becomes* | $\cdot$ | `CROSS` | *becomes* | $\times$ |
| `CUP` | *becomes* | $\cup$ | `CAP` | *becomes* | $\cap$ |
| `BOTTOM` | *becomes* | $\bot$ | `TOP` | *becomes* | $\top$ |
| `SUM` | *becomes* | $\sum$ | `PROD` | *becomes* | $\prod$ |
| `INTEGRAL` | *becomes* | $\int$ | `EMPTYSET` | *becomes* | $\emptyset$ |
| `SUBSET` | *becomes* | $\subset$ | `NOTSUBSET` | *becomes* | $\not\subset$ |
| `SUBSETEQ` | *becomes* | $\subseteq$ | `NOTSUBSETEQ` | *becomes* | $\not\subseteq$ |
| `EQUIV` | *becomes* | $\equiv$ | `NOTEQUIV` | *becomes* | $\not\equiv$ |
| `IN` | *becomes* | $\in$ | `NOTIN` | *becomes* | $\notin$ |
| `LT` | *becomes* | $<$ | `LE` | *becomes* | $\leq$ |
| `GT` | *becomes* | $>$ | `GE` | *becomes* | $\geq$ |
| `EQ` | *becomes* | $=$ | `NE` | *becomes* | $\neq$ |
| `AND` | *becomes* | $\wedge$ | `OR` | *becomes* | $\vee$ |
| `NOT` | *becomes* | $\neg$ | `XOR` | *becomes* | $\oplus$ |
| `INF` | *becomes* | $\infty$ | `SQRT` | *becomes* | $\surd$ |

## 2.4 Some Common Types in Fortress

Fortress provides a wide variety of standard types, including $\mathrm{String}$, $\mathrm{Boolean}$, and various numeric types. The floating-point type $\mathbb{R}64$ (written in ASCII as `RR64`) is the type of 64-bit precision floating-point numbers. For example, the following function takes a 64-bit float and returns a 64-bit float:

$$\mathit{halve}(x{:}\,\mathbb{R}64){:}\,\mathbb{R}64 = x/2$$

64-bit integers are denoted by the type $\mathbb{Z}64$ and 32-bit integers by the type $\mathbb{Z}32$.

## 2.5 Functions in Fortress

Fortress allows recursive, and mutually recursive function definitions. Here is a simple definition of a $\mathit{factorial}$ function in Fortress:

```
factorial(n) =
    if n = 0 then 1
    else n factorial(n − 1) end
```

### 2.5.1 Juxtaposition and function application

In the definition of $\mathit{factorial}$, note the juxtaposition of parameter $n$ with the recursive call $\mathit{factorial}(n-1)$. In Fortress, as in mathematics, multiplication is represented through juxtaposition. By default, two expressions of numeric type that are juxtaposed represent a multiplication. On the other hand, juxtaposition of an expression of function type with another expression to its right represents function application, as in the following example:

$$\sin x$$

Moreover, juxtaposition of expressions of string type represents concatenation, as in the following example:

```
"Hi," " it's" " me" " again."
```

In fact, juxtaposition is an operator in Fortress, just like $+$ and $-$, that is overloaded based on the runtime types of its arguments.

### 2.5.2 Varargs Parameters

It is also possible to define functions that take a variable number of arguments. For example:

```
$printFirst(xs{:}\,\mathbb{Z}32\ldots) =$
  if $xs.reduce(\mathrm{SizeReduction}[\![\mathbb{Z}32]\!]()) > 0$ then $println\ xs_0$
  else throw $\mathrm{Error}$ end
```

This function takes an arbitrary number of integers and prints the first one (unless it is given zero arguments; then it throws an exception).

### 2.5.3 Function Overloading

Functions can be overloaded in Fortress by the types of their parameters. Calls to overloaded functions are resolved based on the runtime types of the arguments. For example, the following function is overloaded based on parameter type:

$$size(x{:}\,\mathrm{Nil}) = 0$$
$$size(x{:}\,\mathrm{Cons}) = 1 + size(rest(x))$$

If we call $size$ on an object with runtime type $\mathrm{Cons}$, the second definition of $size$ will be invoked *regardless of the static type of the argument.*

### 2.5.4 Function Contracts

Fortress allows *contracts* to be included in function declarations. Among other things, contracts allow us to *require* that the argument to a function satisfies a given set of constraints, and to *ensure* that the resulting value satisfies some constraints. They provide essential documentation for the clients of a function, enabling us to express semantic properties that cannot be expressed through the static type system.

Contracts are placed at the end of a function header, before the function body. For example, we can place a contract on our $factorial$ function requiring that its argument be nonnegative as follows:

```
$factorial(n)$ requires $\{\, n \geq 0 \,\}$
  $=$ if $n = 0$ then $1$
    else $n\, factorial(n-1)$ end
```

We can also ensure that the result of $factorial$ is itself nonnegative:

```
$factorial(n)$
  requires $\{\, n \geq 0 \,\}$
  ensures $\{\, result \geq 0 \,\}$
  $=$ if $n = 0$ then $1$
    else $n\, factorial(n-1)$ end
```

The variable $result$ is bound in the `ensures` clause to the return value of the function.

## 2.6 Some Common Expressions in Fortress

We have already seen an `if` expression in Fortress. Here's an example of a `while` expression:

```
while $x < 10$ do
  $println\ x$
  $x \mathrel{+}= 1$
end
```

A sequence of expressions in Fortress may be delimited by `do` and `end`. Here is an example of a function that prints three words:

```
$printThreeWords() =$ do
  $print$ “print”
  $print$ “ three”
  $print$ “ words”
end
```

A *tuple* expression contains a sequence of elements delimited by parentheses and separated by commas:

```
(“this”, “is”, “a”, “tuple”, “of”, “mostly”, “strings”, 0)
```

When a tuple expression is evaluated, the various subexpressions are evaluated *in parallel*. For example, the following tuple expression denotes a parallel computation:

$(factorial(10), factorial(5), factorial(2))$

The elements in this expression may be evaluated in parallel. This same computation can be expressed as follows:

```
do
  $factorial(10)$
also do
  $factorial(5)$
also do
  $factorial(2)$
end
```

## 2.7 For Loops Are Parallel by Default

Here is an example of a simple `for` loop in Fortress:

```
for $i \leftarrow 1\!:\!10$ do
  $print(i$ “ ”$)$
end
```

This `for` loop iterates over all elements $i$ between $1$ and $10$ and prints the value of $i$. Expressions such as $1\!:\!10$ are referred to as *range expressions*. They can be used in any context where we wish to denote all the integers between a given pair of integers.

A significant difference between Fortress and most other programming languages is that *for loops are parallel by default*. Thus, printing in the various iterations of this loop can occur in an arbitrary order, such as:

5 4 6 3 7 2 9 10 1 8

## 2.8 Atomic Expressions

In order to control interactions of parallel executions, Fortress includes the notion of *atomic expressions*, as in the following example:

```
atomic do
   $x \mathrel{+}= 1$
   $y \mathrel{+}= 1$
end
```

An atomic expression is executed in such a manner that all other threads observe either that the computation has completed, or that it has not yet begun; no other thread observes an atomic expression to have only partially completed. Consider the following parallel computation:

```
do
   $x\colon \mathbb{Z}32 := 0$
   $y\colon \mathbb{Z}32 := 0$
   $z\colon \mathbb{Z}32 := 0$
   atomic do
      $x \mathrel{+}= 1$
      $y \mathrel{+}= 1$
   also atomic do
      $z := x + y$
   end
   $z$
end
```

Both parallel blocks are atomic; thus the second parallel block either observes that both $x$ and $y$ have been updated, or that neither has. Thus, possible values of the outermost `do` expression are 0 and 2, but not 1.

## 2.9 Aggregate Expressions

As with mathematical notation, Fortress includes special syntactic support for writing down many common kinds of collections, such as arrays, matrices, vectors, maps, sets, and lists simply by enumerating all of the collection's elements. We refer to an expression formed by enumerating the elements of a collection as an *aggregate expression*. The elements of an aggregate expression are computed in parallel.

For example, we can define an array $a$ in Fortress by explicitly writing down its elements, enclosed in brackets and separated by whitespace, as follows:

$$a : \mathbb{Z}32[5] = [0\ 1\ 2\ 3\ 4]$$

Two-dimensional arrays can be written down by separating rows by newlines (or by semicolons). For example, we can bind $b$ to a two-dimensional array as follows:

$$b : \mathbb{Z}32[2,2] = \begin{bmatrix} 3 & 4 \\ 5 & 6 \end{bmatrix}$$

There is also support for writing down arrays of dimension three and higher. We bind $c$ to a three-dimensional array as follows:

$$\begin{aligned} c\colon \mathbb{Z}32[2,2,3] = [\,&1\ 2\\ &3\ 4;;5\ 6\\ &\quad 7\ 8;;9\ 10\\ &\qquad 11\ 12] \end{aligned}$$

Various slices of the array along the third dimension are separated by pairs of semicolons. (Higher dimensional arrays are also supported. When writing a four-dimensional array, slices along the fourth dimension are separating by triples of semicolons, and so on.)

Vectors are written down just like one-dimensional arrays. Similarly, matrices are written down just like two-dimensional arrays. Of course, all elements of vectors and matrices must be numbers.

Note that there is a syntactic conflict between the use of juxtaposition to represent elements along a row in an array and the use of juxtaposition to represent applications of the juxtaposition operator. In order to include an application of the juxtaposition operator as an outermost subexpression of an array aggregate, it is necessary to include extra parentheses. For example, the following aggregate expression represents a counterclockwise rotation matrix by an angle $\theta$ in $\mathbb{R}^2$:

$$\begin{aligned} d\colon \mathbb{R}64[2,2] = [\,&(\cos\theta)(-\sin\theta)\\ &(\sin\theta)(\cos\theta)] \end{aligned}$$

A set can be written down by enclosing its elements in braces and separating its elements by commas. Here we bind $s$ to the set of integers $0$ through $4$:

$$s = \{0, 1, 2, 3, 4\}$$

The elements of a list are enclosed in angle brackets (written in ASCII as `<|` and `|>`):

$$l = \langle 0, 1, 2, 3, 4\rangle$$

The elements of a map are enclosed in curly braces, with key/value pairs joined by the arrow $\mapsto$ (written in ASCII as `|->`):

$$m = \{\,\text{“a”} \mapsto 0,\ \text{“b”} \mapsto 1,\ \text{“c”} \mapsto 2\,\}$$

## 2.10 Comprehensions

Another way in which Fortress mimics mathematical notation is in its support for *comprehensions*. Comprehensions describe the elements of a collection by providing a rule that holds for all of the collection's elements. The elements of the collection are computed in parallel by default. For example, we define a set $s$ that consists of all elements of another set $t$ divided by 2, as follows:

$$s = \{x/2 \mid x \leftarrow t\}$$

The expression to the left of the vertical bar explains that elements of $s$ consist of every value $x/2$ for every valid value of $x$ (determined by the right hand side). The expression to the right of the vertical bar explains how the elements $x$ are to be *generated* (in this case, from the set $t$). The right hand side of a comprehension can consist of multiple generators. For example, the following set consists of every element resulting from the sum of an element of $s$ with an element of $t$:

$$u = \{x + y \mid x \leftarrow s, y \leftarrow t\}$$

The right hand side of a comprehension can also contain *filtering expressions* to constrain the elements provided by other clauses. For example, we can stipulate that $v$ consists of all nonnegative elements of $t$ as follows:

$$v = \{x \mid x \leftarrow t, x \geq 0\}$$

As another example, here is a list comprehension in Fortress:

$l = \langle\, 2x \mid x \leftarrow v \,\rangle$

The elements of this list consist of all elements of the set $v$ multiplied by $2$.

## 2.11 Summations and Products

As with mathematical notation, Fortress provides syntactic support for summations and productions (and other big operations) over the elements of a collection. For example, an alternative definition of *factorial* is as follows:

$$\mathit{factorial}(n) = \prod_{i \leftarrow 1:n} i$$

This function definition can be written in ASCII as follows:

```
factorial(n) = PROD[i <- 1:n] i
```

The character $\prod$ is written `PROD`. Likewise, $\sum$ is written `SUM`. As with comprehensions, the values in the iteration are generated from specified collections.

## 2.12 Tests and Properties

Fortress includes support for automated program testing. New test in a component can be defined using the `test` modifier on a top-level function definition with type "$() \rightarrow ()$". For example, here is a test function that calls to the *factorial* function on some representative values result in values greater than or equal to what was provided:

```
test $\mathit{factorialResultLarger}() =$ do
    $\mathit{assert}(0 \leq \mathit{factorial}(0))$
    $\mathit{assert}(1 \leq \mathit{factorial}(1))$
    $\mathit{assert}(10 \leq \mathit{factorial}(10))$
    $\mathit{println}$ "Test factorialResultLarger succeeded."
end
```

## 2.13 Objects and Traits

A great deal of programming can be done simply through the use of functions, top-level variables, and standard types. However, Fortress also includes a trait and object system for defining new types, as well as objects that belong to them. Traits in Fortress exist in a multiple inheritance hierarchy. A trait declaration includes a set of method declarations, some of which may be abstract. For example, here is a declaration of a simple trait named Moving:

```
trait Moving extends { Tangible, Object }
    $\mathit{position}(): \mathbb{R}^3$
    $\mathit{velocity}(): \mathbb{R}^3$
end
```

The set of traits extended by trait Moving are listed in braces after `extends`. Trait Moving inherits all methods declared by each trait it extends. The two methods *position* and *velocity* declared in trait Moving are *abstract*; they contain no body. Their return types are vectors of length 3, whose elements are of types $\mathbb{R}$. As in mathematical notation, a vector of length $n$ with element type $T$ is written $T^n$.

Traits can also declare concrete methods, as in the following example:

```
trait Fast extends Moving
  velocity() = [0 0 299792458]
end
```

Trait $\mathrm{Fast}$ extends a single trait, $\mathrm{Moving}$. (Because it extends only one trait, we can elide the braces in its `extends` clause.) It inherits both abstract methods defined in $\mathrm{Moving}$, and it provides a concrete body for method $\mathit{velocity}$.

Trait declarations can be extended by other trait declarations, as well as by *object declarations*. There are two kinds of object declarations: *singleton declarations* and *constructor declarations*.

A singleton declaration declares a sole, stand-alone, *singleton object*. For example:

```
object Sol extends { Moving, Stellar }
  spectralClass = G_2
  position() = [0 0 0]
  velocity() = [0 0 0]
end
```

The object $\mathrm{Sol}$ extends two traits: $\mathrm{Moving}$ and $\mathrm{Stellar}$, and provides definitions for the abstract methods it inherits. Objects must provide concrete definitions for all abstract methods they inherit. $\mathrm{Sol}$ also defines a field $\mathit{spectralClass}$.

The self parameter of a method can be given a position other than the default position. For example, here is a definition of a type where the self parameters appear in nonstandard positions:

```
trait List
  cons(x: Object, self): List
  append(xs: List, self): List
end
```

In both methods $\mathit{cons}$ and $\mathit{append}$, the self parameter occurs as the second parameter. Calls to these methods look more like function calls than method calls. For example, in the following call to $\mathit{append}$, the receiver of the call is $l_2$:

$$\mathit{append}(l_1, l_2)$$

A constructor declaration declares an *object constructor*. In contrast to singleton declarations, a constructor declaration includes value parameters in its header, as in the following example:

```
object Particle(position: ℝ^3, velocity: ℝ^3)
  extends Moving
end
```

Every call to the constructor of $\mathrm{Particle}$ yields a new object. For example:

$$\mathit{pos}\colon \mathbb{R}^3 = [3\ 2\ 5]$$
$$\mathit{vel}\colon \mathbb{R}^3 = [1\ 0\ 0]$$
$$p_1 = \mathrm{Particle}(\mathit{pos}, \mathit{vel})$$

The parameters to an object constructor implicitly define fields of the object.

## 2.14 Features for Library Development

The language features introduced above are sufficient for the vast majority of applications programming. However, Fortress has also designed to be a good language for *library* programming. In fact, much of the Fortress language as viewed by applications programmers actually consists of code defined in libraries, written in a small core language. By defining as much of the language as possible in libraries, our aim is to allow the language to evolve gracefully as new demands are placed on it. In this section, we briefly mention some of the features that make Fortress a good language for library development.

### 2.14.1 Generic Types and Static Parameters

As in other languages, Fortress allows types to be parametric with respect to other types, as well as other "static" parameters, including integers, booleans, and operators. Fortress provides some standard parametric types, such as $\mathrm{Array}$ and $\mathrm{Vector}$. Programmers can also define new traits, objects, and functions that include static parameters.

### 2.14.2 Operator Overloading

Operators in Fortress can be overloaded with new definitions. Here is an alternative definition of the $factorial$ function, defined as a postfix operator:

$$\texttt{opr}\ (n{:}\,\mathbb{Z}32)! = \prod_{i \leftarrow 1:n} i$$

As with mathematical notation, Fortress allows operators to be defined as prefix, postfix, and infix. More exotic operators can even be defined as subscripting (i.e., applications of the operators look like subscripts), and as bracketing (i.e., applications of the operators look like the operands have been simply enclosed in brackets).

# Part II

# Fortress for Application Programmers

# Chapter 3

# Programs

A *program* consists of a finite sequence of Unicode 5.0 abstract characters. In order to more closely approximate mathematical notation, certain sequences of characters are rendered as subscripts or superscripts, italicized or boldface text, or text in special fonts, according to the rules in Appendix B. Although much of the program text in this specification is rendered as formatted Unicode, some text is presented unformatted to aid in exposition.

A program is *valid* if it satisfies all *static constraints* stipulated in this specification. Failure to satisfy a static constraint is a *static error*. Only valid programs can be *executed*; the validity of a program must be checked before it is executed.

Executing a valid Fortress program consists of *evaluating expressions*. Evaluation of an expression may modify the *program state* yielding a *result*. A result is either a *value*, or an *abrupt completion*.

The characters of a valid program determine a sequence of *input elements*. In turn, the input elements of a program determine the *program constructs*. Some program constructs may contain other program constructs. The two most common kinds of constructs in Fortress are *declarations* and *expressions*. We explain the structure of input elements, and of each program construct, in turn, along with accompanying static constraints. We also explain how the outcome of a program execution is determined from the sequence of constructs in the program.

Programs are developed and compiled as *components* as described in Chapter 20.

Fortress is block-structured: A Fortress program consists of nested *blocks* of code. The entire program is a single block. Each component is a block. Any top-level declaration is a block, as is any function declaration. Several expressions are also blocks, or have blocks as parts of the expression, or both (e.g., a `while` expression is a block, and its body is a different block). See Chapter 13 for a discussion of expressions in Fortress. In addition, a local declaration begins a block that continues to the end of the smallest enclosing block unless it is a local function declaration and is immediately preceded by another local function declaration. (This exception allows overloaded and mutually recursive local function declarations.) Because Fortress is block-structured, and because the entire program is a block, the smallest block that syntactically contains a program construct is always well defined.

Fortress is expression-oriented: What are often called "statements" in other languages are just expressions with type $()$ in Fortress. They do not evaluate to an interesting value, and are typically evaluated solely for their effects.

Fortress is whitespace-sensitive: Fortress has different contexts influencing the whitespace-sensitivity of expressions as described in Appendix D.

# Chapter 4

# Lexical Structure

A Fortress program consists of a finite sequence of Unicode 5.0 abstract characters [22]. Every character in a program is part of an input element. The partitioning of the character sequence into input elements is uniquely determined by the characters themselves. In this chapter, we explain how the sequence of input elements of a program is determined from a program's character sequence.

This chapter also describes standard ways to *render* (that is, *display*) individual input elements in order to approximate conventional mathematical notation more closely. Other rules, presented in later chapters, govern the rendering of certain *sequences* of input elements; for example, the sequence of three input elements $a$, ^ , and $b$ may be rendered $a^b$ . The rules of rendering are "merely" a convenience intended to make programs more readable. Alternatively, the reader may prefer to think of the rendered presentation of a program as its "true form" and to think of the underlying sequence of Unicode characters as "merely" a convenient way of encoding mathematical notation for keyboarding purposes.

Most of the program text in this specification is shown in rendered presentation form. However, sometimes, particularly in this chapter, unformatted code is presented to aid in exposition. In many cases the unformatted form is shown alongside the rendered form in a table, or following the rendered form in parentheses.

## 4.1 Characters

A Unicode 5.0 abstract character is the smallest element of a Fortress program. Many characters have standard *glyphs*, which are how these characters are most commonly depicted. However, more than one character may be represented by the same glyph. Thus, Unicode 5.0 specifies a representation for each character as a sequence of *code points*. Fortress programs are not permitted to contain characters that map to multiple code points, or to sequences of code points of length greater than 1. Thus, every character in a Fortress program is associated with a single code point, designated by a hexadecimal numeral preceded by "U+". Unicode also specifies a name for each character;[1] when introducing a character, we specify its code point and name, and sometimes the glyph we use to represent it in this specification. In some cases, we use such glyphs without explicitly introducing the characters (as, for example, with the simple upper- and lowercase letters of the Latin and Greek alphabets). When the character represented by a glyph is unclear and the distinction is important, we specify the code point or the name (or both). The Unicode Standard [22] specifies a *general category* for each character, which we use to describe sets of characters below.

We partition the Unicode 5.0 character set into the following (disjoint) classes:

[1]There are sixty-five "control characters", which do not have proper names. However, many of them have *Unicode 1.0 names*, or other standard names specified in the Unicode Character Database, which we use instead.

- *special non-operator characters*, which are:

| | | | | | |
|---|---|---|---|---|---|
| U+0026 | AMPERSAND | & | U+0027 | APOSTROPHE | ′ |
| U+0028 | LEFT PARENTHESIS | ( | U+0029 | RIGHT PARENTHESIS | ) |
| U+002C | COMMA | , | U+002E | FULL STOP | . |
| U+0038 | SEMICOLON | ; | U+005C | REVERSE SOLIDUS | \ |

- *special operator characters*, which are

| | | | | | |
|---|---|---|---|---|---|
| U+002A | ASTERISK | $*$ | U+002F | SOLIDUS | / |
| U+003A | COLON | : | U+003C | LESS-THAN SIGN | $<$ |
| U+003D | EQUALS SIGN | $=$ | U+003E | GREATER-THAN SIGN | $>$ |
| U+005B | LEFT SQUARE BRACKET | [ | U+005D | RIGHT SQUARE BRACKET | ] |
| U+005E | CIRCUMFLEX ACCENT | ˆ | U+007B | LEFT CURLY BRACKET | { |
| U+007C | VERTICAL LINE | \| | U+007D | RIGHT CURLY BRACKET | } |
| U+2192 | RIGHTWARDS ARROW | $\rightarrow$ | U+21A6 | RIGHTWARDS ARROW FROM BAR | $\mapsto$ |
| U+21D2 | RIGHTWARDS DOUBLE ARROW | $\Rightarrow$ | | | |

- *letters*, which are ASCII characters
- *digits*
- *whitespace characters*, which are:

| | | | |
|---|---|---|---|
| U+000A | LINE FEED | U+000C | FORM FEED |
| U+000D | CARRIAGE RETURN | U+0020 | SPACE |

- *character literal delimiters*, which are:

| | | |
|---|---|---|
| U+0060 | GRAVE ACCENT | \` |

- *string literal delimiters*, which are:

| | | |
|---|---|---|
| U+0022 | QUOTATION MARK | " |

- *ordinary operator characters*, enumerated (along with the special operator characters) in Appendix C, which include the following characters with code points less than U+007F:

| | | | | | |
|---|---|---|---|---|---|
| U+0021 | EXCLAMATION MARK | ! | U+0023 | NUMBER SIGN | # |
| U+0024 | DOLLAR SIGN | $ | U+0025 | PERCENT SIGN | % |
| U+002B | PLUS SIGN | $+$ | U+002D | HYPHEN-MINUS | $-$ |
| U+003F | QUESTION MARK | ? | U+0040 | COMMERCIAL AT | @ |
| U+007E | TILDE | ˜ | | | |

  and most Unicode characters specified to be *mathematical operators* (i.e., characters with code points in the range 2200-22FF) are operators in Fortress, but there are some exceptions (e.g., $\forall$, $\exists$ and $:=$).

- *other characters*

Some other classes of characters, which overlap with the ones above, are useful to distinguish:

- *ASCII characters* are those with code points U+007F and below;
- *word characters* are letters, digits, and apostrophe;
- *restricted-word characters* are ASCII letters, ASCII digits, and the underscore character (i.e., ASCII word characters other than apostrophe);
- *hexadecimal digits* are the digits and the ASCII letters `A`, `B`, `C`, `D`, `E`, `F`, `a`, `b`, `c`, `d`, `e` and `f`;
- *operator characters* are special operator characters and ordinary operator characters;

- *special characters* are special non-operator characters and special operator characters;
- *enclosing characters* are the enclosing operator characters enumerated in Section C.1, left and right parenthesis characters, and mathematical left and right white square brackets;

### Forbidden and Restricted Characters

It is a static error for the following characters to occur outside a comment:

| | | | |
|---|---|---|---|
| U+0009 | `CHARACTER TABULATION` | U+000B | `LINE TABULATION` |
| U+001C | `INFORMATION SEPARATOR FOUR` | U+001D | `INFORMATION SEPARATOR THREE` |
| U+001E | `INFORMATION SEPARATOR TWO` | U+001F | `INFORMATION SEPARATOR ONE` |

Thus, `LINE FEED`, `FORM FEED` and `CARRIAGE RETURN` are the only control characters—and the only whitespace characters other than spaces, `LINE SEPARATOR`, and `PARAGRAPH SEPARATOR`—outside of comments in a valid Fortress program.

## 4.2 Words and Chunks

In a sequence of characters, a *chunk* is a nonempty contiguous subsequence. A *word* is a maximal chunk consisting of only word characters (letters, digits, and apostrophe); that is, a word is one or more consecutive word characters delimited by characters other than word characters (or the beginning or end of the entire sequence). A *restricted word* is a maximal chunk that has only restricted-word characters (ASCII letters and digits and underscore characters). Note that a restricted word is *not* a word if it is delimited by word characters.

For example, in the sequence

$$A'\mathtt{bc}_{123}.def\ \phi_{456}$$

there are three words (`A'bc_123`, `def` and $\phi$`456`), and four restricted words (`A`, `bc_123`, `def` and `456`).

## 4.3 Lines, Pages and Position

The characters in a Fortress program are partitioned into *lines* and into *pages*, delimited by *line terminators* and *page terminators* respectively. A *page terminator* is an occurrence of the character `FORM_FEED`. A *line terminator* is an occurrence of any of the following:

- `LINE FEED` or
- `CARRIAGE RETURN` not immediately followed by `LINE FEED`.

If a character (or any other syntactic entity) $x$ precedes another character $y$ in the program, we say that $x$ is *to the left of* $y$ and that $y$ is *to the right of* $x$, regardless of how they may appear in a typical rendered display of the program. Thus, it is always meaningful to speak, for example, of the left-hand and right-hand operands of a binary operator, or the left-hand side of an assignment expression.

## 4.4 Input Elements and Scanning

A Fortress program is partitioned into *input elements* by a process called *scanning*. Scanning transforms a Fortress program from a sequence of Unicode characters to a sequence of input elements. Input elements are always chunks:

the characters that comprise an input element always appear contiguously in the program. Input elements are either *whitespace elements* (including *comments*) or *tokens*. A token is a *reserved word*, a *literal*, an *identifier*, an *operator token*, or a *special token*. There are five kinds of literals: boolean literals, character literals, string literals, the void literal, and numerals (i.e., numeric literals).

Conceptually, we can think of scanning as follows: First, the comments, character literals and string literals are identified. Then the remaining characters are partitioned into words (i.e., maximal chunks of letters, digits, primes and apostrophes), whitespace characters, and other characters. In some cases, words separated by a single ' . ' are joined to form a single numeral (see Section 4.12). Words that are not so joined are classified as reserved words, boolean literals, numerals, identifiers, or operator tokens, as described in later sections in this chapter. It is a static error if any word in a Fortress program is not part of one of the input elements described above. All remaining whitespace characters, together with the comments, form whitespace elements, which may be *line-breaking*. Finally, chunks of symbols (and a few other special cases) are checked to see whether they form void literals (see Section 4.11) or multicharacter operator tokens (see Section 4.13). Every other character is a token by itself, either a special token (if it is a special character) or an operator token.

## 4.5 Comments

When not within string literals, occurrences of " $(*$ " and " $*)$ " are *opening comment delimiters* and *closing comment delimiters* respectively. In a valid program, every opening comment delimiter is balanced by a closing comment delimiter; it is a static error if comment delimiters are not properly balanced. All the characters between a balanced pair of comment delimiters, including the comment delimiters themselves, comprise a *comment*. Comments may be nested. For example, the following illustrates three comments, one of which is nested:

```
(* This is a comment. *) (* As is this (* nested *)
comment *)
```

## 4.6 Whitespace Elements

A whitespace element is a maximal chunk consisting of comments and whitespace characters that are not within string or character literals.

We distinguish *line-breaking whitespace* from *non-line-breaking whitespace* using the following terminology:

- A *line-terminating comment* is a comment that encloses one or more line terminators. All other comments are called *spacing comments*.
- *Spacing* refers to any chunk of spaces, `FORM FEED` characters and spacing comments.
- A *line break* is a line terminator possibly with intervening spacing.
- *Whitespace* refers to any nonempty sequence of spacing, line terminators, and line-terminating comments.
- *Line-breaking whitespace* is whitespace that contains at least one line break.

## 4.7 Reserved Words

The following tokens are *reserved words*:

| | | | | | | |
|---|---|---|---|---|---|---|
| `BIG` | `FORALL` | `SI_unit` | `absorbs` | `abstract` | `also` | `api` |
| `as` | `asif` | `at` | `atomic` | `bool` | `case` | `catch` |
| `coerce` | `coerces` | `component` | `comprises` | `default` | `dim` | `do` |
| `elif` | `else` | `end` | `ensures` | `except` | `excludes` | `exit` |
| `export` | `extends` | `finally` | `fn` | `for` | `forbid` | `from` |
| `getter` | `hidden` | `if` | `import` | `int` | `invariant` | `io` |
| `juxtaposition` | `label` | `most` | `nat` | `native` | `object` | `of` |
| `opr` | `or` | `override` | `private` | `property` | `provided` | `requires` |
| `self` | `settable` | `setter` | `spawn` | `syntax` | `test` | `then` |
| `throw` | `throws` | `trait` | `transient` | `try` | `tryatomic` | `type` |
| `typecase` | `unit` | `value` | `var` | `where` | `while` | `widens` |
| `with` | `wrapped` | | | | | |

To avoid confusion, Fortress reserves the following tokens:

```
goto   idiom   public   pure   reciprocal   static
```

They do not have any special meanings but they cannot be used as identifiers.

## 4.8 Character Literals

A character literal consists of one character enclosed in single quotation marks.

## 4.9 String Literals

A string literal is a sequence of characters enclosed in double quotation marks (for example, `"Hello, world!"`). As discussed in Section 13.1, a string literal evaluates to a value of type String, which represents a finite sequence of characters.

## 4.10 Boolean Literals

The boolean literals are $\mathit{false}$ and $\mathit{true}$.

## 4.11 The Void Literal

The void literal is $()$ (pronounced “void”).

## 4.12 Numerals

A numeric literal, or *numeral*, in Fortress is a maximal chunk consisting one or more words (and the intervening characters) satisfying the following properties:

- each word consists of only digits and letters;
- consecutive words are separated by exactly one character, a ‘ . ’ character; and

- the first word begins with a digit.

A numeral is *simple* if it does not contain a '.' character; otherwise, the number is *compound*. It is a static error if a numeral contains more than one '.' character.

## 4.13 Operator Tokens

In this section, we describe how to determine the operator tokens of a program. Because operator tokens do not occur within comments, string literals and character literals, we henceforth in this section consider only characters that are outside these constructs.

An *operator word* of a program is a word that is not reserved, consists only of uppercase letters and underscores (no digits or non-uppercase letters), does not begin or end with an underscore, and has at least two different letters.

A *base operator* is an ordinary operator character,[2] an operator word, a (contiguous) sequence of two or more vertical-line characters (U+007C), or a multicharacter enclosing operator, as defined in Section 4.13.1. A base operator is *maximal* in a program if it is not contained within any other base operator of that program. It is a static error if two maximal base operators overlap (which is only possible if both are multicharacter enclosing operators). A *simple operator* is a maximal base operator that is an operator character, or an operator word.

A maximal base operator is an operator token unless it is a simple operator other than an enclosing or vertical-line operator character, and it is immediately preceded by '^' or immediately followed by '='. Such an operator is an *enclosing operator* if it is an enclosing operator character (see Section C.1) or a multicharacter enclosing operator; it is a *vertical-line operator* if it has only vertical-line operator characters (see Section C.2); otherwise, it is an *ordinary operator*.

If a simple operator is immediately preceded by '^' then the '^' and the simple operator together comprise a single operator token; such an operator token is called a *superscripted postfix operator*. In addition, "$\hat{}T$" is also a superscripted postfix operator provided that it is not immediately followed by a word character. It is a static error for a superscripted postfix operator to be immediately followed by a word character.

Finally, if a simple operator is not immediately preceded by '^' and is immediately followed by '=' then the simple operator and the '='together comprise an operator token; such an operator token is called a *compound assignment operator*.

### 4.13.1 Multicharacter Enclosing Operators

The following multicharacter sequences (in which there must be no other characters, and particularly no whitespace) can be used as brackets as described below:

1. Any chunk of vertical-line characters is a vertical-line operator. Such an operator can be used in an enclosing pair matching itself.

2. Any of '(' or '[' or '{' may be immediately followed by any number of '/' characters or by any number of '\' characters. Such a token is a left bracket, and it is matched by the multicharacter token consisting of the same number and kind of '/' or '\' characters followed immediately by a matching ')' or ']' or '}', as appropriate. Thus, for example, "`(////`" and "`////)`" are matching left and right brackets respectively. (In the future, we may allow tokens with mixtures of '/' and '\', in which case the left and right brackets must match from outside in. But for now, such tokens are simply illegal.)

[2]The ordinary operator characters are enumerated in Appendix C, less the special operator characters listed in Section 4.1.

3. One or more '<' characters may be followed immediately by one or more '|' characters. Such a token is a left bracket, and it is matched by the multicharacter token consisting of the same number of '|' characters followed by as many '>' characters as there are '<' characters in the left bracket.
4. One or more '<' characters, or one or more '|' characters may be followed immediately by one or more '/' characters or by one or more '\' characters. Such a token is a left bracket, and it is matched by the multicharacter token consisting of the same number and *opposite* kind of '/' or '\' characters followed immediately by as many matching '>' or '|' characters as appropriate. Thus, for example, "<<//" matches "\\>>', and "|\\\" matches "///|". (As in case 2 above, we may allow tokens with mixtures of '/' and '\' in the future.)
5. Finally, any number of '*' (U+002A) or '.' (U+002E) characters may be placed within any of the above multicharacter sequences, except those that contain '(' or ')', as long as no '*' or '.' is the first or last character in the sequence, and no '*' or '.' characters are adjacent. The rule for matching is as above, except that in addition, the positions of the '*' and the '.' characters must match from the outside in.

### 4.13.2 Special Operators

Note that in the preceding discussion, a single operator character can be an operator token only if it is an ordinary operator character. In some cases, some of the special operator characters (and even some special non-operator characters) form part of an operator token. However, most of the special operator characters cannot be determined to be operators before parsing the program because they are also used for various parts of Fortress syntax. The one exception is '^': if an occurrence of this character is not part of a superscripted postfix operator, then it is an operator token by itself: the special *superscripting operator*. This operator is always an infix operator, and it is a static error if it appears in a context in which a prefix or postfix operator is expected.

Every other special operator character, when not part of an operator token, is a special token that may be used as an operator. There is also a special `juxtaposition` operator (described in Section 16.8), which is also always infix, but this operator is a reserved word rather than an operator token. Occurrences of this operator are determined by the Fortress grammar.

## 4.14 Identifiers

A word is an identifier if it begins with a letter or an underscore and is not a reserved word, an operator, or all or part of a numeral.

## 4.15 Special Tokens

Every special character (operator or non-operator) that is not part of a token (or within a comment) as described above is a *special token* by itself. The special operator characters may be operators in the appropriate context.

## 4.16 Rendering of Fortress Programs

In order to more closely approximate mathematical notation, Fortress mandates standard rendering for various input elements, particularly for numerals and identifiers, as specified in this section. In the remainder of this specification, programs are presented formatted unless stated otherwise.

### 4.16.1 Fonts

Throughout this section, we refer to different fonts or styles in which certain characters are rendered, with names suggestive of their appearance.

- roman
- italic
- math (often identical to italic)
- script
- fraktur
- sans-serif
- italic sans-serif
- monospace
- double-struck

Additionally, the following fonts may be specified to be bold: roman, italic, script, fraktur, sans-serif, italic sans-serif.

However, a particular environment may substitute different fonts either because of local practice or because the desired fonts are not available.

### 4.16.2 Numerals

A numeral is rendered in roman type.

|  |  |  |
|---|---|---|
| `27` | *is rendered as* | 27 |
| `3.143159265` | *is rendered as* | 3.143159265 |

Note: the elegant way to write Avogadro's number is `6.02 TIMES 10^23`, which is not a single token but is a constant expression; its rendered form is $6.02 \times 10^{23}$.

### 4.16.3 Identifiers

Fortress has rather complicated rules for rendering an identifier; as in other parts of Fortress, the rules are complicated so that the simple cases will be very simple, but also so that difficult cases of interest will be possible.

It is conventional in mathematical notation to make use of variables, particularly single-letter variables, in a number of different fonts or styles, with italic being the most common, then boldface, and roman: $a$, $\mathbf{b}$, $\mathrm{c}$. Frequently such variables are also decorated with accents and subscripts: $\bar{p}$, $q'$, $\hat{r}$, $T_{\mathrm{max}}$, $\vec{\mathbf{u}}$, $\mathbf{v}_{\mathrm{x}}$, $w_{17}$, $\bar{z}'_{17}$. Fortress provides conventions for typing such variables using plain ASCII characters: for example, the unformatted presentations of these same variables are `a`, `_b`, `c_`, `p_bar`, `q'` or `q_prime`, `r_hat`, `T_max`, `_u_vec`, `_v_x`, `w17`, and `z17_bar'`. The rules are also intended to accommodate the typical use of multicharacter variable names for computer programming, such as $\mathit{count}$, $\mathit{isUpperCase}$, and $\mathrm{Boolean}$.

The most important rules of thumb are that simple variables are usually italic $z$ (`z`), a leading underscore usually means boldface font $\mathbf{z}$ (`_z`), a trailing underscore usually means roman font $\mathrm{z}$ (`z_`), and a doubled capital letter means double-struck (or "blackboard bold") font $\mathbb{Z}$ (`ZZ`). However, mixed-case variable names that begin with a capital letter, which are usually used as names of types, are rendered in roman font even if there is no trailing underscore.

The detailed rules are described in Section B.1.

### 4.16.4 Other Rendering Rules

Reserved words are rendered in monospace. Operator words are rendered in monospace. Comments are rendered in roman font. Any character within a character literal or string literal is rendered in monospace if possible.

The detailed rules for rendering other constructions are described in Section B.2.

# Chapter 5

# Evaluation

The state of an executing Fortress program consists of a set of *threads*, and a *memory*. Communication with the outside world is accomplished through *input and output actions*. In this chapter, we give a general overview of the execution process and introduce terminology used throughout this specification to describe the behavior of Fortress constructs.

Executing a Fortress program consists of *evaluating* the body expression of the $run$ function and the initial-value expressions of all top-level variables and singleton object fields in parallel. All initializations must complete normally before any reference to the objects or variables being initialized.

Threads evaluate expressions by taking *steps*. We say evaluation of an expression *begins* when the first step is taken. A step may *complete* the evaluation, in which case no more steps are possible on that expression, or it may result in an *intermediate expression*, which requires further evaluation. *Dynamic program order* is a partial order among the expressions evaluated in a particular execution of a program. When two expressions are ordered by dynamic program order, the first must complete execution before any step can be taken in the second. Chapter 13 gives the dynamic program order constraints for each expression in the language.

Intermediate expressions are generalizations of ordinary Fortress expressions: some intermediate expressions cannot be written in programs. We say that one expression is *dynamically contained* within a second expression if all steps taken in evaluating the first expression must be taken between the beginning and completion of the second. A step may also have effects on the program state beyond the thread taking the step, for example, by modifying one or more locations in memory, creating new threads to evaluate other expressions, or performing an input or output action. Threads are discussed further in Section 5.4. The memory consists of a set of *locations*, which can be *read* and *written*. New locations may also be *allocated*. The memory is discussed further in Section 5.3. Finally, *input actions* and *output actions* are described in Section 5.6.

## 5.1 Values

A *value* is the result of normal completion of the evaluation of an expression. (See Section 5.2 for a discussion of completion of evaluation.) A value is an object, a tuple or the void value $()$. Every value has a *type*, and every object has an environment (see Section 5.5). See Chapter 6 for a description of the types corresponding to these different values. An object may be a *value object*, a *reference object*, or a *function*. A nonfunction object has a finite set of *fields*; the names and types of these fields are specified by the type of the object. The type of a nonfunction object also specifies its methods.

A field consists of a name, a type, and either a value or a location: in value objects, fields have values; in reference objects, locations. The name of a field is either an identifier or an index. Every field in a value object is *immutable*. Reference objects may have both *mutable* and *immutable* fields. No two distinct values share any mutable field.

Values are constructed:

1. by top-level function declarations (see Section 9.1) and singleton declarations (see Section 11.1), and
2. by evaluating an object expression (see Section 13.9), a function expression (see Section 13.8), a local function declaration (see Section 9.4), a call to an object constructor (declared by a constructor declaration; see Chapter 11), a literal (see Section 13.1), a `spawn` expression (see Section 13.23), a tuple expression (see Section 13.26), an aggregate expression (see Section 13.27), or a comprehension (see Section 13.28).

In the latter case, the constructed value is the result of the normal completion of such an evaluation.

## 5.2 Normal and Abrupt Completion of Evaluation

Conceptually, an expression is evaluated until it *completes*. Evaluation of an expression may *complete normally*, resulting in a value, or it may *complete abruptly*. Each abrupt completion has an associated value, either an exception value that is thrown and uncaught or the exit value of an `exit` expression (described in Section 13.12). In addition to programmer-defined exceptions thrown explicitly by a `throw` expression (described in Section 13.24), there are predefined exceptions thrown by the Fortress standard libraries. For example, dividing an integer by zero (using the $/$ operator) causes a $\mathrm{DivisionByZero}$ to be thrown.

When an expression completes abruptly, control passes to the dynamically immediately enclosing expression. This continues until the abrupt completion is handled either by a `try` expression (described in Section 13.25) if an exception is being thrown or by an appropriately tagged `label` expression (described in Section 13.12) if an `exit` expression was evaluated. If abrupt completion is not handled within a thread and its outermost expression completes abruptly, the thread itself completes abruptly. If the main thread of a program completes abruptly, the program as a whole also completes abruptly.

## 5.3 Memory and Memory Operations

In this specification, the term *memory* refers to a set of abstract *locations*; the memory is used to model sharing and mutation. A location has an associated type, and *contains* a value of that type (i.e., the type of the value is a subtype of the type of the location). Locations can have non-object trait types; by contrast, a value always has an object type.

There are three operations that can be performed on memory: *allocation*, *read*, and *write*. Reads and writes are collectively called *memory accesses*. Intuitively, a read operation takes a location and returns the value contained in that location, and a write operation takes a location and a value of the location's type and changes the contents of the location so the location contains the given value. Accesses need not take place in the order in which they occur in the program text; a detailed account of memory behavior appears in Chapter 19.

Allocation creates a new location of a given type. Allocation occurs when a mutable variable is declared, or when a reference object is constructed. In the latter case, a new location is allocated for each field of the object. Abstractly, locations are never reclaimed; in practice, memory reclamation is handled by garbage collection.

A freshly allocated location is *uninitialized*. Any location whose value can be written after initialization is *mutable*. Mutable locations consist of mutable variables and mutable fields of reference objects. Any location whose value cannot be written after initialization is *immutable*. Immutable locations are the immutable fields of a reference object.

## 5.4 Threads and Parallelism

There are two kinds of threads in Fortress: *implicit* threads and *spawned* (or *explicit*) threads. Spawned threads are objects created by the `spawn` construct, described in Section 13.23.

A thread may be in one of five states: *not started*, *executing*, *suspended*, *normally completed* or *abruptly completed*. We say a thread is *not started* after it is created but before it has taken a step. This is only important for purposes of determining whether two threads are executing simultaneously; see Section 21.3. A thread is *executing* or *suspended* if it has taken a step, but has not completed; see below for the distinction between executing and suspended threads. A thread is *complete* if its expression has completed evaluation. If the expression completes normally, its value is the result of the thread.

Every thread has a *body* and an *execution environment*. The body is an intermediate expression, which the thread evaluates in the context of the execution environment; both the body and the execution environment may change when the thread takes a step. This environment is used to look up names in scope but bound in a block that encloses the construct that created the thread. The execution environment of a newly created thread is the environment of the thread that created the new thread.

In Fortress, a number of constructs are *implicitly parallel*. An implicitly parallel construct creates a *group* of one or more implicit threads. The implicitly parallel constructs are:

- **Tuple expressions:** Each element expression of the tuple expression is evaluated in a separate implicit thread (see Section 13.26).
- `also do` **blocks:** each sub-block separated by an `also` clause is evaluated in a separate implicit thread (see Section 13.11.3).
- **Method invocations and function calls:** The receiver or the function and each of the arguments is evaluated in a separate implicit thread (see Section 13.4, Section 13.5, and Section 13.6).
- `for` **loops, comprehensions, sums, generated expressions, and big operators:** Parallelism in looping constructs is specified by the generators used (see Section 13.14). Programmers must assume that generators that do not extend $\mathrm{SequentialGenerator}$ can execute each iteration of the body expression in a separate implicit thread. The functional method $seq$ and the equivalent function $sequential$ can be used to turn any $\mathrm{Generator}$ into a $\mathrm{SequentialGenerator}$.
- **Extremum expressions:** Each guarding expression of the extremum expression is evaluated in a separate implicit thread (see Section 13.20).
- **Tests:** Each test is evaluated in a separate implicit thread (see Chapter 17).

Implicit threads run fork-join style: all threads in a group are created together, and they all must complete before the group as a whole completes. There is no way for a programmer to single out an implicit thread and operate upon it in any way; they are managed purely by the compiler, runtime, and libraries of the Fortress implementation. Implicit threads need not be scheduled fairly; indeed, a Fortress implementation may choose to serialize portions of any group of implicit threads, interleaving their operations in any way it likes. For example, the following code fragment may loop forever:

$r\colon \mathbb{Z}64 = 0$
$(r := 1, \texttt{while } r = 0 \texttt{ do end})$

If any implicit thread in a group completes abruptly, the group as a whole will complete abruptly as well. Every thread in the group will either not run at all, complete (normally or abruptly), or may be terminated early as described in Section 21.6. The result of the abrupt completion of the group is the result of one of the constituent threads that completes abruptly.

Spawned thread objects are reference objects of type $\mathrm{Thread}[\![T]\!]$, where $T$ is the static type of the expression spawned; this trait has the following methods, each taking no arguments:

- The $val$ method returns the value computed by the subexpression of the `spawn` expression. If the thread has not yet completed execution, the invocation of $val$ blocks until it has done so.
- The $wait$ method waits for a thread to complete, but does not return a value.
- The $ready$ method returns $true$ if a thread has completed, and returns $false$ otherwise.
- The $stop$ method attempts to terminate a thread as described in Section 21.6.

We say a spawned thread has been *observed to complete* after invoking the $val$ or $wait$ methods, or when an invocation of the $ready$ method returns $true$.

In the absence of sufficient parallel resources, an attempt is made to run the subexpression of the `spawn` expression before continuing succeeding evaluation. We can imagine that it is actually the *rest* of the evaluation *after* the parallel block which is spawned off in parallel. This is a subtle technical point, but makes the sequential execution of parallel code simpler to understand, and avoids subtle problems with the asymptotic stack usage of parallel code [17, 6].

There are three ways in which a thread can be suspended. First, a thread that begins evaluating an implicitly parallel construct, creating a thread group, is suspended until that thread group has completed. Second, a thread that invokes $val$ or $wait$ is suspended until the spawned thread completes. Finally, invoking the $abort$ function from within an `atomic` expression may cause a thread to suspend; see Section 21.5.

Threads in a Fortress program can perform operations simultaneously on shared objects. In order to synchronize data accesses, Fortress provides `atomic` expressions (see Section 13.22). Chapter 19 describes the memory model which is obeyed by Fortress programs.

## 5.5 Environments

An *environment* maps *names* to values or locations. Environments are immutable, and two environments that map exactly the same names to the same values or locations are identical.

A program starts executing with an empty environment. Environments are extended with new mappings by variable, function, and object declarations, and functional calls (including calls to object constructors). After initializing all top-level variables and singleton objects, the *top-level environment* for each component is constructed.

The environment of a value is determined by how it is *constructed*. For all but object expressions, function expressions and local function declarations, the environment of the constructed value is the top-level environment of the component in which the expression or declaration occurs. For object and function expressions and local function declarations, the environment of the constructed value is the lexical environment in which the expression or declaration was evaluated.

We carefully distinguish a spawned thread from its associated spawned thread object. In particular, note that the execution environment of a spawned thread, in which the body expression is evaluated, is distinct from the environment of the associated thread object, in which calls to the thread methods are evaluated.

## 5.6 Input and Output Actions

Certain functionals in Fortress perform primitive input/output (I/O) actions. These actions have an externally visible effect.

Any primitive I/O action may take many internal steps; each step may read or write any memory locations referred to either directly or transitively by object references passed as arguments to the action. Each I/O action is free to complete either normally or abruptly. I/O actions may block and be prevented from taking a step until any necessary external conditions are fulfilled (input is available, data has been written to disk, and so forth).

Each I/O action taken by an expression is considered part of that expression's effects. The steps taken by an I/O action are considered part of the context in which an expression evaluates, in much the same way as effects of simultaneously-executing threads must be considered when describing the behavior of an expression. For example, we cannot consider two functionals to be equivalent unless the possible I/O actions they take are the same, given identical internal steps by each I/O action.

# Chapter 6

# Types

The Fortress type system supports a mixture of nominal and structural types. Specifically, Fortress provides nominal trait types, which are defined in programs (often in libraries), and a few constructs that combine types structurally.

Every expression has a *static type*, and every value has a *runtime type* (also known as a *dynamic type*). Sometimes we abuse terminology by saying that an expression has the runtime type of the value it evaluates to (see Section 5.2 for a discussion about evaluation of expressions).

Some types may be parameterized by *static parameters*; we call these types *generic types*. See Chapter 12 for a discussion of static parameters. A static parameter may be instantiated with a type or a value depending on whether it is a type parameter. Two types are identical if and only if they are the same kind and their names and static arguments (if any) are identical. Types are related by several relationships as described in Section 6.2.

Syntactically, the positions in which a type may legally appear (i.e., *type contexts*) is determined by the nonterminal *Type* in the Fortress grammar, defined in Appendix D.

## 6.1 Kinds of Types

Fortress supports the following kinds of types:

- trait types,
- object expression types,
- tuple types,
- arrow types,
- function types, and
- three special types: $\mathrm{Any}$, $\mathrm{BottomType}$ and $()$.

Object expression types, function types and $\mathrm{BottomType}$ are not first-class types: they cannot be written in a program. However, some values have object expression types and function types and `throw` and `exit` expressions have the type $\mathrm{BottomType}$.

Collectively, trait types and object expression types are *defined* types; the trait and object declarations and object expressions are the types' *definitions*.

## 6.2 Relationships between Types

We define two relations on types: *subtype* and *exclusion*.

The subtype relation is a partial order on types that defines the type hierarchy; that is, it is reflexive, transitive and antisymmetric. $\mathrm{Any}$ is the top of the type hierarchy: all types are subtypes of $\mathrm{Any}$. $\mathrm{BottomType}$ is the bottom of the type hierarchy: all types are supertypes of $\mathrm{BottomType}$. For types $T$ and $U$, we write $T \preceq U$ when $T$ is a subtype of $U$, and $T \prec U$ when $T \preceq U$ and $T \neq U$; in the latter case, we say $T$ is a *strict* subtype of $U$. We also say that $T$ is a *supertype* of $U$ if $U$ is a subtype of $T$. We say that a value is *an instance of* its runtime type and of every supertype of its runtime type.

The exclusion relation is a symmetric relation between two types whose intersection is $\mathrm{BottomType}$. Because $\mathrm{BottomType}$ is uninhabited (i.e., no value has type $\mathrm{BottomType}$), types that exclude each other are disjoint: no value can have a type that is a subtype of two types that exclude each other. Note that $\mathrm{BottomType}$ excludes every type including itself (because the intersection of $\mathrm{BottomType}$ and any type is $\mathrm{BottomType}$), and also that no type other than $\mathrm{BottomType}$ excludes $\mathrm{Any}$ (because the intersection of any type $T$ and $\mathrm{Any}$ is $T$). $\mathrm{BottomType}$ is the only type that excludes itself. If two types exclude each other then any subtypes of these types also exclude each other.

These relations are defined more precisely in the following sections describing each kind of type in more detail. Specifically, the relations are the smallest ones that satisfy all the properties given in those sections (and this one).

## 6.3 Trait Types

Syntax:

*Type* ::= *TraitType*

A *trait type* is a named type defined by a trait or object declaration. It is an *object trait type* if it is defined by an object declaration.

A trait type is a subtype of every type that appears in the `extends` clause of its definition. In addition, every trait type is a subtype of $\mathrm{Any}$.

A trait declaration may also include an `excludes` clause, in which case the defined trait type excludes every type that appears in that clause. Every trait type also excludes every arrow type and every tuple type, and the special types $()$ and $\mathrm{BottomType}$.

### 6.3.1 Object Trait Types

An object trait type is a trait type defined by an object declaration. Object declarations do not include `excludes` clauses, but an object trait type cannot be extended. Thus, an object trait type excludes any type that is not its supertype.

## 6.4 Object Expression Types

An object expression type is defined by an object expression and the program location of the object expression. Every evaluation of a given object expression has the same object expression type. Two object expressions at different program locations have different object expression types.

Like trait types, an object expression type is a subtype of all trait types that appear in the `extends` clause of its definition, as well as the type $\mathrm{Any}$.

Like object trait types, an object expression type excludes any type that is not its supertype.

## 6.5 Tuple Types

Syntax:

| | | |
|---|---|---|
| *TupleType* | ::= | ( *Type* , *TypeList* ) |
| *TypeList* | ::= | *Type*( , *Type*)* |

A tuple type consists of a parenthesized, comma-separated list of two or more types.

Every tuple type is a subtype of $\mathrm{Any}$. No other type encompasses all tuple types. Tuple types cannot be extended by trait types. Tuple types are covariant: a tuple type $X$ is a subtype of tuple type $Y$ if and only if they have the same number of element types and each element type of $X$ is a subtype of the corresponding element type of $Y$.

A tuple type excludes any non-tuple type other than $\mathrm{Any}$. A tuple type excludes every tuple type that does not have the same number of element types. Also, tuple types that have the same number of element types exclude each other if any pair of corresponding element types exclude each other.

## 6.6 Arrow Types

Syntax:

| | | |
|---|---|---|
| *Type* | ::= | *ArgType* → *Type Throws*? |
| *ArgType* | ::= | ( (*Type* , )* *Type* ... ) |
| | \| | *TupleType* |

The static type of an expression that evaluates to a function value is an *arrow type* (or possibly an intersection of arrow types). An arrow type has three constituent parts: a *parameter type*, a *return type*, and a set of *exception types*. (Multiple parameters or return values are handled by having tuple types for the parameter type or return type respectively.) Syntactically, an arrow type consists of the parameter type followed by the token →, followed by the return type, and optionally a `throws` clause that specifies the set of exception types. If there is no `throws` clause, the set of exception types is empty.

The parameter type of an arrow type may end with a "varargs entry", $T\ldots$. The resulting parameter type is not a first-class type; rather, it is the union of all the types that result from replacing the varargs entry with zero or more entries of the type $T$. For example, $(T\ldots)$ is the union of $()$, $T$, $(T,T)$, $(T,T,T)$, and so on.

We say that an arrow type is *applicable* to a type $A$ if $A$ is a subtype of the parameter type of the arrow type. (If the parameter type has a varargs entry, then $A$ must be a subtype of one of the types in the union defined by the parameter type.)

Every arrow type is a subtype of $\mathrm{Any}$. No other type encompasses all arrow types. A type parameter can be instantiated with an arrow type. Arrow types cannot be extended by trait types. Arrow types are covariant in their return type and exception types and contravariant in their parameter type; that is, arrow type "$A \rightarrow B$ `throws` $C$" is a subtype of arrow type "$D \rightarrow E$ `throws` $F$" if and only if:

- $D$ is a subtype of $A$,
- $B$ is a subtype of $E$, and
- for all $X$ in $C$, there exists $Y$ in $F$ such that $X$ is a subtype of $Y$.

For arrow types $S$ and $T$, we say that $S$ is *more specific than* $T$ if the parameter type of $S$ is a subtype of the parameter type of $T$. We also say that $S$ is *more restricted than* $T$ if the return type of $S$ is a subtype of the return type of $T$ and for every $X$ in the set of exception types of $S$, there exists $Y$ in the set of exception types of $T$ such that $X$ is a subtype of $Y$. Thus, $S$ is a subtype of $T$ if and only if $T$ is more specific than $S$ and $S$ is more restricted than $T$.

An arrow type excludes any non-arrow type other than $\mathrm{Any}$ and the function types that are its subtypes (see Section 6.7). However, arrow types do not exclude other arrow types because of overloading as described in Chapter 15.

Here are some examples:

$f\colon (\mathbb{R}64, \mathbb{R}64) \rightarrow \mathbb{R}64$
$g\colon \mathbb{Z}32 \rightarrow (\mathbb{Z}32, \mathbb{Z}32)$ `throws` $\mathrm{IOFailure}$

## 6.7 Function Types

Function types are the runtime types of function values. We distinguish them from arrow types to handle overloaded functions. A function type consists of a finite set of arrow types. However, not every set of arrow types is a well-formed function type. Rather, a function type $F$ is *well-formed* if, for every pair $(S_1, S_2)$ of distinct arrow types in $F$, the following properties hold:

- the parameter types of $S_1$ and $S_2$ are not the same,
- if $S_1$ is more specific than $S_2$ then $S_1$ is more restricted than $S_2$, and
- if the intersection of the parameter types of $S_1$ and $S_2$ is not $\mathrm{BottomType}$ (i.e., the parameter types of $S_1$ and $S_2$ do not exclude each other) then $F$ has some constituent arrow type that is more specific than both $S_1$ and $S_2$ (recall that the more specific relation is reflexive, so the required constituent type may be $S_1$ or $S_2$).

Henceforth, we consider only well-formed function types. The overloading rules ensure that all function values have well-formed function types.

We say that a function type $F$ is *applicable* to a type $A$ if any of its constituent arrow types is applicable to $A$. We extend this terminology to values of the corresponding types. That is, for example, we say that a function of type $F$ is applicable to a value of type $A$ if $F$ is applicable to $A$. Note that if a well-formed function type $F$ is applicable to a type $A$, then among all constituent arrow types of $F$ that are applicable to $A$ (and there must be at least one), one is more specific than all the others. We say that this constituent type is the *most specific* type of $F$ applicable to $A$.

A function type is a subtype of each of its constituent arrow types, and also of $\mathrm{Any}$. Like object trait types and object expression types, a function type excludes every type that is not its supertype.

## 6.8 Special Types

Fortress provides three special types: $\mathrm{Any}$, $\mathrm{BottomType}$ and $()$. The type $\mathrm{Any}$ is the top of the type hierarchy: it is a supertype of every type. The only type it excludes is $\mathrm{BottomType}$.

The type $()$ is the type of the value $()$. Its only supertype (other than itself) is $\mathrm{Any}$, and it excludes every other type.

Fortress provides a special *bottom type*, $\mathrm{BottomType}$, which is an uninhabited type. No value in Fortress has $\mathrm{BottomType}$; `throw` and `exit` expressions have $\mathrm{BottomType}$. $\mathrm{BottomType}$ is a subtype of every type and it excludes every type (including itself). As mentioned above, $\mathrm{BottomType}$ is not a first-class type: programmers must not write $\mathrm{BottomType}$.

# Chapter 7

# Names and Declarations

Names are used to refer to certain kinds of entities in a Fortress program. A name is either an identifier or an operator. An operator may be an operator token, a special token corresponding to a special operator character, or a matching pair of enclosing operator tokens.

Names are typically introduced by declarations, which bind the name to an entity. In some cases, the declaration is implicit. Every declaration has a scope, in which the declared name can be used to refer to the declared entity.

*Declarations* introduce *named entities*; we say that a declaration *declares* an entity and a name by which that entity can be referred to, and that the declared name *refers to* the declared entity. As discussed later, there is not a one-one correspondence between declarations and named entities: some declarations declare multiple named entities, some declare multiple names, and some named entities are declared by multiple declarations.

Some declarations contain other declarations. For example, a trait declaration may contain method declarations, and a function declaration may contain parameter declarations.

The positions in which a declaration may legally appear in a component are determined by the nonterminal *Decl* in the simplified Fortress grammar in Appendix D.

## 7.1 Kinds of Declarations

Syntax:

```
Decl  ::=  TraitDecl
        |  ObjectDecl
        |  VarDecl
        |  FnDecl
```

There are two kinds of declarations: *top-level declarations* and *local declarations*.

Top-level declarations occur at the top level of a component, not within any other declaration or expression. A top-level declaration is one of the following:[1]

- trait declarations (see Chapter 10);
- object declarations (see Chapter 11), which may be singleton declarations or constructor declarations;

[1]The Fortress component system, defined in Chapter 20, includes declarations of *components* and *APIs*. Because component names are not used in a Fortress program and API names are used only in `import` and `export` statements, we do not discuss them in this chapter.

- top-level variable declarations (see Section 8.1);
- top-level function declarations (see Chapter 9), including top-level operator declarations (see Chapter 16);

Local declarations occur in another declaration or in some expression (or both). A local declaration is one of the following:

- field declarations (see Section 11.2), which occur in object declarations and object expressions, and include field declarations in the parameter list of a constructor declaration;
- method declarations (see Section 10.2), which occur in trait and object declarations and object expressions;
- local variable declarations (see Section 8.2), which occur in expression blocks;
- local function declarations (see Section 9.4), which occur in expression blocks;
- labeled blocks (see Section 13.12), which are expressions;
- static-parameter declarations, which may declare type parameters, `nat` parameters, `int` parameters, `bool` parameters, or `opr` parameters (see Chapter 12), and occur in static-parameter lists of trait and object declarations, top-level function declarations, method declarations, and local function declarations.
- (value) parameter declarations, which occur in parameter lists of constructor declarations, top-level function declarations, method declarations, local function declarations, and function expressions

In addition to these explicit declarations, there is one case in which names are declared implicitly:

- the special name `self` is implicitly declared as a parameter of dotted methods (see Section 10.2 for details); and
- the name *result* is implicitly declared as a variable for the `ensures` clause of a contract (see Section 9.3 for details).

Trait declarations, object declarations, and type-parameter declarations, are collectively called *type declarations*; they declare names that refer to *types* (see Chapter 6). Constructor declarations, top-level function declarations, method declarations, and local function declarations are collectively called *functional declarations*. Singleton declarations, top-level variable declarations, field declarations, local variable declarations (including implicit declarations of *result*) and (value) parameter declarations (including implicit declarations of `self` ) are collectively called *variable declarations*. Static-parameter declarations are also called *static-variable declarations*. Note that static-variable declarations are disjoint from variable declarations.

The groups of declarations defined in the previous paragraph are neither disjoint nor exhaustive. For example, labeled blocks are not included in any of these groups, and an object declaration is both a type declaration and either a function or variable declaration, depending on whether it is singleton.

Most declarations declare a single name given explicitly in the declaration (though, as discussed in Section 7.2, they may declare this name in multiple namespaces).

Method declarations in a trait may be either *abstract* or *concrete*. Abstract declarations do not have bodies; concrete declarations, sometimes called *definitions*, do.

## 7.2 Namespaces

Fortress supports three namespaces, one for types, one for values, and one for labels. (If we consider the Fortress component system, there is another namespace for APIs.) These namespaces are logically disjoint: names in one namespace do not conflict with names in another.

Type declarations, of course, declare names in the type namespace. Function and variable declarations declare names in the value namespace. (This implies that object names are declared in both the type and value namespaces.) Labeled blocks declare names in the label namespace. Although they are not all type declarations, all the static-variable declarations declare names in the type namespace. In addition, `nat` parameters, `int` parameters, `bool` parameters, `opr` parameters are also declared in the value namespace. Section 12.4 describes `opr` parameters in more detail.

A reference to a name is resolved to the entity that the name refers to the namespace appropriate to the context in which the reference occurs. For example, a name refers to a label if and only if it occurs immediately following the reserved word `exit`. It refers to a type if and only if it appears in a type context (described in Chapter 6). Otherwise, it refers to a value.

## 7.3 Reach and Scope of a Declaration

In this section, we define the *reach* and *scope* of a declaration, which determine where a declared name may be used to refer to the entity declared by the declaration. It is an error for a reference to a name to occur at any point in a program at which the name is not in scope in the appropriate namespace (as defined below), except immediately after the '.' of a dotted field access or dotted method invocation when the name is the name of the field or dotted method of the static type of the receiver expression (see Sections 13.3 and 13.4).

We first define a declaration's *reach*. The reach of a labeled block is the block itself. The reach of a functional method declaration is the component containing that declaration. A dotted method declaration not in an object expression or declaration must be in the declaration of some trait $T$, and its reach is the declaration of $T$ and any trait or object declarations or object expressions that extend $T$; that is, if the declaration of trait $T$ contains a method declaration, and trait $S$ extends trait $T$, then the reach of that method declaration includes the declaration of trait $S$. The reach of any other declaration is the smallest block strictly containing that declaration (i.e., not just the declaration itself). For example, the reach of any top-level declaration (including any imported declaration) is the component containing (or importing) that declaration; the reach of a field declaration is the enclosing object declaration or expression; the reach of a parameter declaration is the constructor declaration, functional declaration, or function expression in whose parameter list it occurs; and the reach of a local variable declaration is the smallest block in which that declaration occurs (recall from Chapter 3 that a local variable declaration always starts a new block). The reach of an implicit declaration of `self` for a dotted method declaration is the method declaration, and the reach of an implicit declaration of $result$ for an `ensures` clause is the `ensures` clause. We say that a declaration *reaches* any point within its reach.

It is an error for two (explicit) declarations with overlapping reaches to declare the same name, *even if the name is declared in different namespaces*, unless one of the following (disjoint) conditions holds:

- both declarations are functional declarations with the same reach;
- both declarations are dotted method declarations that occur in different trait or object declarations;
- both declarations are dotted method declarations, one in an object expression and the other inherited by that object expression;
- one declaration is a field or dotted method declaration that is provided by (i.e., occurs in or is inherited by) some trait or object declaration and the other is a top-level declaration or a functional method declaration; or
- one declaration is a field or dotted method declaration that is provided by (i.e., occurs in or is inherited by) some object expression that is strictly contained in the reach of the other declaration, and the other declaration is not a dotted method declaration provided by a trait that is extended by the object expression.

If any of the first three conditions holds, or if one declaration is a method declaration that occurs in an object declaration or expression that inherits the other declaration (which therefore must be a method declaration), then the two declarations are *overloaded*, and subject to the overloading rules (see Chapter 15).

If two declarations with overlapping reaches declare the same name in the same namespace, and the declarations are not overloaded, then at any point that their reaches overlap, one declaration *shadows* the other for that name in that namespace; references at such points resolve to the entity declared by the shadowing declaration. We often elide mentioning the name and namespace when they are clear from context. Shadowing is permitted only in the following cases:

- In a trait or object declaration, any top-level declaration or functional method declaration is shadowed if it declares a name of a field or dotted method provided (declared or inherited) by the trait or object.
- In an object expression, any declaration of `self` (including implicit declarations) and any declaration which declares a name of a field or dotted method provided (declared or inherited) by the object expression in a block enclosing the expression is shadowed unless the declaration is a method declaration of a trait extended by the object expression.
- In a method declaration, any declaration of `self` (including implicit declarations) in a block enclosing the method declaration is shadowed.
- In the `ensures` clause of a contract, any declaration of *result* in a block enclosing the `ensures` clause is shadowed.
- Any immutable local variable declaration without an initial-value expression is shadowed if the variable is initialized later in the enclosing block.

We say that a name is *in scope* in a namespace at any point in the program within the reach of any declaration that declares that name in that namespace unless one of the following conditions holds:

- the declaration is shadowed at the program point for the name in that namespace;
- the declaration is a field, local variable or parameter declaration, and the program point is in the declaration or lexically precedes the declaration; or
- the declaration is a labeled block, and the program point is in a `spawn` expression in the labeled block.

We say that the *scope* of a declaration for a name in a namespace consists of those points at which the name is in scope for the namespace and the declaration is not shadowed for that name and that namespace. Again, when it is clear from context, we may omit the name and namespace.

# Chapter 8

# Variables

## 8.1 Top-Level Variable Declarations

Syntax:

| | | |
|---|---|---|
| *VarDecl* | ::= | *VarMods*? *VarWTypes InitVal* |
| | \| | *VarMods*? *BindIdOrBindIdTuple* = *Expr* |
| | \| | *VarMods*? *BindIdOrBindIdTuple* : *Type ... InitVal* |
| | \| | *VarMods*? *BindIdOrBindIdTuple* : *TupleType InitVal* |
| *VarMods* | ::= | *VarMod*$^+$ |
| *VarMod* | ::= | *AbsVarMod* |
| *VarWTypes* | ::= | *VarWType* |
| | \| | ( *VarWType*( , *VarWType*)$^+$ ) |
| *VarWType* | ::= | *BindId IsType* |
| *InitVal* | ::= | ( = \| := ) *Expr* |
| *TupleType* | ::= | ( *Type* , *TypeList* ) |
| *TypeList* | ::= | *Type*( , *Type*)* |
| *IsType* | ::= | : *Type* |

A *variable*'s name can be any valid Fortress identifier. There are three forms of variable declarations. The first form:

$id : \mathrm{Type} = expr$

declares $id$ to be an immutable variable with static type $\mathrm{Type}$ whose value is computed to be the value of the *initializer* expression $expr$. The static type of $expr$ must be a subtype of $\mathrm{Type}$.

The second (and most convenient) form:

$id = expr$

declares $id$ to be an immutable variable whose value is computed to be the value of the expression $expr$; the static type of the variable is the static type of $expr$.

The third form:

`var` $id : \mathrm{Type} = expr$

declares $id$ to be a mutable variable of type $\mathrm{Type}$ whose initial value is computed to be the value of the expression $expr$. As before, the static type of $expr$ must be a subtype of $\mathrm{Type}$. The modifier `var` is optional when ‘ := ’ is used instead of ‘ = ’ as follows:

`var`? $\mathit{id}$ : Type := $\mathit{expr}$

In short, immutable variables are declared and initialized by ' = ' and mutable variables are declared and initialized by ' := ' except when they are declared as the third form above with the modifier `var`.

All forms can be used with *tuple notation* to declare multiple variables together. Variables to declare are enclosed in parentheses and separated by commas, as are the types declared for them:

$(\mathit{id}(, \mathit{id})^{+}) : (\mathrm{Type}(, \mathrm{Type})^{+})$

Alternatively, the types can be included alongside the respective variables, optionally eliding types that can be inferred from context:

$(\mathit{id}(\colon \mathrm{Type})?(, \mathit{id}(\colon \mathrm{Type})?)^{+})$

Alternatively, a single type followed by ' . . . ' can be declared for all of the variables:

$(\mathit{id}(, \mathit{id})^{+})\colon \mathrm{Type}\ldots$

This notation is especially helpful when a function application returns a tuple of values.

The initializer expressions of top-level variable declarations can refer to variables declared later in textual order.

Here are some simple examples of variable declarations:

$\pi = 3.141592653589793238462643383279502884197169399375108209749445923078$

declares the variable $\pi$ to be an approximate representation of the mathematical object $\pi$. It is also legal to write:

$\pi\colon \mathbb{R}64 = 3.141592653589793238462643383279502884197169399375108209749445923078$

This definition enforces that $\pi$ has static type $\mathbb{R}64$.

The following example declares multiple variables using tuple notation:

`var` $(x, y)\colon \mathbb{Z}64\ldots = (5, 6)$

The following three declarations are equivalent:

$(x, y, z)\colon (\mathbb{Z}64, \mathbb{Z}64, \mathbb{Z}64) = (0, 1, 2)$
$(x\colon \mathbb{Z}64, y\colon \mathbb{Z}64, z\colon \mathbb{Z}64) = (0, 1, 2)$
$(x, y, z)\colon \mathbb{Z}64\ldots = (0, 1, 2)$

## 8.2 Local Variable Declarations

Syntax:

| | | |
|---|---|---|
| *LocalVarDecl* | ::= | `var` ? *LocalVarWTypes InitVal* |
| | \| | `var` ? *LocalVarWTypes* |
| | \| | `var` ? *LocalVarWoTypes* = *Expr* |
| | \| | `var` ? *LocalVarWoTypes* : *Type ... InitVal*? |
| | \| | `var` ? *LocalVarWoTypes* : *TupleType InitVal*? |
| *LocalVarWTypes* | ::= | *LocalVarWType* |
| | \| | ( *LocalVarWType*( , *LocalVarWType*)$^{+}$ ) |
| *LocalVarWType* | ::= | *BindId IsType* |
| *LocalVarWoTypes* | ::= | *LocalVarWoType* |
| | \| | ( *LocalVarWoType*( , *LocalVarWoType*)$^{+}$ ) |
| *LocalVarWoType* | ::= | *BindId* |

Variables can be declared within expression blocks (described in Section 13.11) via the same syntax as is used for top-level variable declarations (described in Section 8.1) except that local variables must not include modifiers besides `var` and additional syntax is allowed as follows:

- The form:

  `var`? $id$ : Type

  declares a variable without giving it an initial value (where mutability is determined by the presence of the `var` modifier). It is an error if such a variable is referred to before it has been given a value; an immutable variable is initialized by another variable declaration and a mutable variable is initialized by assignment. It is also an error if an immutable variable is initialized more than once. Whenever a variable bound in this manner is assigned a value, the type of that value must be a subtype of its declared type. This form allows declaration of the types of variables to be separated from definitions, and it allows programmers to delay assigning to a variable before a sensible value is known. In the following example, the declaration of the type of a variable and its definition are separated:

  $\pi\colon \mathbb{R}64$
  $\pi = 3.14159265358979323846264338327950288419716939937510820974944592307 8$

# Chapter 9

# Functions

A *function* is a value that has a function type (described in Section 6.7). Each function takes exactly one argument, which may be a tuple, and returns exactly one result, which may be a tuple. A function may be declared as top level or local as described in Section 7.1. Fortress allows functions to be *overloaded* (as described in Chapter 15); there may be multiple function declarations with the same function name in a single lexical scope. Functions can be passed as arguments and returned as values. Single variables may be bound to functions including overloaded functions.

## 9.1 Function Declarations

Syntax:

| | | |
|---|---|---|
| *FnDecl* | ::= | *FnMods*? *FnHeaderFront FnHeaderClause* ( = *Expr*)? |
| *FnMods* | ::= | *FnMod*$^+$ |
| *FnMod* | ::= | *AbsFnMod* |
| *AbsFnMod* | ::= | `test` |
| *FnHeaderFront* | ::= | *NamedFnHeaderFront* |
| | \| | *OpHeaderFront* |
| *NamedFnHeaderFront* | ::= | *Id StaticParams*? *ValParam* |
| *ValParam* | ::= | *BindId* |
| | \| | (*Params*?) |
| *Params* | ::= | (*Param* , )* *Varargs* |
| | \| | *Param*( , *Param*)* |
| *VarargsParam* | ::= | *BindId* : *Type* ... |
| *Varargs* | ::= | *VarargsParam* |
| *Keyword* | ::= | *Param* = *Expr* |
| *PlainParam* | ::= | *BindId IsType*? |
| | \| | *Type* |
| *Param* | ::= | *PlainParam* |
| *FnHeaderClause* | ::= | *IsType*? *FnClauses* |
| *FnClauses* | ::= | *Throws*? *Contract* |
| *Throws* | ::= | `throws` *MayTraitTypes* |

Syntactically, a function declaration consists of the name of the function, optional static parameters (described in Chapter 12), the value parameter with its (optionally) declared type, an optional type of a return value, an optional declaration of thrown checked exceptions (discussed in Chapter 14), a contract for the function (discussed in Section 9.3), and finally an optional body expression preceded by the token = . A `throws` clause does not include naked type variables. Every element in a `throws` clause is a subtype of $\mathrm{CheckedException}$. When a function declaration includes a body expression, it is called a *function definition*. Function declarations can be mutually recursive.

A function takes exactly one argument, which may be a tuple. When a function takes a tuple argument, we abuse terminology by saying that the function takes multiple arguments. Value parameters cannot be mutated inside the function body.

A function's value parameter consists of a parenthesized, comma-separated list of bindings where each binding is one of:

- A plain binding "$\mathit{identifier}$" or "$\mathit{identifier} : T$"
- A varargs binding "$\mathit{identifier} : T \ldots$"

When the parameter is a single plain binding without a declared type, enclosing parentheses may be elided. The following restrictions apply: No two bindings may have the same identifier. Note that it is permitted to have a single plain binding, or to have no bindings. The latter case, "()", is considered equivalent to a single plain binding of the ignored identifier "_" of type (), that is, "$(\_ : ())$". Also, there can be at most one varargs binding.

A parameter declared by varargs binding is called a *varargs parameter*; it is used to pass a variable number of arguments to a function as a single heap sequence. If a function does not have a varargs parameter then the number of arguments is fixed by the function's type.

For example, here is a simple polymorphic function for creating lists:

```
List⟦T extends Object, nat length⟧(rest: T[length]) =
  if length = 0 then Empty
  else Cons(rest_0, List(rest[1 : (length − 1)]))
  end
```

## 9.2 Function Applications

Fortress allows functions to be overloaded; there may be multiple function declarations with the same function name in a single lexical scope. Thus, we need to determine which functional declarations are applicable to a function application.

An overloaded function has multiple arrow types in its function type, and it associates a declaration with each constituent arrow type. When an overloaded function of type $T$ is called with an argument of type $A$, the call is "dispatched" to the declaration associated with the most specific type of $T$ applicable to $A$ (if no such type exists, then the function is not applicable to $A$).

When a function is called (See Section 13.6 for a discussion of function call expressions), function arguments are evaluated in parallel, and the body of the function is evaluated in a new environment, extending the environment in which it is defined with all parameters bound to their arguments.

If the application of a function $f$ ends by calling another function $g$, tail-call optimization may be applied. Storage used by the new environments constructed for the application of $f$ may be reclaimed.

Here are some examples:

$$\sin(\pi)$$
$$\arctan(y, x)$$

If the function's argument is not a tuple, then the argument need not be parenthesized:

$$\sin x$$
$$\log \log n$$

## 9.3 Function Contracts

Syntax:

| | | |
|---|---|---|
| *Contract* | ::= | *Requires*? *Ensures*? *Invariant*? |
| *Requires* | ::= | `requires` { *ExprList*? } |
| *Ensures* | ::= | `ensures` { *EnsuresClauseList*? } |
| *EnsuresClauseList* | ::= | *EnsuresClause*( , *EnsuresClause*)* |
| *EnsuresClause* | ::= | *Expr* (`provided` *Expr*)? |
| *Invariant* | ::= | `invariant` { *ExprList*? } |

Function contracts consist of three optional clauses: a `requires` clause, an `ensures` clause, and an `invariant` clause. All three clauses are evaluated in the scope of the function body.

The `requires` clause consists of a sequence of expressions of type $\mathrm{Boolean}$ separated by commas and enclosed in curly braces. The `requires` clause is evaluated during a function call before the body of the function. If any expression in a `requires` clause does not evaluate to $\mathit{true}$, a $\mathrm{CallerViolation}$ exception is thrown.

The `ensures` clause consists of a sequence of `ensures` subclauses. Each such subclause consists of an expression of type $\mathrm{Boolean}$, optionally followed by a `provided` subclause. A `provided` subclause begins with `provided` followed by an expression of type $\mathrm{Boolean}$. For each subclause in the `ensures` clause of a contract, the `provided` subclause is evaluated immediately after the `requires` clause during a function call (before the function body is evaluated). If a `provided` subclause evaluates to $\mathit{true}$, then the expression preceding this `provided` subclause is evaluated after the function body is evaluated. If the expression evaluated after function evaluation does not evaluate to $\mathit{true}$, a $\mathrm{CalleeViolation}$ exception is thrown. The expression preceding the `provided` subclause can refer to the return value of the function. A $\mathit{result}$ variable is implicitly bound to a return value of the function and is in scope of the expression preceding the `provided` subclause. The implicitly declared $\mathit{result}$ shadows any other declaration with the same name in scope.

The `invariant` clause consists of a sequence of expressions of *any type* enclosed by curly braces. These expressions are evaluated before and after a function call. For each expression $e$ in this sequence, if the value of $e$ when evaluated before the function call is not equal to the value of $e$ after the function call, a $\mathrm{CalleeViolation}$ exception is thrown.

Here are some examples:

```
factorial(n: ℤ64) requires { n ≥ 0 } =
  if n = 0 then 1
  else n factorial(n − 1)
  end
mangle(input: List) ensures { sorted(result) provided sorted(input) } =
  if input ≠ Empty
  then mangle(first(input))
       mangle(rest(input))
  end
```

## 9.4 Local Function Declarations

Syntax:

| | | |
|---|---|---|
| *LocalFnDecl* | ::= | *NamedFnHeaderFront FnHeaderClause* = *Expr* |

Functions can be declared within expression blocks (described in Section 13.11) via the same syntax as is used for top-level function declarations (described in Chapter 9) except that locally declared functions must not be operators.

As with top-level function declarations, locally declared functions in a single scope are allowed to be overloaded and mutually recursive.

# Chapter 10

# Traits

*Traits* are declared by trait declarations. Traits define new named types. A trait specifies a collection of *methods* (described in Section 10.2). One trait can extend others, which means that it inherits the methods from those traits, and that the type defined by that trait is a subtype of the types of traits it extends.

## 10.1 Trait Declarations

Syntax:

```
TraitDecl          ::= TraitMods? TraitHeaderFront TraitClauses GoInATrait? end
TraitHeaderFront   ::= trait Id StaticParams? ExtendsWhere?
TraitClauses       ::= TraitClause*
TraitClause        ::= Excludes
                    |  Comprises
GoInATrait         ::= GoFrontInATrait GoBackInATrait?
                    |  GoBackInATrait
GoFrontInATrait    ::= GoesFrontInATrait+
GoesFrontInATrait  ::= AbsFldDecl
GoBackInATrait     ::= GoesBackInATrait+
GoesBackInATrait   ::= MdDecl
TraitMods          ::= TraitMod+
TraitMod           ::= AbsTraitMod
AbsTraitMod        ::= value
ExtendsWhere       ::= extends TraitTypeWheres
TraitTypeWheres    ::= TraitTypeWhere
                    |  { TraitTypeWhereList }
TraitTypeWhereList ::= TraitTypeWhere( , TraitTypeWhere)*
TraitTypeWhere     ::= TraitType
Excludes           ::= excludes TraitTypes
Comprises          ::= comprises TraitTypes
TraitTypes         ::= TraitType
                    |  { TraitTypeList }
TraitTypeList      ::= TraitType( , TraitType)*
```

| | | |
|---|---|---|
| *TraitType* | ::= | *TypeRef* |
| | \| | *Type* [ *ArraySize*? ] |
| | \| | *Type* ^ *IntExpr* |
| | \| | *Type* ^ ( *ExtentRange* ( × *ExtentRange*)* ) |
| *ArraySize* | ::= | *ExtentRange*( , *ExtentRange*)* |
| *ExtentRange* | ::= | *StaticArg*? # *StaticArg*? |
| | \| | *StaticArg*? : *StaticArg*? |
| | \| | *StaticArg* |
| *StaticArgs* | ::= | ⟦*StaticArgList* ⟧ |
| *StaticArgList* | ::= | *StaticArg*( , *StaticArg*)* |
| *StaticArg* | ::= | *Op* |
| | \| | (*StaticArg*) |
| | \| | *IntExpr* |
| | \| | *BoolExpr* |
| | \| | *Type* |
| | \| | *Expr* |

Syntactically, a trait declaration starts with an optional sequence of modifiers followed by `trait`, followed by the name of the trait, an optional sequence of static parameters (described in Chapter 12), an optional set of *extended* traits, an optional set of *excluded* traits, and zero or more declarations of methods separated by newlines or semicolons, and finally `end`.

Each of `extends` and `excludes` clauses consists of `extends` and `excludes` respectively followed by a set of trait references separated by commas and enclosed in braces '{' and '}'. If such a clause contains only one trait, the enclosing braces may be elided.

A trait with an `extends` clause *explicitly extends* those traits listed in its `extends` clause. We define the extension relation to be the transitive closure of explicit extension. That is, trait $T$ *extends* trait $U$ if and only if $T$ explicitly extends $U$ or if there is some trait $S$ that $T$ explicitly extends and that extends $U$. The extension relation induced by a program is the smallest relation satisfying these conditions. This relation must form an acyclic hierarchy. If a trait $T$ extends trait $U$, or if $T$ and $U$ are the same trait, we say that $T$ is a subtrait of $U$ and that $U$ is a supertrait of $T$.

We say that trait $T$ *strictly extends* trait $U$ if and only if (*i*) $T$ extends $U$ and (*ii*) $T$ is not $U$. We say that trait $T$ *immediately extends* trait $U$ if and only if (*i*) $T$ strictly extends $U$ and (*ii*) there is no trait $V$ such that $T$ strictly extends $V$ and $V$ strictly extends $U$. We call $U$ an *immediate supertrait* of $T$ and $T$ an *immediate subtrait* of $U$.

A trait with an `excludes` clause excludes every trait listed in its `excludes` clause. If a trait $T$ excludes a trait $U$, the two traits are mutually exclusive: neither can extend the other, and no trait can extend them both. As discussed in Section 6.2, the exclusion relation is symmetric. Thus, if $T$ excludes $U$ then $U$ excludes $T$ whether or not $T$ is listed in an `excludes` clause in the declaration of $U$.

For example, the following trait declaration:

```
trait Catalyst extends Object
   catalyze(reaction: Reaction): ()
end
```

declares a trait Catalyst with no modifiers, no static parameters, and no `excludes` clauses. Trait Catalyst extends a trait named Object. A single method (named *catalyze*) is declared, which has a parameter of type Reaction and the return type ().

## 10.2 Method Declarations

Syntax:

| | | |
|---|---|---|
| *MdDecl* | ::= | *MdDef* |
| | \| | `abstract` ? *MdMods*? `getter` ? *MdHeaderFront FnHeaderClause* |
| *MdDef* | ::= | *MdMods*? `getter` ? *MdHeaderFront FnHeaderClause* = *Expr* |
| *AbsMdDecl* | ::= | `abstract` ? *AbsMdMods*? `getter` ? *MdHeaderFront FnHeaderClause* |
| *MdMods* | ::= | *MdMod*$^+$ |
| *MdMod* | ::= | *FnMod* |
| *FnMod* | ::= | *AbsFnMod* |
| *AbsFnMod* | ::= | `test` |
| *AbsMdMods* | ::= | *AbsMdMod*$^+$ |
| *AbsMdMod* | ::= | *AbsFnMod* |
| *MdHeaderFront* | ::= | *NamedMdHeaderFront* |
| | \| | *OpMdHeaderFront* |
| *NamedMdHeaderFront* | ::= | *Id StaticParams*? *MdValParam* |
| *MdValParam* | ::= | ( *MdParams*? ) |
| *MdParams* | ::= | (*MdParam* , )$^*$ *Varargs* |
| | \| | *MdParam*( , *MdParam*)$^*$ |
| *MdParam* | ::= | *Param* |
| | \| | `self` |
| *Param* | ::= | *PlainParam* |
| *PlainParam* | ::= | *BindId IsType*? |
| | \| | *Type* |
| *FnHeaderClause* | ::= | *IsType*? *FnClauses* |
| *IsType* | ::= | : *Type* |
| *FnClauses* | ::= | *Throws*? *Where*? *Contract* |
| *Throws* | ::= | `throws` *MayTraitTypes* |
| *MayTraitTypes* | ::= | {} |
| | \| | *TraitTypes* |

A trait declaration contains a set of method declarations. Syntactically, a method declaration begins with an optional sequence of modifiers followed by the method's name, optional static parameters (described in Chapter 12), the value parameter with its (optionally) declared type, an optional type of a return value, an optional declaration of thrown checked exceptions (discussed in Chapter 14), a contract for the method (discussed in Section 10.3), and finally an optional body expression preceded by the token $=$. If a method declaration has the `getter` modifier, it does not affect the behavior of a program; rather, it documents that the method is intended to serve as a getter. A `throws` clause does not include naked type variables. Every element in a `throws` clause is a subtype of $\mathrm{CheckedException}$.

We say that a method declaration *occurs* in a trait declaration. A trait declaration *declares* a method declaration that occurs in that trait declaration. A trait declaration *inherits* method declarations from the declarations of its immediate supertraits. A trait declaration *provides* the method declarations that it declares or inherits.

There are two sorts of method declarations: *dotted method* declarations and *functional method* declarations. Syntactically, a dotted method declaration is identical to a function declaration. When a method is invoked, a special self parameter is bound to the object on which it is invoked.

A functional method declaration has a parameter named `self` at an arbitrary position in its parameter list. This parameter is not given a type and implicitly has the type of the enclosing trait or object. Semantically, functional method declarations can be viewed as top-level function declarations. For example, the following overloaded functional method $f$ declared within a trait declaration $A$:

```
trait A
    f(self, t : T) = e_1
    f(s : S, self) = e_2
end
```

$f(a, t)$

may be rewritten as top-level functions as follows:

```
trait A
    internalF(t : T) = e_1
    internalF(s : S) = e_2
end
f_1(a : A, t : T) = a.internalF(t)
f_2(s : S, a : A) = a.internalF(s)

f_1(a, t)
```

where *internalF* is a freshly generated name. Functional method declarations may be overloaded with top-level function declarations.

When a method declaration includes a body expression, it is called a *method definition*. A method declaration that does not have its body expression is referred to as an *abstract method declaration*. An abstract method declaration declares an *abstract method*; any object inheriting an abstract method must define a body expression for the method. An abstract method declaration may include the modifier `abstract`. It may elide parameter names but parameter types cannot be omitted except for the self parameter.

Here is an example trait Enzyme which provides methods *reactionSpeed* and *catalyze*:

```
trait Enzyme extends { OrganicMolecule, Catalyst }
  reactionSpeed(): Speed
  catalyze(reaction) = reaction.accelerate(reactionSpeed())
end
```

Enzyme declares the abstract method *reactionSpeed* and declares the concrete method *catalyze* which is inherited as an abstract method from its supertrait Catalyst.

## 10.3 Method Contracts

Syntax:

| | | |
|---|---|---|
| *Contract* | ::= | *Requires*? *Ensures*? *Invariant*? |
| *Requires* | ::= | `requires` { *ExprList*? } |
| *Ensures* | ::= | `ensures` { *EnsuresClauseList*? } |
| *EnsuresClauseList* | ::= | *EnsuresClause*( , *EnsuresClause*)* |
| *EnsuresClause* | ::= | *Expr* (`provided` *Expr*)? |
| *Invariant* | ::= | `invariant` { *ExprList*? } |

Method contracts consist of three optional clauses: a `requires` clause, an `ensures` clause, and an `invariant` clause. All three clauses are evaluated in the scope of the method body. See Section 9.3 for a discussion of each clause.

## 10.4 Value Traits

Syntax:

| | | |
|---|---|---|
| *TraitMod* | ::= | `value` |

If a trait declaration has the modifier `value`, all subtraits of that trait must also have the modifier `value`, and all objects extending that trait are required to be value objects (described in Section 11.3). See Section 11.3 for a discussion of updating fields of value objects.

# Chapter 11

# Objects

*Objects* are values that have object trait types described in Section 6.3.1. Objects contain *methods* (described in Section 10.2) and *fields* (described in Section 11.2).

## 11.1 Object Declarations

Syntax:

| | | |
|---|---|---|
| *ObjectDecl* | ::= | *ObjectMods*? *ObjectHeader* *GoInAnObject*? `end` |
| *ObjectHeader* | ::= | `object` *Id* *StaticParams*? *ObjectValParam*? *ExtendsWhere*? *FnClauses* |
| *ObjectMods* | ::= | *TraitMods* |
| *ObjectValParam* | ::= | `(` *ObjectParams*? `)` |
| *ObjectParams* | ::= | (*ObjectParam* `,`)$^*$ *ObjectVarargs* |
| | \| | *ObjectParam* (`,` *ObjectParam*)$^*$ |
| *ObjectVarargs* | ::= | *Varargs* |
| *ObjectParam* | ::= | *ParamFldMods*? *Param* |
| | \| | *Param* |
| *ParamFldMods* | ::= | *ParamFldMod*$^+$ |
| *ParamFldMod* | ::= | `var` |
| *GoInAnObject* | ::= | *GoFrontInAnObject* *GoBackInAnObject*? |
| | \| | *GoBackInAnObject* |
| *GoFrontInAnObject* | ::= | *GoesFrontInAnObject*$^+$ |
| *GoesFrontInAnObject* | ::= | *FldDecl* |
| *GoBackInAnObject* | ::= | *GoesBackInAnObject*$^+$ |
| *GoesBackInAnObject* | ::= | *MdDef* |
| *FnClauses* | ::= | *Throws*? *Contract* |
| *Throws* | ::= | `throws` *MayTraitTypes* |

Object declarations declare both object values and object trait types. Object declarations extend a set of traits from which they inherit methods. An object declaration inherits the concrete methods of its supertraits and must include a definition for every method declared but not defined by its supertraits. Especially, an object declaration must not include abstract methods (discussed in Section 10.2); it must define all abstract methods inherited from its supertraits. It is also allowed to define overloaded declarations of concrete methods inherited from its supertraits.

Syntactically, an object declaration begins with `object`, followed by the identifier of the object, optional static parameters (described in Chapter 12), optional value parameters, optional traits the object extends, zero or more declarations of fields, and zero or more declarations of methods, separated by newlines or semicolons, and finally `end`. Method declarations in object declarations are syntactically identical to method declarations in trait declarations.

There are two kinds of object declarations: singleton declarations and constructor declarations. A singleton declaration declares a sole, stand-alone, singleton object. It may have static parameters but it does not have a list of value parameters; every instantiation of such an object with the same static arguments yields the same singleton object. A constructor declaration declares an object constructor function. It may have static parameters and it includes a list of value parameters; every call to the constructor of such an object with the same argument yields a new object. Initialization of singleton objects is nondeterministic as described in Chapter 5. The type of an object constructor function is an arrow type consisting of the object's value parameter type followed by the token $\rightarrow$, followed by the object trait type.

For example, the following leaf object extending trait Tree:

```
object Leaf extends Tree
  printTree() : () = println "leaf"
  size() : ℤ32 = 0
end
```

has no fields and two methods.

Here is an example of a Cons object constructor extending trait $\mathrm{List}[\![T]\!]$:

```
object Cons⟦T⟧(first : T, rest : List⟦T⟧)
  extends List⟦T⟧
  cons(x) = Cons(x, self)
  append(xs) = Cons(first, rest.append(xs))
end
```

Note that this declaration introduces the "factory" function $\mathrm{Cons}[\![T]\!](\mathit{first} : T, \mathit{rest} : \mathrm{List}[\![T]\!]) : \mathrm{Cons}[\![T]\!]$ which is used in the body of the object declaration to define the *cons* and *append* methods. Multiple factory functions can be defined by overloading an object constructor with functions. For example:

```
Cons⟦T⟧(first : T) = Cons(first, Empty)
```

## 11.2 Field Declarations

Syntax:

| | | |
|---|---|---|
| *FldDecl* | ::= | *FldMods*? *FldWTypes InitVal* |
| | \| | *FldMods*? *BindIdOrBindIdTuple* = *Expr* |
| | \| | *FldMods*? *BindIdOrBindIdTuple* : *Type ... InitVal* |
| | \| | *FldMods*? *BindIdOrBindIdTuple* : *TupleType InitVal* |
| *FldMods* | ::= | *FldMod*$^+$ |
| *FldMod* | ::= | `var` |
| *FldWTypes* | ::= | *FldWType* |
| | \| | ( *FldWType*( , *FldWType*)$^+$ ) |
| *FldWType* | ::= | *BindId IsType* |

Fields are variables local to an object. They must not be referred to outside their enclosing object declarations. Each field is either a value parameter of a constructor declaration, or it is explicitly defined by a field declaration within the body of a constructor declaration, a singleton declaration, or an object expression. A field declaration in an object declaration is syntactically identical to a top-level variable declaration (described in Section 8.1), with the same meanings attached to the form of variable declarations.

Each field declaration includes an *initial-value* expression, which specifies the initial value of that field. Field initialization occurs in textual program order: evaluation of each initial-value expression must complete before evaluation of the next initial-value expression, and all previous field names (and the parameters of the constructor, in a constructor declaration) are in scope when evaluating an initial-value expression (see Section 7.3). All fields of an object are initialized before that object is made available for subsequent computation; thus, it is illegal to invoke methods on an object being initialized: `self` is *not* implicitly declared by a field declaration.

Within an object declaration or object expression, a field can be accessed by a "naked" identifier reference (see Section 13.2).

## 11.3 Value Objects

An object declaration with the modifier `value` declares a value object that is called in many languages a *primitive* value. The object trait type declared by a value object implicitly has the modifier `value`.

# Chapter 12

# Static Parameters

Trait, object, and functional declarations may be parameterized with *static parameters*. Static parameters are *static variables* listed in white square brackets $[\![$ and $]\!]$ immediately after the name of a trait, object, or functional and they are in scope of the entire body of the declaration. In this chapter, we describe the forms that these static parameters can take.

Syntax:

| | | |
|---|---|---|
| *StaticParams* | ::= | $[\![$*StaticParamList*$]\!]$ |
| *StaticParamList* | ::= | *StaticParam*( , *StaticParam*)$^*$ |

## 12.1 Type Parameters

Syntax:

| | | |
|---|---|---|
| *StaticParam* | ::= | *Id Extends*? ( `absorbs unit` )? |

Static parameters may include one or more type parameters. Syntactically, a type parameter consists of an identifier followed by an optional `extends` clause.

Type parameters are instantiated with types such as trait types, tuple types, and arrow types (See Chapter 6 for a discussion of Fortress types). We use the term *naked type variable* to refer to an occurrence of a type variable as a stand-alone type (rather than as a parameter to another type). Type parameters can appear in any context that an ordinary type can appear, except that a naked type variable must not appear in the `extends` clause of a trait or object declaration, nor in the `throws` clause of a functional or object declaration.

Here is a parameterized trait List:

```
trait List⟦T⟧
   first(): T
   rest(): List⟦T⟧
   cons(T): List⟦T⟧
   append(List⟦T⟧): List⟦T⟧
end
```

## 12.2 Nat and Int Parameters

Syntax:

*StaticParam* ::= `nat` *Id*
  | `int` *Id*

Static parameters may include one or more `nat` and `int` parameters. Syntactically, a `nat` parameter consists of `nat` followed by an identifier. An `int` parameter consists of `int` followed by an identifier. These parameters are instantiated at runtime with numeric values. A `nat` parameter may be used to instantiate other `nat` parameters. An `int` parameter may be used to instantiate other `int` parameters, or to appear in any context that a variable of type $\mathbb{Z}32$ can appear, except that it cannot be assigned to.

For example, the following function $f$:

$$\mathit{makeVector}[\![T\ \texttt{extends}\ \mathrm{Number}, \texttt{nat}\ s_0]\!]() : \mathrm{Vector}[\![T, s_0]\!] = \mathit{vector}[\![T, s_0]\!]$$

declares a `nat` parameter $s_0$, which appears in both the parameter type and return type of $f$.

## 12.3 Bool Parameters

Syntax:
*StaticParam* ::= `bool` *Id*

Static parameters may include one or more `bool` parameters. Syntactically, a `bool` parameter consists of `bool` followed by an identifier. These parameters are instantiated at runtime with boolean values. They may be used to instantiate other `bool` parameters, or to appear in any context that a variable of type Boolean can appear, except that they cannot be assigned to.

For example, the following `coerce` declared in the trait Boolean:

```
trait Boolean
  coerce ⟦bool b⟧(x: BooleanLiteral⟦b⟧) = f(x)
end
```

declares a `bool` parameter $b$, which appears in the parameter type.

## 12.4 Operator Parameters

Syntax:
*StaticParam* ::= `opr` *Op*

Static parameters may include one or more operator names. Syntactically, an operator parameter begins with `opr` followed by an operator name.

Operator parameters may be freely intermixed with other static parameters. For example, the following trait IdentityOp:

```
trait IdentityOp⟦T extends IdentityOp⟦T, ⊙⟧, opr ⊙⟧
  opr ⊙(self) : T = self
end
```

is parameterized with a type parameter $T$ and an operator parameter $\odot$. Unlike other static parameters, operator parameters may be used in both type context and value context. The operator parameter $\odot$ is declared as the second static parameter of IdentityOp, instantiated as a static argument in $\mathrm{IdentityOp}[\![T, \odot]\!]$ which is the bound of $T$, and declared as an operator method in IdentityOp. If a trait or object has an operator parameter `OP` and uses `OP` in an expression, then the trait or object must declare or inherit at least one operator method for `OP` of matching fixity.

Operator parameters are instantiated with operators. They may be used to instantiate other operator parameters and the names of operator declarations. For example, the following trait $\mathrm{MyIdentity}$:

```
object MyIdentity extends IdentityOp⟦MyIdentity, IDENTITY⟧ end
```

instantiates the operator parameter of its supertrait $\mathrm{IdentityOp}$ with the operator name `IDENTITY`. It inherits the `IDENTITY` operator from $\mathrm{IdentityOp}$. Two restrictions apply to instantiations of operator parameters: 1) If a trait or object $T$ has operator parameters $\mathtt{OP}_1$ and $\mathtt{OP}_2$, and $T$ declares or inherits operator methods of the same fixity for both $\mathtt{OP}_1$ and $\mathtt{OP}_2$, then the actual operators passed in any instantiation of $T$ must be different. Note that the clause "of the same fixity" allows both $\mathtt{OP}_1$ and $\mathtt{OP}_2$ to be given the same actual operator, say "$-$", if $\mathtt{OP}_1$ has only a prefix definition and $\mathtt{OP}_2$ has only an infix definition; this is desirable. 2) If a trait or object $T$ has an operator parameter `OP` and declares or inherits an operator method for `OP`, and also declares or inherits an operator method of the same fixity for any actual operator $@$ (that is, where $@$ is not an operator parameter), then the actual operator passed for the `OP` parameter must not be $@$.

Note that any declarations that may be associated with an actual operator name that is passed for an operator parameter are *irrelevant* to the behavior of the operator parameter within the parameterized trait, object, or functional. When a trait or object extends a trait parameterized by operator names, the subtrait inherits methods whose names are the actual operator names instead of the operator parameter names.

An operator method declaration whose name is one of the operator parameters of its enclosing trait or object may be overloaded with other operator declarations in the same component; the operator parameter may be instantiated with any operator in the same component. Therefore, such an operator method declaration must satisfy the overloading rules (described in Chapter 15) with every operator declaration in the same component. For example, the above `IDENTITY` operator declaration is overloaded with the following `IDENTITY` operator declaration:

```
opr IDENTITY(x: ℤ) = x
opr +(x: ℤ) = x
```

where both `IDENTITY` and $+$ are top-level operator declarations in the same component. Therefore, the declaration of the operator $\odot$ in $\mathrm{IdentityOp}$ must satisfy the overloading rules with `IDENTITY` and with $+$.

# Chapter 13

# Expressions

Fortress is an expression-oriented language. The positions in which an expression may legally appear (*value context*) are determined by the nonterminal *Expr* in the Fortress grammar defined in Appendix D. We say that an expression is a *subexpression* of any expression (or any other program construct) that (syntactically) contains it. When evaluation of one subexpression must complete before another subexpression is evaluated, those subexpressions are ordered by dynamic program order (see Chapter 5). This constrains the memory behavior of program constructs, as described in Chapter 19. Unless otherwise specified, abrupt completion of the evaluation of a subexpression causes the evaluation of the expression as a whole to complete abruptly in the same way. Also, if one expression precedes another by dynamic program order, and the evaluation of the first expression completes abruptly, the second is not evaluated at all.

## 13.1 Literals

Syntax:

| | | |
|---|---|---|
| *LiteralExpr* | ::= | `( )` |
| | \| | *NumericLiteralExpr* |
| | \| | *CharLiteralExpr* |
| | \| | *StringLiteralExpr* |

Fortress provides boolean literals, () literal, character literals, string literals, and numeric literals. Literals are values; they do not require evaluation.

## 13.2 Identifier References

Syntax:

| | | |
|---|---|---|
| *Primary* | ::= | *VarOrFnRef* |
| | \| | `self` |
| *VarOrFnRef* | ::= | *Id StaticArgs*? |

A name that is not an operator appearing in an expression context is called an *identifier reference*. It evaluates to the value of the name in the enclosing scope in the value namespace. The type of an identifier reference is the declared type of the name. See Chapter 7 for a discussion of names. An identifier reference performs a memory read operation. Note in particular that if a name is not in scope, it is a static error (as described in Section 7.3).

An identifier reference which denotes a polymorphic function may include explicit static arguments (described in Chapter 9) but most identifier references do not include them. For example, $\mathit{double}[\![\mathrm{String}]\!]$ is an identifier reference with an explicit static argument where the function $\mathit{double}$ is defined as follows:

$$\mathit{double}[\![T]\!](x : T) : (T, T) = (x, x)$$

The special name `self` is declared as a parameter of a method. When a dotted method is invoked, its receiver is bound to the `self` parameter; the value of `self` is the receiver. When a functional method is invoked, the corresponding argument is bound to the `self` parameter; the value of `self` is the argument passed to it. The type of `self` is the type of the trait or object being declared by the innermost enclosing trait or object declaration or object expression. See Section 10.2 for details about `self` parameters.

## 13.3 Dotted Field Accesses

Syntax:

| | | |
|---|---|---|
| *Primary* | ::= | *Primary* . *Id* |

An expression consisting of a single subexpression (called the *receiver expression*), followed by '`.`', followed by an identifier, not immediately followed by a parenthesis or a white square bracket, is a *field access*.

## 13.4 Dotted Method Invocations

Syntax:

| | | |
|---|---|---|
| *Primary* | ::= | *Primary* . *Id StaticArgs*? *ParenthesisDelimited* |
| *ParenthesisDelimited* | ::= | *Parenthesized* |
| | \| | *ArgExpr* |
| | \| | ( ) |
| *Parenthesized* | ::= | ( *Expr* ) |
| *ArgExpr* | ::= | *TupleExpr* |
| | \| | ( (*Expr* ,)$^*$ *Expr* ... ) |
| *TupleExpr* | ::= | ( (*Expr* ,)$^+$ *Expr* ) |

A *dotted method invocation* consists of a subexpression (called the receiver expression), followed by '`.`', followed by an identifier, an optional list of static arguments (described in Chapter 9) and a subexpression (called the *argument expression*). Unlike in function calls (described in Section 13.6), the argument expression must be parenthesized, even if it is not a tuple. There must be no whitespace on the left-hand side of the '`.`' and the left-hand side of the left parenthesis of the argument expression. The receiver expression evaluates to the receiver of the invocation (bound to the self parameter (discussed in Section 10.2) of the method). A method invocation may include explicit instantiations of static parameters but most method invocations do not include them.

The receiver and arguments of a method invocation are each evaluated in parallel in a separate implicit thread (see Section 5.4). After this thread group completes normally, the body of the method is evaluated with the parameter of the method bound to the value of the argument expression (thus evaluation of the body occurs after evaluation of the receiver and arguments in dynamic program order). The value and the type of a dotted method invocation are the value and the type of the method body.

We say that methods or functions (collectively called as *functionals*) may be *applied to* (also "*invoked on*" or "*called with*") an argument. We use "call", "invocation", and "application" interchangeably.

Here are some examples:

$myString.toUppercase()$
$myString.replace($“`foo`”, “`few`”$)$
$myNum.add(otherNum)(*$ NOT myNum.add otherNum $*)$

## 13.5 Naked Method Invocations

Syntax:

*Primary* ::= *Id Primary*

Method invocations that are not prefixed by receivers are *naked method invocations*. A naked method invocation is either a functional method call (see Section 10.2 for a discussion of functional methods) or a method invocation within a trait or object that provides the method declaration. Syntactically, a naked method invocation is same as a function call except that the method name is used instead of an arbitrary expression denoting the applied method. Like function calls, an argument expression need not be parenthesized unless it is a tuple. After the evaluation of the argument expression completes normally, the body of the method is evaluated with the parameter of the method bound to the value of the argument expression. The value and the type of a naked method invocation are the value and the type of the method body.

## 13.6 Function Calls

Syntax:

*Primary* ::= *Primary Primary*

A *function call* consists of two subexpressions: an expression denoting the applied function and an argument expression. The argument expression and the expression denoting the applied function are evaluated *in parallel* in separate implicit threads (see Section 5.4). As with languages such as Scheme and the Java Programming Language, function calls in Fortress are call-by-value. After the evaluation of the function and its arguments completes normally, the body of the function is evaluated with the parameter of the function bound to the value of the argument expression. The value and the type of a function call are the value and the type of the function body. See Section 9.2 for a detailed description of function calls.

Here are some examples:

$\cos(x)$
$\arctan(y, x)$

If the function's argument is not a tuple, then the argument need not be parenthesized:

$\sin x$
$\log \log n$

## 13.7 Operator Applications

Syntax:

| | | |
|---|---|---|
| *OpExpr* | ::= | *EncloserOp OpExpr*? *EncloserOp*? |
| | \| | *OpExpr EncloserOp OpExpr*? |
| | \| | *Primary* |
| *EncloserOp* | ::= | *Encloser* |
| | \| | *Op* |
| *Primary* | ::= | *LeftEncloser ExprList*? *RightEncloser* |
| | \| | *Primary* ^ *Exponent* |
| | \| | *Primary ExponentOp* |
| *Exponent* | ::= | *Id* |
| | \| | *ParenthesisDelimited* |
| | \| | *LiteralExpr* |
| | \| | `self` |
| *ExponentOp* | ::= | `^T` \| ^*Op* |

To support a rich mathematical notation, Fortress allows most Unicode characters that are specified to be mathematical operators to be used as operators in Fortress expressions, as well as various tokens described in Chapter 16. Most of the operators can be used as prefix, infix, postfix, or nofix operators as described in Section 16.3; the fixity of an operator is determined syntactically, and the same operator may have definitions for multiple fixities.

Syntactically, an operator application consists of an operator and its argument expressions. If the operator is a prefix operator, it is followed by its argument expression. If the operator is an infix operator, its two argument expressions come both sides of the operator. If the operator is a postfix operator, it comes right after its argument expression. Like function calls, argument expressions are evaluated *in parallel* in separate implicit threads. After evaluation of arguments completes normally, the body of the operator definition is evaluated with the parameters of the operator bound to the values of the argument expressions. The value and the type of an operator application are the value and the type of the operator body. See Chapter 16 for a detailed description of operators.

Here are some examples:

$(-b + \surd(b^2 - 4a\,c))/2a$
$n^n\,e^n + \surd(2\pi\,n)$
$x_1\,y_2 - x_2\,y_1$
$1/2g\,t^2$
$n(n+1)/2$
$((j+k)!)/(j!k!)$
$1/3\ 3/5\ 5/7\ 7/9\ 9/11$
*println*(“`The answers are` ” $(p+q)$ “ `and` ” $(p-q)$)

## 13.8 Function Expressions

Syntax:

*Expr* ::= `fn` *ValParam IsType*? *Throws*? ⇒ *Expr*

Function expressions denote function values; they do not require evaluation. Syntactically, they start with `fn` followed by a parameter, optional return type, optional `throws` clause, ⇒, and finally an expression. The type of a function expression is an arrow type consisting of the function's parameter type followed by the token →, followed by the function's return type, and the function's optional `throws` clause. Unlike declared functions (described in Chapter 9), function expressions are not allowed to include static parameters.

Here is a simple example:

`fn` $(x{:}\,\mathbb{R}64) \Rightarrow$ `if` $x < 0$ `then` $-x$ `else` $x$ `end`

## 13.9 Object Expressions

Syntax:

$$\textit{DelimitedExpr} \quad ::= \quad \texttt{object}\ \textit{ExtendsWhere}?\ \textit{GoInAnObject}?\ \texttt{end}$$

Object expressions denote object values. Syntactically, they start with `object`, followed by an optional `extends` clause, a series of field declarations, method declarations, and finally `end`. The type of an object expression is an anonymous object expression type that extends the traits listed in the `extends` clause of the object expression. The object expression type does not include the methods introduced by the object expression (i.e., those methods not provided by any supertraits of the object expression). Each object expression type is associated with a program location; every evaluation of a given object expression has the same object expression type. Two object expressions at different program locations have different object expression types.

Unlike object declarations (described in Chapter 11), object expressions are not allowed to include modifiers, value parameters, or static parameters. Object expressions may include free static variables unlike object declarations, which must not include any free static variables (i.e., each static variable in an object declaration must occur as a static parameter). An object expression is not allowed to declare "new" functional methods; it can provide a functional method only with a name that is already declared by one of its supertraits. Field initializers and methods may refer to any variables that are in scope in the context in which the object expression occurs. As with object declarations, initializers are evaluated in textual program order and may refer to previous fields; the object being constructed might not be referred to in any way.

In an object expression, any declaration of `self` (including implicit declarations) and any declaration which declares a name of a field or dotted method provided (declared or inherited) by the object expression in a block enclosing the expression is shadowed unless the declaration is a method declaration of a trait extended by the object expression. In order for an object to be referred to within a nested object expression, the outer object must be renamed before the object expression because `self` declared in the outer object is shadowed in the object expression:

```
object O
  m() = do outer = self
           object
             getOuterSelf() = outer(* outer "self" *)
             getInnerSelf() = self(* inner "self" *)
           end
         end
end
```

For example, the following object expression:

```
f⟦T⟧(x: T) = object f: T = x end
```

has a static variable $T$ that is not its static parameter.

The following example expression evaluates to a new object extending trait Tree:

```
object extends Tree
  printTree() : () = println "leaf"
  size(): ℤ32 = 0
end
```

## 13.10 Assignments

Syntax:

| | | |
|---|---|---|
| *AssignExpr* | ::= | *AssignLefts AssignOp Expr* |
| *AssignLefts* | ::= | ( *AssignLeft*( , *AssignLeft*)$^*$ ) |
| | \| | *AssignLeft* |
| *AssignLeft* | ::= | *SubscriptExpr* |
| | \| | *FieldSelection* |
| *SubscriptExpr* | ::= | *Primary LeftEncloser ExprList*? *RightEncloser* |
| *FieldSelection* | ::= | *Primary* . *Id* |
| *AssignOp* | ::= | := \| *Op* = |

An assignment expression consists of a left-hand side (*AssignLefts*) indicating one or more variables, subscripted expressions (as described in Section 22.7), or field accesses to be updated, an assignment token, and a right-hand-side expression. Multiple left-hand sides must be grouped using tuple notation (comma-separated and parenthesized). Variables updated in an assignment expression must already be declared.

The assignment token ' := ' indicates ordinary assignment. Ordinary assignment proceeds in two phases. In the first, the evaluation phase, the right-hand-side expression is evaluated in parallel with each of the left-hand-side subexpressions, forming an implicit thread group. Evaluating a left-hand variable does nothing. Evaluating a left-hand field reference evaluates the receiving object. Evaluating a left-hand subscripting operation evaluates the receiving object and the index in parallel in a nested thread group. After the outer implicit thread group completes normally, the assignment phase begins. Each component of the right-hand-side value is assigned to the corresponding component of the left-hand side in parallel, forming an implicit thread group. Assigning a left-hand variable simply changes the binding for the variable's location to contain the new value. Assigning a left-hand subscript simply calls the subscripted assignment operation on the corresponding object with the new value.

Any operator (other than ' : ' or ' = ' or ' $<$ ' or ' $>$ ') followed by ' = ' with no intervening whitespace indicates compound (updating) assignment. This adds an additional phase to assignment between the two phases of ordinary assignment. In the first phase, a left-hand subscripting operation invokes the subscripting operator on the receiving object once receiver and index have been evaluated. A left-hand variable simply returns the current value of the location associated with that variable. In the new second phase, the operator indicated in the assignment is invoked. The left-hand argument is the value of the left-hand expression evaluated in the first phase; the right-hand argument is the value evaluated for the right-hand side of the assignment expression. When the operator evaluation completes normally, the assignment phase is run as in ordinary assignment.

The important point to understand is that compound assignment evaluates its subexpressions exactly once, and that the parts of assignment proceed implicitly in parallel where possible. The value and type of an assignment expression is $()$.

Consider the following assignment:

$$(a_i, c) \mathrel{+}= f(t, u)$$

Here in the first phase we evaluate $a$ and $i$ in parallel, then when this completes we invoke the indexing method of $a$ to evaluate $a_i$. In parallel we look up the value of variable $c$. Finally, we evaluate $f(t, u)$ in parallel with all of these left-hand sides. In the second phase, we combine the results using the $+$ operator, to evaluate $(a_i, c) \mathrel{+}= f(t, u)$. This must return a tuple of two values $(p, q)$. In the final phase, in parallel we call the indexed assignment operator $a_i := p$ and perform the local assignment $c := q$.

Here are some simpler and more commonplace examples of assignment:

$x := f(0)$
$n[i\,j] = n[i\,j] + l[i\,k]\,m[k\,j]$
$(a, b, c) := (b, c, a)$(∗ Permuta a, b, and c ∗)
$x \mathrel{+}= 1$
$(x, y) \mathrel{+}= (\delta_{\mathrm{x}}, \delta_{\mathrm{y}})$
$\mathit{myBag} \cup = \mathit{newItems}$

## 13.11 Do Expressions

Syntax:

| | | |
|---|---|---|
| *Do* | ::= | (*DoFront* `also`)* *DoFront* `end` |
| *DoFront* | ::= | (`at` *Expr*)? `atomic`? `do` *BlockElems*? |
| *BlockElems* | ::= | *BlockElem*$^+$ |
| *BlockElem* | ::= | *LocalVarFnDecl* |
| | \| | *Expr*( , *GeneratorClauseList*)? |
| *LocalVarFnDecl* | ::= | *LocalFnDecl*$^+$ |
| | \| | *LocalVarDecl* |

A `do` expression consists of a series of *expression blocks* separated by `also` and terminated by `end`. Each expression block is preceded by an optional `at` expression (described in Section 21.2), an optional `atomic`, and `do`. When prefixed by `at` or `atomic`, it is as though that expression block were evaluated as the body expression of an `at` or `atomic` expression (described in Section 13.22), respectively. An expression block consists of a (possibly empty) series of *elements*–expressions, generated expressions (described in Section 13.11.2), local variable declarations, or local function declarations–separated by newlines or semicolons.

A single expression block evaluates its elements in order: each element must complete before evaluation of the next can begin, and the expression block as a whole does not complete until the final element completes. If the last element of the expression block is an expression, the value and type of this expression are the value and type of the expression block as a whole. Otherwise, the value and type of the expression block is `()`. Each expression block introduces a new scope. Some compound expressions have clauses that are implicitly expression blocks.

Here are examples of function declarations whose bodies are `do` expressions:

```
$f(x:\mathbb{R}64) =$ do
  $(\sin(x)+1)^2$
end

$foo(x:\mathbb{R}64) =$ do
  $y = x$
  $z = 2x$
  $y + z$
end

$mySum(i:\mathbb{Z}64):\mathbb{Z}64 =$ do
  $acc:\mathbb{Z}64 := 0$
  for $j \leftarrow 0:i$ do
    $acc := acc + j$
  end
  $acc$
end
```

### 13.11.1 Distinguishing a Local Declaration from an Equality Expression

Because a local declaration shares a syntax with an *equality expression*, we provide rules for disambiguation:

- If an expression of the form "$e = e$" occurs as a proper subexpression in any construct that is not an expression block, it is an equality expression.
- If such an expression occurs as an immediate subexpression of an expression block, it is a local declaration. Adding parentheses makes the expression an equality expression.

### 13.11.2 Generated Expressions

If a subexpression of a `do` expression has type $()$, the expression may be followed by a '`,`' and a generator clause list (described in Section 13.14). Writing "$\mathit{expr}, \mathit{gens}$" is equivalent to writing "`for` $\mathit{gens}$ `do` $\mathit{expr}$ `end`". See Section 13.15 for the semantics of the `for` expression. Note in particular that $\mathit{expr}$ can be a reducing assignment of the form "$\mathit{variable}$ `OP`$=$ $\mathit{expr}$".

### 13.11.3 Parallel Do Expressions

A series of blocks may be run in parallel using the `also` construct. Any number of contiguous blocks may be joined together by `also`. Each block is run in a separate implicit thread; these threads together form a group. The expression as a whole completes when the group is complete. A thread can be placed in a particular region by using an `at` expression as described in Section 21.2. When multiple expression blocks are separated by `also`, each expression block must have type $()$; the result and type of the parallel `do` expression is also $()$.

For example:

```
treeSum(t: TreeLeaf) = 0
treeSum(t: TreeNode) = do
  var accum : ℤ32 := 0
  do
    accum += treeSum(t.left)
  also do
    accum += treeSum(t.right)
  also do
    accum += t.datum
  end
  accum
end
```

## 13.12 Label and Exit

Syntax:

| | | |
|---|---|---|
| *DelimitedExpr* | ::= | `label` *Id* *BlockElems* `end` *Id* |
| *FlowExpr* | ::= | `exit` *Id*? (`with` *Expr*)? |

An expression block may be labeled using a `label` expression, which consists of `label`, an identifier (the block's *name*), an expression block (its *body*), `end`, and finally its name again (it is a static error if the identifier after `end` is not the name). The name of a `label` expression is in scope in the label namespace at any point in its body that is not within a `spawn` subexpression of the body. A `label` expression is evaluated by evaluating its body.

An `exit` expression consists of `exit`, an optional identifier (the *target*) and an optional `with` clause, which consists of `with` followed by an expression. If a target is specified, it is an error if the target is not in scope in the label namespace at the `exit` expression. That is, the target must be the name of a statically enclosing `label` expression, and the `exit` expression must not be within a `spawn` expression that is contained in the `label` expression. If no target is specified, the target is implicitly the name of the smallest statically enclosing `label` expression; it is an error if there is no such expression.

An `exit` expression with a `with` clause evaluates its `with` clause expression to yield an *exit value*. The `exit` expression completes abruptly with the exit value (see Section 5.2). An `exit` expression with no `with` clause has an implicit `with` clause whose expression is $()$. The type of an `exit` expression is $\mathrm{BottomType}$.

If the evaluation of the body of a `label` expression completes normally, its value is the value of the body. If the evaluation of the body completes abruptly with an `exit` expression whose target is the name of the `label` expression, then the evaluation of the `label` expression completes normally and its value is the exit value of the `exit` expression. The type of a `label` expression is the union of the type of the last expression of its expression block and the types of the values of any `exit` expressions within the `label` expression whose target is the `label` expression's name.

If one or more `try` expressions are nested between an `exit` expression and the targeted `label` block, the `finally` clauses of these expressions are run in order, from innermost to outermost, as described in Section 13.25. If any `finally` clause completes abruptly by throwing an exception, the `exit` expression fails to exit, the evaluation of the `label` expression completes abruptly, and the exception is propagated.

Here is a simple example:

```
label I_95
  if goingTo(Sun)
  then exit I_95 with x32B
  else x32A
  end
end I_95
```

The expression "`exit` $I_{95}$ `with` $x32B$" completes abruptly and attempts to transfer control to the end of the targeted labeled block "`label` $I_{95}$". The targeted labeled block completes normally with value $x32B$.

## 13.13 While Loops

Syntax:

| | | |
|---|---|---|
| *DelimitedExpr* | ::= | `while` *Expr Do* |

A `while` loop consists of a *condition expression* of type Boolean followed by a simple `do` expression (see Section 13.11). An iteration of a `while` loop evaluates the condition expression; if it completes normally and returns *true* it then evaluates the body expression to completion. When one iteration completes a new one is run until either an iteration completes abruptly (in which case the evaluation of the `while` expression completes abruptly), or the condition expression has the value *false* (in which case the `while` loop completes normally with value ()).

## 13.14 Generators

Syntax:

| | | |
|---|---|---|
| *GeneratorClauseList* | ::= | *GeneratorBinding*( , *GeneratorClause*)* |
| *GeneratorBinding* | ::= | *BindIdOrBindIdTuple* ← *Expr* |
| *GeneratorClause* | ::= | *GeneratorBinding* |
| | \| | *Expr* |
| *BindIdOrBindIdTuple* | ::= | *BindId* |
| | \| | ( *BindId* , *BindIdList* ) |
| *BindIdList* | ::= | *BindId*( , *BindId*)* |
| *BindId* | ::= | *Id* |
| | \| | _ |

Fortress makes extensive use of comma-separated *generator clause lists* to express parallel iteration. Generator clause lists occur in generated expressions (described in Section 13.11.2) and `for` loops (described in Section 13.15), sums and big operators (described in Section 13.17), and comprehensions (described in Section 13.28). We refer to these collectively as *expressions with generator clauses*. Every expression with generator clauses contains a *body expression* which is evaluated for each combination of values bound in the generator clause list (each such combination yields an *iteration* of the body).

A generator clause is either a *generator binding* or an expression of type $\mathrm{Generator}[\![()]\!]$ (this includes the type $\mathrm{Boolean}$). A generator clause list must begin with a generator binding. A generator binding consists of one or more comma-separated identifiers followed by the token $\leftarrow$, followed by a subexpression (called the *generator expression*). A generator expression evaluates to an object whose type is $\mathrm{Generator}$. A generator encapsulates zero or more *generator iterations*. By default, the programmer must assume that generator iterations are run in parallel in separate implicit threads unless the generators are instances of $\mathrm{SequentialGenerator}$; the actual behavior of generators is dictated by library code, as described in Section 21.7. No generator iterations are run until the generator expression completes. For each generator iteration, a generator object produces a value or a tuple of values. These values are bound to the identifiers to the left of the arrow, which are in scope of the subsequent generator clause list and of the body of the construct containing the generator clause list.

An expression of type $\mathrm{Generator}[\![()]\!]$ in a generator clause list is interpreted as a *filter*. A generator iteration is performed only if the filter yields $()$. If the filter yields no value, subsequent expressions in the generator clause list will not be evaluated. Note in particular that $\mathit{true}$ is a $\mathrm{Boolean}$ value yielding $()$ exactly once, while $\mathit{false}$ is a $\mathrm{Boolean}$ value that yields no elements.

The order of nesting of generators need not imply anything about the relative order of nesting of iterations. In most cases, multiple generators can be considered equivalent to multiple nested loops. However, the compiler will make an effort to choose the best possible iteration order it can for a multiple-generator loop, and may even combine generators together; there may be no such guarantee for nested loops. Thus loops with multiple generators are preferable to distinct nested loops in general. Note that the early termination behavior of nested looping is subtly different from a single multi-generator loop, since nested loops give rise to nested thread groups; see Section 21.6.

Each generator iteration of the innermost generator clause corresponds to a *body iteration*, or simply an *iteration* of the generator clause list. Each iteration is run in its own implicit thread. Each expression in the generator clause list can be considered to evaluate in a separate implicit thread. Together these implicit threads form a thread group. Evaluation of an expression with generators completes only when this thread group has completed.

Some common $\mathrm{Generator}$s include:

| | |
|---|---|
| $l : u$ | Any range expression |
| $a$ | Array $a$ generates its elements |
| $a.\mathit{indices}()$ | The index set of array $a$ |
| $\{0, 1, 2, 3\}$ | The elements of an aggregate expression |
| $\mathit{sequential}(g)$ | A sequential version of generator $g$ |

The generator $\mathit{sequential}(g)$ forces the iterations using values from $g$ to be performed in order. Every generator has an associated *natural order* which is the order obtained by $\mathit{sequential}$. For example, a sequential `for` loop starting at 1 and going to $n$ can be written as follows:

```
for i ← sequential(1 : n) do
  ...
end
```

The $\mathit{sequential}$ generator respects generator clause list ordering; it will always nest strictly inside preceding generator clauses and outside succeeding ones.

Given a multidimensional array, the $\mathit{indices}$ generator returns a tuple of values, which can be bound by a tuple of variables to the left of the arrow:

```
(i, j) ← my2DArray.indices()
```

## 13.15 For Loops

Syntax:

| | | |
|---|---|---|
| *DelimitedExpr* | ::= | `for` *GeneratorClauseList* *DoFront* `end` |
| *DoFront* | ::= | (`at` *Expr*)? `atomic`? `do` *BlockElems*? |

A `for` loop consists of `for` followed by a generator clause list (discussed in Section 13.14), followed by a non-parallel `do` expression (the loop *body*; see Section 13.11). Parallelism in `for` loops is specified by the generators used (see Section 13.14); in general the programmer must assume that each loop iteration will occur independently in parallel unless every generator is explicitly *sequential*. For each iteration, the body expression is evaluated in the scope of the values bound by the generators. The value and type of a `for` loop is $()$.

## 13.16 Ranges

Syntax:

| | | |
|---|---|---|
| *Range* | ::= | *Expr*? : *Expr*?( : *Expr*?)? |
| | \| | *Expr* # *Expr* |

A *range expression* is used to create a set of integers, called a $\mathrm{Range}$, useful for indexing an array or controlling a `for` loop. Generators in general are discussed further in Section 13.14.

An *explicit range* is self-contained and completely describes a set of integers. Assume that $a$, $b$, and $c$ are expressions that produce integer values.

- The range $a : b$ is the set of $n = \max(0, b - a + 1)$ integers $\{a, a+1, a+2, \ldots, b-2, b-1, b\}$.
- The range $a \# n$ is the set of $\max(0, n)$ integers $\{a, a+1, a+2, \ldots, a+n-3, a+n-2, a+n-1\}$.

Non-static components of a range expression are computed in separate implicit threads. The range is constructed when all components have completed normally.

An *implicit range* may be used only in certain contexts, such as array subscripts, that can supply implicit information. Suppose an implicit range is used as a subscript for an axis of an array for which the lower bound is $l$ and the upper bound is $u$.

- The implicit range $:$ is treated as $l : u$.
- The implicit range $: b$ is treated as $l : b$.
- The implicit range $a :$ and $a \#$ are treated as $a : u$.
- The implicit range $\# s$ is treated as $l \# s$.

One may test whether an integer is in a range by using the operator $\in$:

`if` $j \in (a : b)$ `then` *println* "`win`" `end`

Note that a range is very different from an interval with integer endpoints. The range $3 : 5$ contains only the values 3, 4, and 5, whereas the interval $[3, 5]$ contains all real numbers $x$ such that $3 \leq x \leq 5$.

## 13.17 Summations and Other Reduction Expressions

Syntax:

| | | |
|---|---|---|
| *FlowExpr* | ::= | *Accumulator StaticArgs*? ( `[` *GeneratorClauseList* `]` )? *Expr* |
| *Accumulator* | ::= | $\sum$ \| $\prod$ \| `BIG` *Op* |

A *reduction expression* begins with a big operator such as $\sum$ or $\prod$ followed by an optional static arguments and an optional generator clause list (described in Section 13.14), followed by a body expression. There is no explicit relationship between `BIG` *Op* and *Op*. Instead, a reduction expression corresponds to a call to the `BIG` *Op* operator, which has the following header:

`opr BIG` $Op[\![T]\!](g : (\text{Reduction}[\![R_0]\!], T \rightarrow R_0) \rightarrow R_0) : R$

Here $R$ is the result type of the operator, and $g$ corresponds to the method $generate[\![R_0]\!]$ of the trait $\text{Generator}[\![T]\!]$: it is a function that takes a reduction (of type $\text{Reduction}[\![R_0]\!]$) and a body (of type $T \rightarrow R_0$) and returns a value of type $R_0$ by running body on each generated element and combining them using the reduction. When a generator clause list is provided, generator clauses produce values and bind the values to the identifiers that are used in the subexpression. Each iteration of the body expression is assumed to be evaluated in a separate implicit thread as described in Section 13.14. The resulting values are combined together using *Op*. The value of a reduction expression is this combined value. When no generator clause list is provided, the body is taken to be an object of type Generator and the elements it generates are combined. Thus, the reduction expression $\sum a$ is equivalent to $\sum_{x \leftarrow a} x$.

A reduction expression without a generator clause list:

$\sum g$

is equivalent to the following:

$\sum[x \leftarrow g]x$

Note that reduction expressions without generator clause lists can be used to conveniently sum any aggregate expression (described in Section 13.27), since every aggregate expression is a generator.

## 13.18 If Expressions

Syntax:

| | | |
|---|---|---|
| *DelimitedExpr* | ::= | `if` *Expr* `then` *BlockElems Elifs*? *Else*? `end` |
| | \| | `(` `if` *Expr* `then` *BlockElems Elifs*? *Else* `end`? `)` |
| *Elifs* | ::= | *Elif*$^+$ |
| *Elif* | ::= | `elif` *Expr* `then` *BlockElems* |
| *Else* | ::= | `else` *BlockElems* |

An `if` expression consists of `if` followed by a condition expression of type Boolean, followed by `then`, an expression block, an optional sequence of `elif` clauses (each consisting of `elif` followed by a condition expression, `then`, and an expression block), an optional `else` clause (consisting of `else` followed by an expression block), and finally `end`. Each clause forms an expression block and has the various properties of expression blocks (described in Section 13.11). An `if` expression first evaluates its condition expression. If the condition expression completes normally and results in *true*, the `then` clause is evaluated. If the condition expression results in *false*, the next clause (either `elif` or `else`), if any, is evaluated. An `elif` clause works just as the original `if`, evaluating its condition expression and continuing with its `then` clause if the condition is *true*. An `else` clause simply evaluates its expression block. The type of an `if` expression is the union of the types of the expression blocks of each clause. If there is no `else` clause in an `if` expression, then every clause must have type `()`. The result of the `if` expression is the result of the expression block which is evaluated; if no expression block is evaluated it is `()`. The reserved word `end` may be elided if the `if` expression is immediately enclosed by parentheses. In such a case, an `else` clause is required.

For example,

```
if $x \in \{0,1,2\}$ then $0$
elif $x \in \{3,4,5\}$ then $3$
else $6$ end
```

## 13.19 Case Expressions

Syntax:

| | | |
|---|---|---|
| *DelimitedExpr* | ::= | `case` *Expr Op*? `of` *CaseClauses CaseElse*? `end` |
| *CaseClauses* | ::= | *CaseClause*$^+$ |
| *CaseClause* | ::= | *Expr* $\Rightarrow$ *BlockElems* |
| *CaseElse* | ::= | `else` $\Rightarrow$ *BlockElems* |

A `case` expression begins with `case` followed by a condition expression, followed by an optional operator, `of`, a sequence of case clauses (each consisting of a *guarding expression* followed by the token ⇒, followed by an expression block), an optional `else` clause (consisting of `else` followed by the token ⇒, followed by an expression block), and finally `end`.

A `case` expression evaluates its condition expression and checks each case clause to determine which case clause matches. To find a matched case clause, the guarding expression of each case clause is evaluated in order and compared to the value of the condition expression. For the first clause, the condition expression and guarding expression are evaluated in separate implicit threads; for subsequent clauses the value of the condition expression is retained and only the guarding expression is evaluated. Once both guard and condition expressions have completed normally, the two values are compared according to an optional operator specified. If the operator is omitted, it defaults to $=$ or $\in$. If the type of the guarding expression is a subtype of type $\mathrm{Generator}$ and the condition expression does not, the default operator is $\in$; otherwise, it is $=$.

If the operator application completes normally and returns $true$, the corresponding expression block is evaluated (see Section 13.11) and its value is returned. If the operator application returns $false$, matching continues with the next clause. If no matched clause is found, a $\mathrm{MatchFailure}$ exception is thrown. The optional `else` clause always matches without requiring a comparison. The value of a `case` expression is the value of the right-hand side of the matched clause. The type of a `case` expression is the union of the types of all right-hand sides of the case clauses.

For example, the following `case` expression specifies the operator $\in$:

```
case planet ∈of
  {“Mercury”, “Venus”, “Earth”, “Mars”} ⇒ “inner”
  {“Jupiter”, “Saturn”, “Uranus”, “Neptune”} ⇒ “outer”
  else⇒ “remote”
end
```

but the following does not:

```
case 2 + 2 of
  4 ⇒ “it really is 4”
  5:7 ⇒ “we were wrong again”
end
```

## 13.20 Extremum Expressions

Syntax:

| | | |
|---|---|---|
| *DelimitedExpr* | ::= | `case most` *Op* `of` *CaseClauses* `end` |
| *CaseClauses* | ::= | *CaseClause*$^+$ |
| *CaseClause* | ::= | *Expr* $\Rightarrow$ *BlockElems* |

An extremum expression uses the same syntax as a `case` expression (described in Section 13.19) except that `most` is used where a `case` expression would have a condition expression, the specified operator is not optional, and an extremum expression does not have an optional `else` clause.

All guarding expressions are evaluated in parallel in separate implicit threads as part of the same group in which the guarding expressions themselves are evaluated. The values of the guarding expressions are compared in parallel according to the operator specified. The specified operator must be a total order operator. Which pairs of guarding expressions are compared is unspecified, except that the pairwise comparisons performed will be sufficient to determine that the chosen clause is indeed the extremum (largest or smallest depending on the specified operator) assuming a total order. Any or all pairwise comparisons may be considered.

The expression block of the clause with the extremum guarding expression (and only that clause) is evaluated. If more than one guarding expressions are tied for the extremum, the first clause in textual order is evaluated to yield the result of the extremum expression. The type of an extremum expression is the union of the types of all right-hand sides of the clauses.

For example, the following code:

```
case most>of
  1 mile ⇒ “miles are larger”
  1 kilometer ⇒ “we were wrong again”
end
```

evaluates to “`miles are larger`”.

## 13.21 Typecase Expressions

Syntax:

| | | |
|---|---|---|
| *DelimitedExpr* | ::= | `typecase` *TypecaseBindings* `of` *TypecaseClauses CaseElse*? `end` |
| *TypecaseBindings* | ::= | *TypecaseVars* ( = *Expr*)? |
| *TypecaseVars* | ::= | *BindId* |
| | \| | ( *BindId*( , *BindId*)$^+$ ) |
| *TypecaseClauses* | ::= | *TypecaseClause*$^+$ |
| *TypecaseClause* | ::= | *TypecaseTypes* $\Rightarrow$ *BlockElems* |
| *TypecaseTypes* | ::= | ( *TypeList* ) |
| | \| | *Type* |
| *CaseElse* | ::= | `else` $\Rightarrow$ *BlockElems* |

A `typecase` expression begins with `typecase` followed by an identifier or a sequence of parenthesized identifiers, optionally followed by the token = and a *binding* expression, followed by `of`, a sequence of typecase clauses (each consisting of a sequence of *guarding types* followed by the token $\Rightarrow$, followed by an expression block), an optional `else` clause (consisting of `else` followed by the token $\Rightarrow$, followed by an expression block), and finally `end`.

A `typecase` expression with a binding expression evaluates the expression first; if it completes normally, the value of the expression is bound to the identifiers and the first matching typecase clause is chosen. If there are multiple identifiers, the binding expression must be evaluated to a tuple value of the same number of elements. A typecase clause matches if the type of the value bound to each identifier is a subtype of the corresponding type in the clause. The expression block of the first matched clause (and only that clause) is evaluated (see Section 13.11) to yield the value of the `typecase` expression. If no matched clause is found, a MatchFailure exception is thrown (MatchFailure is an unchecked exception). Unlike bindings in other contexts, the static type of such an identifier is *not* determined by the static type of the expression it is bound to. If the static type of a subexpression in the bindings for a `typecase` expression has type $T$, when typechecking each typecase clause, the static type of the corresponding identifier is the intersection of $T$ and the corresponding guarding type for that clause. The type of a `typecase` expression is the union of types of all right-hand sides of the typecase clauses.

For example:

```
typecase x = myLoser.myField of
   String ⇒ x "foo"
   Number ⇒ x + 3
   Object ⇒ yogiBerraAutograph
end
```

Note that $x$ has a different type in each clause.

For a `typecase` expression without a binding expression, the identifiers must be immutable variables in scope at that point. In that case, the `typecase` expression is equivalent to one that rebinds the variables to their values, with the new bindings shadowing the old ones: the first matching typecase clause is evaluated, and within that clause, the static type of the variable is the intersection of its original type and the guarding type.

## 13.22 Atomic Expressions

Syntax:

```
FlowExpr      ::=  atomic AtomicBack
                |  tryatomic AtomicBack
AtomicBack    ::=  AssignExpr
                |  OpExpr
                |  DelimitedExpr
```

As Fortress is a parallel language, an executing Fortress program consists of a set of threads (See Section 5.4 for a discussion of parallelism in Fortress). In multithreaded programs, it is often convenient for a thread to evaluate some expressions *atomically*. For this purpose, Fortress provides `atomic` expressions.

An `atomic` expression consists of `atomic` followed by a *body expression*. Evaluating an `atomic` expression is simply evaluating the body expression. All reads and all writes which occur as part of this evaluation will appear to occur simultaneously in a single atomic step with respect to *any* action performed by any thread which is dynamically outside. This is specified in detail in Chapter 19. The value and type of an `atomic` expression are the value and type of its body expression.

A `tryatomic` expression consists of `tryatomic` followed by an expression. It acts exactly like `atomic` except that in certain circumstances (see Section 21.5) it throws TryAtomicFailure and discards the effects of its body.

When the body of an `atomic` expression completes abruptly, the `atomic` expression completes abruptly in the same way. If it completes abruptly by exiting to an enclosing `label` expression, writes within the block are retained and become visible to other threads. If it completes abruptly by throwing an uncaught exception, all writes to objects allocated before the `atomic` expression began evaluation are discarded. Writes to newly allocated objects are retained. Any variable reverts to the value it held before evaluation of the `atomic` expression began. Thus, the only values retained from the abruptly completed `atomic` expression will be reachable from the exception object through a chain of newly allocated objects.

Atomic expressions may be nested arbitrarily; the above semantics imply that an inner `atomic` expression is atomic with respect to evaluations which occur dynamically outside the inner `atomic` expression but dynamically inside an enclosing `atomic`.

Implicit threads may be created dynamically within an `atomic` expression. These implicit threads will complete before the `atomic` expression itself does so. The implicit threads may run in parallel, and will see one another's writes; they may synchronize with one another using nested `atomic` expressions.

Note that `atomic` expressions may be evaluated in parallel with other expressions. An `atomic` expression experiences *conflict* when another thread attempts to read or write a memory location which is accessed by the `atomic` expression. The evaluation of such an expression must be partially serialized with the conflicting memory operation (which might be another `atomic` expression). The exact mechanism by which this occurs will vary; the necessary serialization is provided by the implementation. In general, the evaluation of a conflicting `atomic` expression may be abandoned, forcing the effects of execution to be discarded and execution to be retried. The longer an `atomic` expression evaluates and the more memory it touches the greater the chance of conflict and the larger the bottleneck a conflict may impose.

For example, the following code uses a shared counter atomically:

```
$sum\colon \mathbb{Z}32 := 0$
$accumArray[\![N$ extends $\mathrm{Number}$, nat $x]\!](a\colon \mathrm{Array1}[\![N, 0, x]\!])\colon () =$
  for $i \leftarrow a.indices()$ do
    atomic $sum \mathrel{+}= a_i$
  end
```

The loop body reads $a_i$ and $sum$, then adds them and writes the result back to $sum$; this will appear to occur atomically with respect to all other threads—including both other iterations of the loop body and other simultaneous calls to $accumArray$. Note in particular that the `atomic` expression will appear atomic with respect to reads and writes of $a_i$ and $sum$ that do not occur in `atomic` expressions.

## 13.23 Spawn Expressions

Syntax:

*FlowExpr* ::= `spawn` *Expr*

A spawned thread is created using a `spawn` expression. A `spawn` expression consists of `spawn` followed by an expression. A `spawn` expression spawns a thread which evaluates its subexpression in parallel with any succeeding evaluation. The value of a `spawn` expression is the spawned thread and the type of the expression is $\mathrm{Thread}[\![T]\!]$, where $T$ is the static type of the expression spawned. A `spawn` expression cannot be run within the body of an `atomic` expression. The semantics of spawned threads are discussed in Section 5.4.

## 13.24 Throw Expressions

Syntax:

*FlowExpr* ::= `throw` *Expr*

A `throw` expression consists of `throw` followed by a subexpression. The type of the subexpression must be a subtype of the type $\mathrm{Exception}$ (see Chapter 14). A `throw` expression evaluates its subexpression to an exception value and throws the exception value; the expression completes abruptly and has $\mathrm{BottomType}$.

The type $\mathrm{Exception}$ has exactly two direct mutually exclusive subtypes, $\mathrm{CheckedException}$ and $\mathrm{UncheckedException}$. Every $\mathrm{CheckedException}$ that is thrown must be caught or forbidden by an enclosing `try` expression (see Section 13.25), or it must be declared in the `throws` clause of an enclosing functional declaration (see Section 9.1). Similarly, every $\mathrm{CheckedException}$ declared to be thrown in the static type of a functional called must be either caught or forbidden by an enclosing `try` expression, or declared in the `throws` clause of an enclosing functional declaration.

## 13.25 Try Expressions

Syntax:

| | | |
|---|---|---|
| *DelimitedExpr* | ::= | `try` *BlockElems Catch*? (`forbid` *TraitTypes*)? (`finally` *BlockElems*)? `end` |
| *Catch* | ::= | `catch` *BindId CatchClauses* |
| *CatchClauses* | ::= | *CatchClause*$^+$ |
| *CatchClause* | ::= | *TraitType* $\Rightarrow$ *BlockElems* |

A `try` expression starts with `try` followed by an expression block (the `try` *block*), followed by an optional `catch` clause, an optional `forbid` clause, an optional `finally` clause, and finally `end`. A `catch` clause consists of `catch` followed by an identifier, followed by a sequence of subclauses (each consisting of an exception type followed by the token $\Rightarrow$ followed by an expression block). A `forbid` clause consists of `forbid` followed by a set of exception types. A `finally` clause consists of `finally` followed by an expression block. Note that the `try` block and the clauses form expression blocks and have the various properties of expression blocks (described in Section 13.11).

The expressions in the `try` block are first evaluated in order until they have all completed normally, or until one of them completes abruptly. If the `try` block completes normally, the *provisional* value of the `try` expression is the value of the last expression in the `try` block. In this case, and in case of exiting to an enclosing `label` expression, the `catch` and `forbid` clauses are ignored.

If an expression in the `try` block completes abruptly by throwing an exception, the exception value is bound to the identifier specified in the `catch` clause, and the type of the exception is matched against the subclauses of the `catch` clause in turn, exactly as in a `typecase` expression (Section 13.21). The right-hand-side expression block of the first matching subclause is evaluated. If it completes normally, its value is the provisional value of the `try` expression. If the `catch` clause completes abruptly, the `try` expression completes abruptly. If a thrown exception is not matched by the `catch` clause (or this clause is omitted), but it is a subtype of the exception type listed in a `forbid` clause, a new $\mathrm{ForbiddenException}$ is created with the thrown exception as its argument and thrown.

If an exception thrown from a `try` block is matched by both `catch` and `forbid` clauses, the exception is caught by the `catch` clause. If an exception thrown from a `try` block is not matched by any `catch` or `forbid` clause, the `try` expression completes abruptly.

The expression block of the `finally` clause is evaluated after completion of the `try` block and any `catch` or `forbid` clause. The type of this expression block must be (). The expressions in the `finally` clause are evaluated in order until they have all completed normally, or until one of them completes abruptly. In the latter case, the `try` expression completes abruptly exactly as the subexpression in the `finally` clause does.

If the `finally` clause completes normally, and the `try` block or the `catch` clause completes normally, then the `try` expression completes normally with the provisional value of the `try` expression. Otherwise, the `try` expression completes abruptly as specified above. The type of a `try` expression is the union of the type of the `try` block and the types of all the right-hand sides of the `catch` clauses.

For example, the following `try` expression:

```
try
   inp = read(file)
   write(inp, newFile)
forbid IOFailure
end
```

is equivalent to:

```
try
   inp = read(file)
   write(inp, newFile)
catch e
   IOFailure ⇒ throw ForbiddenException(e)
end
```

The following example ensures that *file* is closed properly even if an IO error occurs:

```
try
   open(file)
   inp = read(file)
   write(inp, newFile)
catch e
   IOFailure ⇒ throw ForbiddenException(e)
finally
   close(file)
end
```

## 13.26 Tuple Expressions

Syntax:

| | | |
|---|---|---|
| *TupleExpr* | ::= | ( (*Expr* , )$^+$ *Expr* ) |

A tuple expression is an ordered sequence of expressions separated by commas and enclosed in parentheses. There must be at least two element expressions. The type of a tuple expression is a tuple type (as discussed in Section 6.5). Each element of a tuple is evaluated in parallel in a separate implicit thread (see Section 5.4).

## 13.27 Aggregate Expressions

Syntax:

| | | |
|---|---|---|
| *Primary* | ::= | *LeftEncloser ExprList*? *RightEncloser* |
| *ExprList* | ::= | *Expr*( , *Expr*)$^*$ |
| *ArrayExpr* | ::= | [ *RectElements* ] |
| *RectElements* | ::= | *Expr MultiDimCons*$^*$ |
| *MultiDimCons* | ::= | *RectSeparator Expr* |
| *RectSeparator* | ::= | ;+ |
| | \| | *Whitespace* |

*Aggregate expressions* evaluate to values that are themselves homogeneous collections of values. Each subexpression of an aggregate expression is evaluated in parallel in a separate implicit thread (see Section 5.4). With the exception of array expressions, the resulting values are passed to the appropriate bracketing operator, which is a function with a varargs parameter constructing the aggregate. The array aggregate operations construct the aggregate directly, after all subexpressions have completed normally. Any aggregate expression may reserve storage for its result before its elements complete evaluation.

Functions defining aggregate expressions are provided in the Fortress standard libraries for sets, maps, lists, matrices, vectors, and arrays.

### 13.27.1 Set Expressions

Set element expressions are enclosed in braces and separated by commas. The type of a set expression is $\mathrm{Set}[\![T]\!]$, where $T$ is the union type of the types of all element expressions of the set expression.

Set containment is checked with the operator $\in$. For example:

$3 \in \{0, 1, 2, 3, 4, 5\}$

evaluates to $\mathit{true}$.

### 13.27.2 Map Expressions

Map entries are enclosed in curly braces, separated by commas, and matching pairs are separated by $\mapsto$. The type of a map expression is $\mathrm{Map}[\![K, V]\!]$ where $K$ is the union of the types of all left-hand-side expressions of the map entries, and $V$ is the union of the types of all right-hand-side expressions of the map entries.

A map $m$ is indexed by placing an element in the domain of $m$ enclosed in brackets immediately after an expression evaluating to $m$. For example, if:

$m = \{$ "a" $\mapsto 0,$ "b" $\mapsto 1,$ "c" $\mapsto 2\}$

then $m[$"b"$]$ evaluates to $1$. In contrast, $m[$"x"$]$ throws a $\mathrm{NotFound}$ exception, as "x" is not an index of $m$.

### 13.27.3 List Expressions

List element expressions are enclosed in angle brackets $\langle$ and $\rangle$ and are separated by commas. The type of a list expression is $\mathrm{List}[\![T]\!]$ where $T$ is the union type of the types of all element expressions.

A list $l$ is indexed by placing an index enclosed in square brackets immediately after an expression evaluating to $l$. Lists are always indexed from $0$. For example:

$\langle 3, 2, 1, 0\rangle[2]$

evaluates to $1$.

### 13.27.4 Array Expressions

Array element expressions are enclosed in brackets. Element expressions along a row are separated only by whitespace. Two dimensional array expressions are written by separating rows with newlines or semicolons. If a semicolon appears, whitespace before and after the semicolon is ignored. The parts of higher-dimensional array expressions are separated by repeated-semicolons, where the dimensionality of the result is equal to one plus the number of repeated semicolons. The type of a $k$-dimensional array expression is $\mathrm{Array}k[\![T, 0, n_0, \cdots, 0, n_{k-1}]\!]$, where $T$ is the union type of the types of the element expressions and $n_0, ..., n_{k-1}$ are the sizes of the array in each dimension[1]. This type can be abbreviated as $T[n_0, \cdots, n_{k-1}]$.

A $k$-dimensional array $A$ is indexed by placing a sequence of $k$ indices enclosed in brackets, and separated by commas, after an expression evaluating to $A$. By default, arrays are indexed from 0. The horizontal dimension of an array is the last dimension mentioned in the array index. For example:

$A\colon \mathbb{Z}32[3, 3] = [1\ 2\ 3; 4\ 5\ 6; 7\ 8\ 9]$

[1] As of March 2008, the Fortress library provides $\mathrm{Array}k$ types for $1 \leq k \leq 3$.

then $A[1,0]$ evaluates to $4$.

An array of two dimensions whose elements are a subtype of $\mathrm{Number}$ is a matrix. Matrices are indexed in the same manner as arrays.

A one-dimensional array whose elements are a subtype of $\mathrm{Number}$ is a vector.

The element expressions in an array expression may be either scalars or array expressions themselves. If an element is an array expression, it is "flattened" (pasted) into the enclosing expression. This pasting works because array expressions never contain other arrays as elements. The elements along a row (or column) must have the same number of columns (or rows), though two elements in different rows (columns) need not have the same number of columns (rows).

The following four examples are all equivalent:

$$\begin{aligned}A{:}\,\mathbb{Z}32[2,2] = [&3\ 4\\ &5\ 6]\end{aligned}$$

$$\begin{aligned}A{:}\,\mathbb{Z}32[2,2] = [&3\ 4;\\ &5\ 6]\end{aligned}$$

$$\begin{aligned}A{:}\,\mathbb{Z}32[2,2] = [&3\ 4\\ ;&5\ 6\,]\end{aligned}$$

$$A{:}\,\mathbb{Z}32[2,2] = [3\ 4; 5\ 6]$$

Here is a $3 \times 3 \times 3 \times 2$ matrix example:

```
[1 0 0
 0 1 0
 0 0 1; ; 0 1 0
          1 0 1
          0 1 0; ; 1 0 1
                   0 1 0
                   1 0 1]
```

## 13.28 Comprehensions

Syntax:

| | | |
|---|---|---|
| *Comprehension* | ::= | `BIG` ? { *StaticArgs*? *Entry* \| *GeneratorClauseList* } |
| | \| | `BIG` ? *LeftEncloser StaticArgs*? *Expr* \| *GeneratorClauseList RightEncloser* |
| *Entry* | ::= | *Expr* $\mapsto$ *Expr* |

Fortress provides *comprehension* syntax, in which a generator clause list binds values used in the body expression on the left-hand side of the token |. As described in Section 13.14, each iteration of the body expression must be assumed to execute in its own implicit thread. Comprehensions evaluate to aggregate values and have corresponding aggregate types. The rules for evaluation of a comprehension are similar to those for a reduction expression (see Section 13.17).

The relationship between a comprehension and an aggregate expression (Section 13.27) is similar to the relationship between a reduction expression (Section 13.17) and the corresponding infix operator application (Section 13.7). The language does not enforce an explicit connection between comprehension syntax and the corresponding aggregate syntax, but in practice libraries that provide definitions for aggregate expressions are expected to define a corresponding comprehension and *vice versa*. As with reduction expressions, a comprehension using a particular set of enclosers corresponds to a call to a *big bracketing operator*. Thus the definition of set comprehensions is given by a function with the following signature:

$\texttt{opr BIG}\{[\![T]\!]g: (\text{Reduction}[\![R_0]\!], T \to R_0) \to R_0\}: \text{Set}[\![T]\!]$

This is almost identical to the signature required to define a reduction expression `BIG` *Op*, shown in Section 13.17. Further information on defining comprehensions are described in Section 21.7.

A set comprehension is enclosed in braces, with a left-hand body separated by the token | from a generator list. For example, the comprehension:

$\{\, x^2 \mid x \leftarrow \{0, 1, 2, 3, 4, 5\}, x \;\texttt{MOD}\; 2 = 0\}$

evaluates to the set

$\{0, 4, 16\}$

Map comprehensions are like set comprehensions, except that the left-hand body must be of the form $e_1 \mapsto e_2$. If $e_1$ produces the same value but $e_2$ a different value on more than one iteration of the generator list, a $\text{KeyOverlap}$ exception is thrown. For example:

$\{\, x^2 \mapsto x^3 \mid x \leftarrow \{0, 1, 2, 3, 4, 5\}, x \;\texttt{MOD}\; 2 = 0\}$

evaluates to the map

$\{0 \mapsto 0, 4 \mapsto 8, 16 \mapsto 64\}$

List comprehensions are like set comprehensions, except that they are syntactically enclosed in angle brackets. For example:

$\langle\, x^2 \mid x \leftarrow \{0, 1, 2, 3, 4, 5\}, x \;\texttt{MOD}\; 2 = 0\rangle$

evaluates to the list

$\langle 0, 4, 16\rangle$

Note that the order of elements in the resulting list corresponds to the *natural order* of the generators in the generator clause list (see Section 21.7).

## 13.29 Type Ascription

Syntax:

*Expr* ::= *Expr* `as` *Type*

An expression consisting of a single subexpression, followed by `as`, followed by a type, is a *type ascription*. The value of the expression is the value of the subexpression. The static type of the expression is the ascribed type. The type of the subexpression must be a subtype of the ascribed type. A type ascription does not affect the dynamic type of the value the expression evaluates to (unlike a type assumption described in Section 13.30).

## 13.30 Type Assumption

Syntax:

*Expr* ::= *Expr* `asif` *Type*

An expression consisting of a single subexpression, followed by `asif`, followed by a type, is a *type assumption*. The value of the expression is the value of the subexpression. The static type of the expression is the given type. The type of the subexpression must be a subtype of the given type. A type assumption considers both the static and the dynamic type of the value of the expression to be the specified type for the purposes of the immediately enclosing function, method, or operator invocation or field access. This is in contrast to type ascription, which only gives a *static* type to an expression. Type assumption is used to access a method provided by a supertrait when multiple supertraits provide different methods with the same name. Fortress thus provides a richer version of type assumption operations such as `super` in the Java Programming Language.

# Chapter 14

# Exceptions

Exceptions are values that can be thrown and caught, via `throw` expressions (described in Section 13.24) and `catch` clauses of `try` expressions (described in Section 13.25). When a `throw` expression "`throw` $e$" is evaluated, the subexpression $e$ is evaluated to an exception. The static type of $e$ must be a subtype of $\mathrm{Exception}$. Then the `throw` expression tries to transfer control to its *dynamically containing block* (described in Chapter 5), from the innermost outward, until either (*i*) an enclosing `try` expression is reached, with a `catch` clause matching a type of the thrown exception, or (*ii*) the outermost dynamically containing block is reached.

If a matching `catch` clause is reached, the right-hand side of the first matching subclause is evaluated. If no matching `catch` clause is found before the outermost dynamically containing block is reached, the outermost dynamically containing block completes abruptly whose cause is the thrown exception.

If an enclosing `try` expression of a `throw` expression includes a `finally` clause, and the `try` expression completes abruptly, the `finally` clause is evaluated before control is transferred to the dynamically containing block.

## 14.1 Causes of Exceptions

Every exception is thrown for one of the following reasons:

1. A `throw` expression is evaluated.
2. An implementation resource is exceeded (e.g., an attempt is made to allocate beyond the set of available locations).

## 14.2 Types of Exceptions

All exceptions are subtypes of the type $\mathrm{Exception}$ declared as follows:

```
trait Exception comprises { CheckedException, UncheckedException }
end
```

Every exception is a subtype of either type $\mathrm{CheckedException}$ or $\mathrm{UncheckedException}$:

```
trait CheckedException extends Exception excludes UncheckedException
end
trait UncheckedException extends Exception excludes CheckedException
```

```
end
```

# Chapter 15

# Overloading and Multiple Dispatch

*In order to synchronize the Fortress language specification with the implementation, it was necessary to drop static overloading checks from the specification. Contrary to the Fortress Language Specification, Version 1.0 $\beta$, there may not exist a unique most specific declaration for a functional call. Thus, it may be ambiguous which declaration shall be applied at run time. This possibility of ambiguous calls at run time will be eliminated by a set of static overloading checks when it is implemented.*

Fortress allows functions and methods (collectively called *functionals*) to be *overloaded*. That is, there may be multiple declarations for the same functional name visible in a single scope (which may include inherited method declarations), and several of them may be applicable to any particular functional call.

However, with these benefits comes the potential for ambiguous calls at run time. For an overloaded functional call, the most specific applicable declaration is chosen, if any. Otherwise, any of the applicable declarations such that no other applicable declaration is more specific than them is chosen.

In this chapter, we describe how to determine which declarations are *applicable* to a particular functional call, and when several are applicable, how to select among them. We also provide overloading rules for the declarations of functionals to eliminate *some possibility* of ambiguous calls at run time, whether or not these calls actually appear in the program. Section 15.1 introduces some terminology and notation. In Section 15.2, we show how to determine which declarations are applicable to a *named functional call* (a function call described in Section 13.6 or a naked method invocation described in Section 13.5) when all declarations have only ordinary parameters (without varargs parameters). We discuss how to handle dotted method calls (described in Section 13.4) in Section 15.3, and declarations with varargs parameters in Section 15.4. Determining which declaration is applied, if several are applicable, is discussed in Section 15.5. Section 15.6 outlines several criteria for valid functional overloading.

## 15.1 Principles of Overloading

Fortress allows multiple functional declarations of the same name to be declared in a single scope. However, recall from Chapter 7 the following shadowing rules:

- dotted method declarations shadow top-level function declarations with the same name, and
- dotted method declarations provided by a trait or object declaration or object expression shadow functional method declarations with the same name that are provided by a different trait or object declaration or object expression.

Also, note that a trait or object declaration or object expression must not have a functional method declaration and a dotted method declaration with the same name, either directly or by inheritance. Therefore, top-level functions can overload with other top-level functions and functional methods, dotted methods with other dotted methods, and functional methods with other functional methods and top-level functions. If a top-level function declaration is overloaded with a functional method declaration, the top-level function declaration must not be more specific than the functional method declaration.

Operator declarations with the same name but different fixity are not a valid overloading; they are unambiguous declarations. An operator method declaration whose name is one of the operator parameters (described in Section 12.4) of its enclosing trait or object may be overloaded with other operator declarations in the same component. Therefore, such an operator method declaration must satisfy the overloading rules (described in Section 15.6) with every operator declaration in the same component.

Recall from Chapter 6 that we write $T \preceq U$ when $T$ is a subtype of $U$, and $T \prec U$ when $T \preceq U$ and $T \neq U$.

## 15.2 Applicability to Named Functional Calls

In this section, we show how to determine which declarations are applicable to a named functional call when all declarations have only ordinary parameters (i.e., without varargs parameters).

For the purpose of defining applicability, a named functional call can be characterized by the name of the functional and its argument type. Recall that a functional has a single parameter, which may be a tuple (a dotted method has a receiver as well). We use *call* $f(C)$ to refer to a named functional call with name $f$ and whose argument, when evaluated, has dynamic type $C$. We assume throughout this chapter that all static variables in functional calls have been instantiated or inferred.

We also use *function declaration* $f(P) : U$ to refer to a function declaration with function name $f$, parameter type $P$, and return type $U$.

For method declarations, we must take into account the self parameter, as follows:

A *dotted method declaration* $P_0.f(P) : U$ is a dotted method declaration with name $f$, where $P_0$ is the trait or object type in which the declaration appears, $P$ is the parameter type, and $U$ is the return type. (Note that despite the suggestive notation, a dotted method declaration does not explicitly list its self parameter.)

A *functional method declaration* $f(P) : U$ *with self parameter at* $i$ is a functional method declaration with name $f$, with the parameter `self` in the $i$th position of the parameter type $P$, and return type $U$. Note that the static type of the self parameter is the trait or object trait type in which the declaration $f(P) : U$ occurs. In the following, we will use $P_i$ to refer to the $i$th element of $P$.

We elide the return type of a declaration, writing $f(P)$ and $P_0.f(P)$, when the return type is not relevant to the discussion. Note that static parameters may appear in the types $P_0$, $P$, and $U$.

A declaration $f(P)$ is *applicable* to a call $f(C)$ if the call is in the scope of the declaration and $C \preceq P$. (See Chapter 7 for the definition of scope.) If the parameter type $P$ includes static parameters, they are inferred as described in Chapter 18 before checking the applicability of the declaration to the call.

Note that a named functional call $f(C)$ may invoke a dotted method declaration if the declaration is provided by the trait or object enclosing the call. To account for this, let $C_0$ be the trait or object declaration immediately enclosing the call. Then we consider a named functional call $f(C)$ as $C_0.f(C)$ if $C_0$ provides dotted method declarations applicable to $f(C)$, and use the rule for applicability to dotted method calls (described in Section 15.3) to determine which declarations are applicable to $C_0.f(C)$.

## 15.3 Applicability to Dotted Method Calls

Dotted method applications can be characterized similarly to named functional applications, except that, analogously to dotted method declarations, we use $C_0$ to denote the dynamic type of the receiver object, and, as for named functional calls, $C$ to denote the dynamic type of the argument of a dotted method call. We write $C_0.f(C)$ to refer to the call.

A dotted method declaration $P_0.f(P)$ is *applicable* to a dotted method call $C_0.f(C)$ if $C_0 \preceq P_0$ and $C \preceq P$. If the types $P_0$ and $P$ include static parameters, they are inferred as described in Chapter 18 before checking the applicability of the declaration to the call.

## 15.4 Applicability for Functionals with Varargs Parameters

A declaration with a varargs parameter corresponds to an infinite number of declarations, one for every number of arguments that may be passed to the varargs parameter. In practice, we can bound that number by the maximum number of arguments that the functional is called with anywhere in the program (in other words, a given program will contain only a finite number of calls with different numbers of arguments). The expansion described here is a conceptual one to simplify the description of the semantics; we do not expect a real implementation to actually expand these declarations at compile time. For example, the following declaration:

$f(x : \mathbb{Z}, y : \mathbb{Z}, z : \mathbb{Z} \ldots) : \mathbb{Z}$

would be expanded into:

$f(x : \mathbb{Z}, y : \mathbb{Z}) : \mathbb{Z}$
$f(x : \mathbb{Z}, y : \mathbb{Z}, z_1 : \mathbb{Z}) : \mathbb{Z}$
$f(x : \mathbb{Z}, y : \mathbb{Z}, z_1 : \mathbb{Z}, z_2 : \mathbb{Z}) : \mathbb{Z}$
$f(x : \mathbb{Z}, y : \mathbb{Z}, z_1 : \mathbb{Z}, z_2 : \mathbb{Z}, z_3 : \mathbb{Z}) : \mathbb{Z}$
$\ldots$

A declaration with a varargs parameter is applicable to a call if any one of the expanded declarations is applicable.

## 15.5 Overloading Resolution

To evaluate a given functional call, it is necessary to determine which functional declaration to dispatch to. To do so, we consider the declarations that are applicable to that call at run time. If there is exactly one such declaration, then the call dispatches to that declaration. If there is no such declaration, then the call is *undefined*, which is an error. If multiple declarations are applicable to the call at run time, then we choose an arbitrary declaration among the declarations such that no other applicable declaration is more specific than them.

We use the subtype relation to compare parameter types to determine a more specific declaration. Formally, a declaration $f(P)$ is *more specific* than a declaration $f(Q)$ if $P \prec Q$. Similarly, a declaration $P_0.f(P)$ is more specific than a declaration $Q_0.f(Q)$ if $P_0 \prec Q_0$ and $P \prec Q$. If the declarations include static parameters, they are inferred as described in Chapter 18 before comparing their parameter types to determine which declaration is more specific.

## 15.6 Overloading Rules for Functional Declarations

This section provides rules for valid functional overloading. If a pair of overloaded declarations satisfies any one of the following three rules, it is considered a valid overloading. The overloading rules given in this section are disjoint; at most one rule applies to each pair of overloaded declarations. It is possible for new rules to be added which allow additional overloadings. Valid overloadings for declarations that contain varargs parameters are determined by analyzing the expansion of these declarations as described in Section 15.4. Therefore, varargs parameters are ignored in the remainder of this section.

### 15.6.1 Type Relationships with Static Parameters

Before describing the overloading rules, we extend two type relations to take static parameters into account. For type parameters, we use their bounds as their supertypes. As with the rest of this chapter, the type relations described in this section may be replaced when the static type checker and the type inference engine are implemented.

We say that type $T$ is a subtype of $U$ if any of the following holds:

- neither type includes static parameters and $T$ is a subtype of $U$ (as described in Section 6.2),
- $U$ does not include any static parameters and any supertype of $T$ that does not include a static parameter is a subtype of $U$ (as described in Section 6.2).

We say that two types exclude each other if any of the following holds:

- neither type can be extended,
- one cannot be extended, and the other is not its supertype, or
- their supertypes exclude each other.

### 15.6.2 Subtype Rule

If the parameter type of one declaration is a subtype of the parameter type of another (and they are not the same) then there is no ambiguity between these two declarations: for every call to which both are applicable, the first is more specific. This is the basis of the Subtype Rule.

The Subtype Rule also requires a relationship between the return types of the two declarations. Without such a requirement, a program may violate type safety.

**The Subtype Rule for Functions and Functional Methods:** Suppose that $f(P) : U$ and $f(Q) : V$ are two distinct function or functional method declarations in a single scope. If $P \prec Q$ and $U \preceq V$ then $f(P)$ and $f(Q)$ are a valid overloading.

**The Subtype Rule for Dotted Methods:** Suppose that $P_0.f(P) : U$ and $Q_0.f(Q) : V$ are two distinct dotted method declarations provided by a trait or object $C$. If $P_0 \prec Q_0$, $P \prec Q$, and $U \preceq V$ then $P_0.f(P)$ and $Q_0.f(Q)$ are a valid overloading.

### 15.6.3 Exclusion Rule

The basic idea behind the Exclusion Rule is that if there is no call to which two overloaded declarations are both applicable then there is no potential for ambiguous calls. In such a case, we say that the declarations are exclusive. We write $T \lozenge U$ if $T$ and $U$ exclude each other. Note that if $T \lozenge U$ then no type is substitutable for both $T$ and $U$.

**The Exclusion Rule for Functions and Functional Methods:** Suppose that $f(P)$ and $f(Q)$ are two distinct function or functional method declarations in a single scope. If $P \diamondsuit Q$ then $f(P)$ and $f(Q)$ are a valid overloading.

**The Exclusion Rule for Dotted Methods:** Suppose that $P_0.f(P)$ and $Q_0.f(Q)$ are two distinct dotted method declarations provided by a trait or object $C$. If $P \diamondsuit Q$ then $P_0.f(P)$ and $Q_0.f(Q)$ are a valid overloading.

### 15.6.4 Meet Rule

If neither the Subtype Rule nor the Incompatibility Rule holds for a pair of overloaded declarations then the declarations introduce the possibility of ambiguity. This ambiguity can be eliminated by a *disambiguating declaration*; this is, for every call to which both declarations are applicable, a third, more specific, declaration that is also applicable can resolve the ambiguity. Thus, at run time, neither of the pair of declarations is executed because the disambiguating declaration is also applicable, and it is more specific than both.

**The Meet Rule for Functions:** Suppose that $f(P)$ and $f(Q)$ are two function declarations in a single scope such that neither $P$ nor $Q$ is a subtype of the other and $P$ and $Q$ do not exclude each other. $f(P)$ and $f(Q)$ are a valid overloading if there is a declaration $f(P \cap Q)$ in the scope.

We write $P \cap Q$ to denote the intersection of types $P$ and $Q$. If for some type $S$ we have $S \preceq P$ and $S \preceq Q$ then $S \preceq (P \cap Q)$, but it is not necessarily the case that $S = (P \cap Q)$ since another type may be more specific than both $P$ and $Q$.

Unlike for functions, the Meet Rule for dotted methods applies only to dotted methods that are provided by the same trait or object. This is possible because two dotted methods are applicable to a given call $A_0.f(A)$ only if they are both provided by the trait or object $A_0$.

**The Meet Rule for Dotted Methods:** Suppose that $P_0.f(P)$ and $Q_0.f(Q)$ are two dotted method declarations provided by a trait or object $C$ such that neither $(P_0, P)$ nor $(Q_0, Q)$ is a subtype of the other and $P$ and $Q$ do not exclude each other. $P_0.f(P)$ and $Q_0.f(Q)$ are a valid overloading if there is a declaration $R_0.f(P \cap Q)$ provided by $C$ with $R_0 \preceq (P_0 \cap Q_0)$.

Recall that functional methods can be viewed semantically as top-level functions, as described in Section 10.2. However, treating functional methods as top-level functions for determining valid overloading is too restrictive. In the following example:

```
trait ℤ
  opr @(self): ℤ
end
trait ℝ
  opr @(self): ℝ
end
```

if the functional methods were interpreted as top-level functions then this program would define two top-level functions with parameter types ℤ and ℝ. These declarations would be an invalid overloading because there is no relation between ℤ and ℝ; another trait may extend them both without declaring its own version of the functional method which may lead to an ambiguous call at run time. However, notice that declarations are ambiguous only for calls on arguments that extend both ℤ and ℝ, and any type that extends both can include a new declaration that disambiguates them. We use this intuition to allow such overloadings.

**The Meet Rule for Functional Methods:** Suppose that $f(P)$ and $f(Q)$ are two functional method declarations occurring in trait or object declarations or object expressions such that neither $P$ nor $Q$ is a subtype of the other and $P$ and $Q$ do not exclude each other. Let $f(P)$ and $f(Q)$ have self parameters at $i$ and $j$ respectively. $f(P)$ and $f(Q)$ are a valid overloading if all of the following hold:

- $i = j$
- if there exists a trait or object $C$ that provides both $f(P)$ and $f(Q)$ then $P \neq Q$ and there is a declaration $f(P \cap Q)$ provided by $C$ having self parameter at $i$.

Notice that the Meet Rule for functional methods requires the self parameters of two overloaded declarations to be in the same position. This requirement guarantees that no ambiguity is caused by the position of the self parameter. Two declarations which differ in the position of the self parameter must satisfy either the Subtype Rule or the Exclusion Rule to be a valid overloading.

Functional method declarations can overload with function declarations. A valid overloading between a functional method declaration and a function declaration is determined by applying the (more restrictive) Meet Rule for functions to both declarations.

# Chapter 16

# Operators

Operators are like functions or methods; operator declarations are described in Chapter 22 and operator applications are described in Section 13.7, Section 13.17, and Section 13.28. Just as functions or methods may be overloaded (see Chapter 15 for a discussion of overloading), so operators may have overloaded declarations of the same fixity. Operator declarations with the same operator name but with different fixities are valid declarations because it is always unambiguous which declaration shall be applied to an application of the operator. Calls to overloaded operators are resolved first via the fixity of the operators based on the context of the calls. Then, among the applicable declarations with that fixity, the most specific declaration is chosen.

Most operators can be used as prefix, infix, postfix, or nofix operators as described in Section 16.3 (nofix operators take no arguments); the fixity of an operator is determined syntactically. A simple example is that '$-$' may be either infix or prefix, as is conventional. As another example, '$!$' may be a postfix operator that computes factorial when applied to integers. These operators might not be used as enclosing operators.

Several pairs of operators can be used as enclosing operators. Any number of '$|$' (vertical line) can be used as both infix operators and enclosing operators.

Some operators are always postfix: a '^' followed by any ordinary operator (with no intervening whitespace) is considered to be a superscripted postfix operator. For example, '$^*$' and '$^+$' and '$^?$' are available for use as part of the syntax of extended regular expressions. As a very special case, '$^T$' is also considered to be a superscripted postfix operator, typically used to signify matrix transposition.

Finally, there are special operators such as juxtaposition. Juxtaposition may be a function application or an infix operator in Fortress. When the left-hand-side expression is a function, juxtaposition performs function application; when the left-hand-side expression is a number, juxtaposition conventionally performs multiplication; when the left-hand-side expression is a string, juxtaposition conventionally performs string concatenation.

## 16.1 Operator Names

To support a rich mathematical notation, Fortress allows most Unicode characters that are specified to be mathematical operators to be used as operators in Fortress expressions, as well as these characters:

```
!   @   #   $   %   *   +   -   =   |   :   <   >   /   ?   ^   ~
```

In addition, a token that is made up of a mixture of uppercase letters and underscores (but no digits), does not begin or end with an underscore, and contains at least two different letters is also considered to be an operator:

```
MAX   MIN
```

The above operators are rendered as: `MAX MIN`. (See Section 4.13 and Appendix C for detailed descriptions of operator names in Fortress.)

## 16.2 Operator Precedence and Associativity

Fortress specifies that certain operators have higher precedence than certain other operators and certain operators are associative, so that one need not use parentheses in all cases where operators are mixed in an expression. (See Appendix C for a detailed description of operator precedence and associativity in Fortress.) However, Fortress does not follow the practice of other programming languages in simply assigning an integer to each operator and then saying that the precedence of any two operators can be compared by comparing their assigned integers. Instead, Fortress relies on defining traditional groups of operators based on their meaning and shape, and specifies specific precedence relationships between some of these groups. If there is no specific precedence relationship between two operators, then parentheses must be used. For example, Fortress does not accept the expression $a + b \cup c$; one must write either $(a + b) \cup c$ or $a + (b \cup c)$. (Whether or not the result then makes any sense depends on what definitions have been made for the $+$ and $\cup$ operators—see Chapter 22.)

Here are the basic principles of operator precedence and associativity in Fortress:

- Member selection (`.`) and method invocation ($.name(\ldots)$) are not operators. They have higher precedence than any operator listed below.
- Subscripting (`[ ]` and any kind of subscripting operators, which can be any kind of enclosing operators), superscripting (`^`), and postfix operators have higher precedence than any operator listed below; within this group, these operations are left-associative (performed left-to-right).
- *Tight juxtaposition*, that is, juxtaposition without intervening whitespace, has higher precedence than any operator listed below. The associativity of tight juxtaposition is type-dependent; see Section 16.8.
- Next, *tight fractions*, that is, the use of the operator '$/$' with no whitespace on either side, have higher precedence than any operator listed below. The tight-fraction operator has no precedence compared with itself, so it is not permitted to be used more than once in a tight fraction without use of parentheses.
- *Loose juxtaposition*, that is, juxtaposition with intervening whitespace, has higher precedence than any operator listed below. The associativity of loose juxtaposition is type-dependent and is different from that for tight juxtaposition; see Section 16.8. Note that *lopsided juxtaposition* (having whitespace on one side but not the other) is a static error as described in Section 16.3.
- Prefix operators have higher precedence than any operator listed below. However, it is a static error for an operand of a loose prefix operator to be an operand of a tight infix operator.
- The infix operators are partitioned into certain traditional groups, as explained below. They have higher precedence than any operator listed below.
- The equal symbol '$=$' in binding context, the assignment operator '$:=$', and compound assignment operators ($+=$, $-=$, $\wedge=$, $\vee=$, $\cap=$, $\cup=$, and so on as described in Section 13.7) have lower precedence than any operator listed above. Note that compound assignment operators themselves are not operator names.

The infix binary operators are divided into four general categories: arithmetic, relational, boolean, and other. The arithmetic operators are further categorized as multiplication/division/intersection, addition/subtraction/union, and other. The relational operators are further categorized as equivalence, inequivalence, ordering, and other. The boolean operators are further categorized as conjunctive, disjunctive, and other.

The arithmetic and relational operators are further divided into groups based on shape:

- “ordinary” operators: + − · × / ± ∓ ⊕ ⊖ ⊙ ⊗ ⊘ ⊞ ⊟ ⊡⊠ <≤≥>≪⋘⋙≫≮≰≱≯ etc.

  The arithmetic operations in this group are further subdivided into “plain” ( + − · × / ± ∓ etc.), “circled” ( ⊕ ⊖ ⊙ ⊗ ⊘ etc.), “boxed” ( ⊞ ⊟ ⊡⊠ etc.), and so on; any of these groups may be used with the plain relational operators ( <≤≥>≪⋘⋙≫≮≰≱≯ etc.), but the groups might not be mixed.

- “rounded horseshoe” or “set” operators: ∩ ⋒ ∪ ⋓ ⊎ ⊂⊆⊇⊃⋐⋑⊄⊈⊉⊅ etc.

- “square horseshoe” operators: ⊓⊔ ⊏⊑⊒⊐⋢⋣ etc.

- “curly” operators: ⋏⋎ ≺≼≽≻⊀⋠⋡⊁ etc.

- “triangular” relations: ◁ ⊴ ⊵▷ ⋪⋬⋭⋫ etc.

- “chickenfoot” relations: <-> etc.

The principles of precedence for binary operators are then as follows:

- A multiplication or division or intersection operator has higher precedence than any addition or subtraction or union operator that is in the same shape group.

- Certain addition and subtraction operators come in pairs, such as $+$ and $-$, or $\oplus$ and $\ominus$, which are considered to have the same precedence and so may be mixed within an expression and are grouped left-associatively. These addition-subtraction pairs are the *only* cases where two different operators are considered to have the same precedence.

- An arithmetic operator has higher precedence than any equivalence or inequivalence operator.

- An arithmetic operator has higher precedence than any relational operator that is in the same shape group.

- A relational operator has higher precedence than any boolean operator.

- A conjunctive boolean operator has higher precedence than any disjunctive boolean operator.

While the rules of precedence are complicated, they are intended to be both unsurprising and conservative. Note that operator precedence in Fortress is not always transitive; for example, while $+$ has higher precedence than $<$ (so you can write $a + b < c$ without parentheses), and $<$ has higher precedence than $\vee$ (so you can write $a < b \vee c < d$ without parentheses), it is *not* true that $+$ has higher precedence than $\vee$—the expression $a \vee b + c$ is not permitted, and one must instead write $(a \vee b) + c$ or $a \vee (b + c)$.

Another point is that the various multiplication and division operators do *not* have “the same precedence”; they may not be mixed freely with each other. For example, one cannot write $u \cdot v \times w$; one must write $(u \cdot v) \times w$ or (more likely) $u \cdot (v \times w)$. Similarly, one cannot write $a \odot b \;/\; c \odot d$; but juxtaposition does bind more tightly than a loose (whitespace-surrounded) division slash, so one is allowed to write $a\,b \;/\; c\,d$, and this means the same as $(a\,b)/(c\,d)$. On the other hand, loose juxtaposition binds less tightly than a tight division slash, so that $a\;b/c\;d$ means the same as $a\;(b/c)\;d$. On the other other hand, tight juxtaposition binds more tightly than tight division, so that $(n+1)/(n+2)(n+3)$ means the same as $(n+1)/((n+2)(n+3))$.

There are two additional rules intended to catch misleading code: it is a static error for an operand of a tight infix or tight prefix operator to be a loose juxtaposition, and it is a static error if the rules of precedence determine that a use of infix operator $a$ has higher or equal precedence than a use of infix operator $b$, but that particular use of $a$ is loose and that particular use of $b$ is tight. Thus, for example, the expression $\sin\; x + y$ is permitted, but $\sin\; x{+}y$ is not permitted. Similarly, the expression $a \cdot b + c$ is permitted, as are $a{\cdot}b + c$ and $a{\cdot}b{+}c$, but $a \cdot b{+}c$ is not permitted. (The rule detects only the presence or absence of whitespace, not the amount of whitespace, so $a \quad \cdot\; b + c$ is permitted. You have to draw the line somewhere.)

When in doubt, just use parentheses. If there’s a problem, the compiler will (probably) let you know.

## 16.3 Operator Fixity

Most operators in Fortress can be used variously as prefix, postfix, infix, or nofix operators. (See Section 16.4 for a discussion of how infix operators may be chained.) Some operators can be used in pairs as enclosing (bracketing) operators—see Section 16.5. The Fortress language dictates only the rules of syntax; whether an operator has a meaning when used in a particular way depends only on whether there is a definition in the program for that operator when used in that particular way (see Chapter 22).

The fixity of a non-enclosing operator is determined by context. To the left of such an operator we may find (1) a *primary tail* (described below), (2) another operator, or (3) a comma, semicolon, or left encloser. To the right we may find (1) a primary tail, (2) another operator, (3) a comma, semicolon, or right encloser, or (4) a line break. A primary tail is an identifier, a literal, a right encloser, or a superscripted postfix operator (exponent operator). Considered in all combinations, this makes twelve possibilities. In some cases one must also consider whether or not whitespace separates the operator from what lies on either side. The rules of operator fixity are specified by Figure 16.1, where the center column indicates the fixity that results from the left and right context specified by the other columns.

| **left context** | **whitespace** | **operator fixity** | **whitespace** | **right context** |
|---|---|---|---|---|
| primary tail | yes | **infix** | yes | primary tail |
| | yes | **error** (infix) | no | |
| | no | **postfix** | yes | |
| | no | **infix** | no | |
| primary tail | yes | **infix** | yes | operator |
| | yes | **error** (infix) | no | |
| | no | **postfix** | yes | |
| | no | **infix** | no | |
| primary tail | yes | **error** (postfix) | | `,` `;` right encloser |
| | no | **postfix** | | |
| primary tail | yes | **infix** | | line break |
| | no | **postfix** | | |
| operator | | **prefix** | | primary tail |
| operator | | **prefix** | | operator |
| operator | | **error** (nofix) | | `,` `;` right encloser |
| operator | | **error** (nofix) | | line break |
| `,` `;` left encloser | | **prefix** | | primary tail |
| `,` `;` left encloser | | **prefix** | | operator |
| `,` `;` left encloser | | **nofix** | | `,` `;` right encloser |
| `,` `;` left encloser | | **error** (prefix) | | line break |

Figure 16.1: Operator Fixity (I)

A case described in the center column of the table as an **error** is a static error; for such cases, the fixity mentioned in parentheses is the recommended treatment of the operator for the purpose of attempting to continuing the parse in search of other errors.

The table may seem complicated, but it all boils down to a couple of practical rules of thumb:

1. *Any* operator can be prefix, postfix, infix, or nofix.

2. An infix operator can be *loose* (having whitespace on both sides) or *tight* (having whitespace on neither side), but it mustn't be *lopsided* (having whitespace on one side but not the other).

3. A postfix operator must have no whitespace before it and must be followed (possibly after some whitespace) by a comma, semicolon, right encloser, or line break.

| **left context** | **whitespace** | **operator fixity** | **whitespace** | **right context** |
|---|---|---|---|---|
| primary tail | yes | **infix** | yes | primary tail |
| | yes | **left encloser** | no | |
| | no | **right encloser** | yes | |
| | no | **infix** | no | |
| primary tail | yes | **infix** | yes | operator |
| | yes | **left encloser** | no | |
| | no | **right encloser** | yes | |
| | no | **infix** | no | |
| primary tail | yes | **error** (right encloser) | | `,` `;` right encloser |
| | no | **right encloser** | | |
| primary tail | yes | **infix** | | line break |
| | no | **right encloser** | | |
| operator | | **error** (left encloser) | yes | primary tail |
| | | **left encloser** | no | |
| operator | | **error** (left encloser) | yes | operator |
| | | **left encloser** | no | |
| operator | | **error** (nofix) | | `,` `;` right encloser |
| operator | | **error** (nofix) | | line break |
| `,` `;` left encloser | | **left encloser** | | primary tail |
| `,` `;` left encloser | | **left encloser** | | operator |
| `,` `;` left encloser | | **nofix** | | `,` `;` right encloser |
| `,` `;` left encloser | | **error** (left encloser) | | line break |

Figure 16.2: Operator Fixity (II)

## 16.4 Chained Operators

Certain infix mathematical operators that are traditionally regarded as *relational* operators, delivering boolean results, may be *chained*. For example, an expression such as $A \subseteq B \subset C \subseteq D$ is treated as being equivalent to $(A \subseteq B) \wedge (B \subset C) \wedge (C \subseteq D)$ except that the expressions $B$ and $C$ are evaluated only once (which matters only if they have side effects such as writes or input/output actions). Similarly, the expression $A \subseteq B = C \subset D$ is treated as being equivalent to $(A \subseteq B) \wedge (B = C) \wedge (C \subset D)$, except that $B$ and $C$ are evaluated only once. Fortress restricts such chaining to a mixture of equivalence operators and ordering operators; if a chain contains two or more ordering operators, then they must be of the same kind and have the same sense of monotonicity; for example, neither $A \subseteq B \leq C$ nor $A \subseteq B \supset C$ is permitted. This transformation is done before type checking. In particular, it is done even if these operators do not return boolean values, and the resulting expression is checked for type correctness. (See Section C.4 for a detailed description of which operators may be chained.)

## 16.5 Enclosing Operators

These operators are always used in pairs as enclosing operators:

```
               (/   /)       (\   \)
[   ]          [/   /]                     [*   *]
{   }          {/   /}       {\   \}       {*   *}
               </   />       <\   \>
              <</   />>     <<\   \>>
```

(ASCII encodings are shown here; they all correspond to particular single Unicode characters.) There are other pairs as well, such as ⌊ ⌋ and ⌈ ⌉ and multicharacter enclosing operators described in Section 4.13.1. Note that the pairs `(` `)` and `[\` `\]` (also known as ⟦ ⟧) are not operators; they play special roles in the syntax of Fortress, and their behavior cannot be redefined by a library. The bracket pairs that may be used as enclosing operators are described in Section C.1.

Any number of '`|`' (vertical line) may also be used in pairs as enclosing operators but there is a trick to it, because on the face of it you can't tell whether any given occurrence is a left encloser or a right encloser. Again, context is used to decide, this time according to Figure 16.2.

This is very similar to Figure 16.1 in Section 16.3; a rough rule of thumb is that if an ordinary operator would be considered a prefix operator, then one of these will be considered a left encloser; and if an ordinary operator would be considered a postfix operator, then one of these will be considered a right encloser.

In this manner, one may use `|...|` for absolute values and `||...||` for matrix norms.

## 16.6 Conditional Operators

If a binary operator other than '`:`' is immediately followed by a '`:`' then it is *conditional*: evaluation of the right-hand operand cannot begin until evaluation of the left-hand operand has completed, and whether or not the right-hand operand is evaluated may depend on the value of the left-hand operand. If the left-hand operand throws an exception, then the right-hand operand is not evaluated.

The Fortress standard libraries define several conditional operators on boolean values including `∧:` and `∨:`.

See Section 22.8 for a discussion of how conditional operators are declared.

## 16.7 Big Operators

A big operator application is either a *reduction expression* described in Section 13.17 or a *comprehension* described in Section 13.28.

The Fortress standard libraries define several big operators including $\sum$, $\prod$, and `BIG ∧`.

See Section 22.9 for a discussion of how big operators are declared.

## 16.8 Juxtaposition

Juxtaposition in Fortress may be a function call or a special infix operator. The Fortress standard libraries include several declarations of a `juxtaposition` operator.

When two expressions are juxtaposed, the juxtaposition is interpreted as follows: if the left-hand-side expression is a function, juxtaposition performs function application; otherwise, juxtaposition performs the `juxtaposition` operator application.

The manner in which a juxtaposition of three or more items must be associated requires type information and awareness of whitespace. (This is an inherent property of customary mathematical notation, which Fortress is designed to emulate where feasible.) Therefore a Fortress compiler must produce a provisional parse in which such multi-element juxtapositions are held in abeyance, then perform a type analysis on each element and use that information to rewrite the n-ary juxtaposition into a tree of binary juxtapositions. All we need to know is whether each element of a juxtaposition has an arrow type.

The rules for reassociating a loose juxtaposition are as follows:

- First the loose juxtaposition is broken into nonempty chunks; wherever there is a non-function element followed by a function element, the latter begins a new chunk. Thus a chunk consists of some number (possibly zero) of functions followed by some number (possibly zero) of non-functions.
- The non-functions in each chunk, if any, are replaced by a single element consisting of the non-functions grouped left-associatively into binary juxtapositions.
- What remains in each chunk is then grouped right-associatively.
- Finally, the sequence of rewritten chunks is grouped left-associatively.

(Notice that no analysis of the types of newly constructed chunks is needed during this process.)

Here is an example: $n\;(n+1)\;\sin\;3\,n\,x\;\log\;\log\;x$. Assuming that $\sin$ and $\log$ name functions in the usual manner and that $n$, $(n+1)$, and $x$ are not functions, this loose juxtaposition splits into three chunks: $n\;(n+1)$ and $\sin\;3\,n\,x$ and $\log\;\log\;x$. The first chunk has only two elements and needs no further reassociation. In the second chunk, the non-functions $3\,n\,x$ are replaced by $((3\;n)\;x)$. In the third chunk, there is only one non-function, so that remains unchanged; the chunk is the right-associated to form $(\log\;(\log\;x))$. Finally, the three chunks are left-associated, to produce the final interpretation $((n\;(n+1))\;(\sin\;((3\;n)\;x)))\;(\log\;(\log\;x))$. Now the original juxtaposition has been reduced to binary juxtaposition expressions.

A tight juxtaposition is always left-associated if it contains any dot (i.e., ".") not within parentheses or some pair of enclosing operators. If such a tight juxtaposition begins with an identifier immediately followed by a dot then the maximal prefix of identifiers separated by dots (whitespace may follow but not precede the dots) is collected into a "dotted id chain", which is subsequently partitioned into the dots and identifiers, which are interpreted as selectors. If the last identifier in the dotted id chain is immediately followed by a left parenthesis, then the last selector together with the subsequent parenthesis-delimited expression is a method invocation.

A tight juxtaposition without any dots might not be entirely left-associated. Rather, it is considered as a nonempty sequence of elements: the front expression, any "math items", and any postfix operator, and subject to reassociation as described below. A math item may be a subscripting, an exponentiation, or one of a few kinds of expressions. It is a static error if an exponentiation is immediately followed by a subcripting or an exponentiation.

The procedure for reassociation is as follows:

- For each expression element (i.e., not a subscripting, exponentiation or postfix operator), determine whether it is a function.
- If some function element is immediately followed by an expression element then, find the first such function element, and call the next element the argument. It is a static error if either the argument is not parenthesized, or the argument is immediately followed by a non-expression element. Otherwise, replace the function and argument with a single element that is the application of the function to the argument. This new element is an expression. Reassociate the resulting sequence (which is one element shorter).
- If there is any non-expression element (it cannot be the first element) then replace the first such element and the element immediately preceding it (which must be an expression) with a single element that does the appropriate operator application. This new element is an expression. Reassociate the resulting sequence (which is one element shorter).
- Otherwise, left-associate the sequence, which has only expression elements, only the last of which may be a function.

(Note that this process requires type analysis of newly created chunks along the way.)

Here is an (admittedly contrived) example: $reduce(f)(a)(x+1)atan(x+2)$. Suppose that $reduce$ is a curried function that accepts a function $f$ and returns a function that can be applied to an array $a$ (the idea is to use the function $f$, which ought to take two arguments, to combine the elements of the array to produce an accumulated result).

The leftmost function is $reduce$, and the following element $(f)$ is parenthesized, so the two elements are replaced with one: $(reduce(f))(a)(x+1)\,atan(x+2)$. Now type analysis determines that the element $(reduce(f))$ is a function.

The leftmost function is $(reduce(f))$, and the following element $(a)$ is parenthesized, so the two elements are replaced with one: $((reduce(f))(a))(x+1)\,atan(x+2)$. Now type analysis determines that the element $((reduce(f))(a))$ is not a function.

The leftmost function is $atan$, and the following element $(x+2)$ is parenthesized, so the two elements are replaced with one: $((reduce(f))(a))(x+1)(atan(x+2))$. Now type analysis determines that the element $(atan(x+2))$ is not a function.

There are no functions remaining in the juxtaposition, so the remaining elements are left-associated:

$$(((reduce(f))(a))(x+1))(atan(x+2))$$

Now the original juxtaposition has been reduced to binary juxtaposition expressions.

# Chapter 17

# Tests

To help make programs more robust, the `test` modifier can appear on a top-level function definition with type "$() \rightarrow ()$". Functions with modifier `test` must not be overloaded with functions that do not have modifier `test`. The collection of all functions in a program that include modifier `test` are referred to collectively as the program's *tests*. Tests can refer to non-tests but non-tests must not refer to any test.

For example, we can write the following test function:

```
test testFactorial_1() = do
    assert(fact(0) = 1)
    assert(fact(5) = 120)
    println("testFactorial1 passed");
end
```

When a program's tests are *run*, each top-level function definition with modifier `test` is run in textual program order.

# Chapter 18

# Type Inference

*In order to synchronize the Fortress language specification with the implementation, it was necessary to drop static type inference from the specification. Contrary to the Fortress Language Specification, Version 1.0 $\beta$, inference of static parameter instantiations is based on the runtime types of the arguments to a functional call as described below. This dynamic type inference will be replaced by a static type inference when it is implemented.*

Before describing the dynamic type inference procedure, we introduce some terminology. We say that an arrow type is an *outermost* arrow type if and only if it is not a constituent type of another arrow type. We define a *covariant context* and a *contravariant context* as follows:

1. The parameter type of an outermost arrow type is a contravariant context.
2. The return type of an outermost arrow type is a covariant context.
3. The parameter type of an arrow type in a covariant context is a contravariant context.
4. The parameter type of an arrow type in a contravariant context is a covariant context.

For example, in the following declaration:

$f[\![W, T, U, V]\!](g : (W, T \to U) \to V) : ()$

$W$ and $U$ are in a covariant context and $T$ and $V$ are in a contravariant context.

Given an invocation of a generic functional (or a non-generic functional method declared by a generic trait or object) $f$ and its most specific declaration, types are inferred for the static parameters of the generic functional (or the trait or object declaring the functional method). For each static parameter $\alpha$, we accumulate a set of constraints as follows:

First, if the static parameter is a type parameter with a bound $T$ declared as "$\alpha$ `extends` $T$", we add the following constraint to the set:

$\text{BottomType} <: \alpha <: T$

Second, if the static parameter is a type parameter without a bound, we add the following constraint to the set:

$\text{BottomType} <: \alpha <: \text{Any}$

Third, we collect constraints on $\alpha$ by unifying the types of the parameters and the types of the arguments of the functional. For the self parameter of a non-generic functional method declared by a generic trait or object $T[\![U, V, W]\!]$, when the self parameter is bound to a value of runtime type $S$ where $S$ has a supertype $T[\![X, Y, Z]\!]$, we use $T[\![X, Y, Z]\!]$ as the type of the self parameter. Note the assumption that there is at most one supertype that is an instantiation of the generic type $T$ which declares the most specific declaration of $f$. If there exists more than one such supertype, inference fails.

Once we have accumulated all constraints for each static parameter, the constraints are solved to get the inferred type or value for the static parameter as follows:

First, for each static parameter occurring in a covariant context, we infer the least specific type or value for the static parameter.

Second, for each remaining static parameter which occurs in a non-covariant context, we infer the most specific type or value for the static parameter.

# Chapter 19

# Memory Model

Fortress programs are highly multithreaded by design; the language makes it easy to expose parallelism. However, many Fortress objects are mutable; without judicious use of synchronization constructs—reductions and `atomic` expressions—*data races* will occur and programs will behave in an unpredictable way. The memory model has two important functions:

1. Define a programming discipline for the use of data and synchronization, and describe the behavior of programs that obey this discipline. This is the purpose of Section 19.2.

2. Define the behavior of programs that do not obey the programming discipline. This constrains the optimizations that can be performed on Fortress programs. The remaining sections of this chapter specify the detailed memory model that Fortress programs must obey.

## 19.1 Principles

The Fortress memory model has been written with several important guiding principles in mind. Violations of these principles must be taken as a flaw in the memory model specification rather than an opportunity to be exploited by the programmer or implementor. The most important principle is this: violations of the Fortress memory model must still respect the underlying data abstractions of the Fortress programming language. All data structures must be properly initialized before they can be read by another thread, and a program must not read values that were never written. When a program fails, it must fail gracefully by throwing an exception.

The second goal is nearly as important, and much more difficult: present a memory model which can be understood thoroughly by programmers and implementors. It must never be difficult to judge whether a particular program behavior is permitted by the model. Where possible, it must be possible to check that a program obeys the programming discipline.

The final goal of the Fortress memory model is to permit aggressive optimization of Fortress programs. A multiprocessor memory model can rule out optimizations that might be performed by a compiler for a uniprocessor. The Fortress memory model attempts to rule out as few behaviors as possible, but more importantly attempts to make it easy to judge whether a particular optimization is permitted or not. The semantics of Fortress already allows permissive operation reordering in many programs, simply by virtue of the implicitly parallel semantics of tuple evaluation and looping.

## 19.2 Programming Discipline

If Fortress programmers obey the following discipline, they can expect sequentially consistent behavior from their Fortress programs:

- Updates to shared mutable locations must always be performed using an `atomic` expression. A location is considered to be shared if and only if that location can be accessed by more than one thread at a time. For example, statically partitioning an array among many threads need not make the array elements shared; only elements actually accessed by more than one thread are considered to be shared.
- Within a thread or group of implicit threads objects must not be accessed through aliased references; this can yield unexpected results. Section 19.2.2 defines the notion of *apparently disjoint* references. An object must not be written through one reference when it is accessed through another apparently disjoint reference.

The following stylistic guidelines reduce the possibility of pathological behavior when a program strays from the above discipline:

- Where feasible, reduction must be used in favor of updating a single shared object.
- Immutable fields and variables must be used wherever practical. We discuss this further in Section 19.2.1.

### 19.2.1 Immutability

Recall from Section 5.3 that we can distinguish mutable and immutable memory locations. Any thread that reads an immutable field will obtain the initial value written when the object was constructed. In this regard it is worth re-emphasizing the distinction between an object reference and the object it refers to. A location that does not contain a value object contains an object reference. If the field is immutable, that means the reference is not subject to change; however, fields of the object referred to may still be modified in accordance with the memory model.

### 19.2.2 Modifying Aliased Objects

In common with Fortran, and unlike most other popular programming languages, Fortress gives special treatment to accesses to a location through different aliases. For the purposes of variable aliasing, it is important to define the notion of *apparently disjoint* (or simply disjoint) object and field references. If two references are not disjoint, we say they are *certainly the same*, or just *the same*. By contrast, we say object references are *identical* if they refer to the same object, and *distinct* otherwise. Accesses to fields reached via apparently disjoint object references may be reordered (except an initializing write is never reordered with respect to other accesses to the identical location).

Distinct references are always disjoint. Two identical references are apparently disjoint if they are obtained in any of the following ways:

- distinct parameters of a single function call
- distinct fields
- a parameter and a field
- identically named fields read from apparently disjoint object references
- distinct reads of a single location for which there may be an interposing write

When comparing variables defined in different scopes, these rules will eventually lead to reads of fields or to reads of parameters in some common containing scope.

We extend this to field references as follows: two field references are apparently disjoint if they refer to distinct fields, or they refer to identically named fields read from apparently disjoint object references.

Consider the following example:

```
$f(x\colon \mathbb{Z}64[2], y\colon \mathbb{Z}64[2])\colon \mathbb{Z}64 =$ do
  $x_0 := 17$
  $y_0 := 32$
end
```

Here $x$ and $y$ in $f$ are apparently disjoint; the writes may be reordered, so the call $f(a, a)$ may assign either 17 or 32 to $a_0$.

A similar phenomenon occurs in the following example:

```
$g(x\colon \mathbb{Z}64[2], y\colon \mathbb{Z}64[2])\colon \mathbb{Z}64 =$ do
  $x_0 := 17$
  $y_0$
end
```

Again $x$ and $y$ are apparently distinct in $g$, so the write to $x_0$ and the read of $y_0$ may be reordered. The call $g(a, a)$ will assign 17 to $a_0$ but may return either the initial value of $a_0$ or 17.

It is safe to *read* an object through apparently disjoint references:

```
$h(x\colon \mathbb{Z}64[2], y\colon \mathbb{Z}64[2])\colon \mathbb{Z}64 =$ do
  $u\colon \mathbb{Z}64 = x_0$
  $v\colon \mathbb{Z}64 = y_0$
  $u + v$
end
```

A call to $h(a, a)$ will read $a_0$ twice without ambiguity. Note, however, that the reads may still be reordered, and if $a_0$ is written in parallel by another thread this reordering can be observed.

If necessary, `atomic` expressions can be used to order disjoint field references:

```
$f'(x\colon \mathbb{Z}64[2], y\colon \mathbb{Z}64[2])\colon () =$ do
  atomic $x_0 := 17$
  atomic $y_0 := 32$
end
```

Here the call $f'(a, a)$ ends up setting $a_0$ to 32. Note that simply using a single `atomic` expression containing one or both writes is not sufficient; the two writes must be in distinct `atomic` expressions to be required to occur in order.

When references occur in distinct calling contexts, they are disambiguated at the point of call:

```
$j(x\colon \mathbb{Z}64[2], y\colon \mathbb{Z}64)\colon () = x_0 := y$
$k(x\colon \mathbb{Z}64[2])\colon () =$ do
  $j(x, 17)$
  $j(x, 32)$
end
$l(x\colon \mathbb{Z}64[2], y\colon \mathbb{Z}64[2])\colon () =$ do
  $j(x, 17)$
  $j(y, 32)$
end
```

Here if we call $k(a)$ the order of the writes performed by the two calls to $j$ is unambiguous, and $a_0$ is 32 in the end. By contrast, $l(a, a)$ calls $j$ with two apparently disjoint references, and the writes in these two calls may thus be reordered.

## 19.3 Read and Write Atomicity

Any read or write to a location is *indivisible*. In practical terms, this means that each read operation will see exactly the data written by a single write operation. Note in particular that indivisibility holds for a mutable location containing a large value object. It is convenient to imagine that every access to a mutable location is surrounded by an `atomic` expression. However, there are a number of ordering guarantees provided by `atomic` accesses that are not respected by non-`atomic` accesses.

## 19.4 Ordering Dependencies among Operations

The Fortress memory model is specified in terms of two orderings: dynamic program order (see Chapter 5 and Chapter 13) and memory order. The actual order of memory operations in a given program execution is *memory order*, a total order on all memory operations. Dynamic program order constrains memory order. However, memory operations need not be ordered according to dynamic program order; many memory operations, even reads and writes to a single field or array element, can be reordered. Programmers who adhere to the model in Section 19.2 can expect sequentially consistent behavior: the global ordering of memory operations will respect dynamic program order.

Here is a summary of the salient aspects of memory order:

- There is a single memory order which is respected in all threads.
- Every read obtains the value of the immediately preceding write to the identical location in memory order.
- Memory order on `atomic` expressions respects dynamic program order.
- Memory order respects dynamic program order for operations that certainly access the same location.
- Initializing writes are ordered before any other memory access to the same location.

### 19.4.1 Dynamic Program Order

Much of the definition of *dynamic program order* is given in the descriptions of individual expressions in Chapter 13. It is important to understand that dynamic program order represents a conceptual, naive view of the order of operations in an execution; this naive view is used to define the more permissive memory order permitted by the memory model. Dynamic program order is a partial order, rather than a total order; in most cases operations in different threads will not be ordered with respect to one another. There is an important exception: there is an ordering dependency among threads when a thread starts or must be complete.

An expression is ordered in dynamic program order after any expression it dynamically contains, with one exception: a `spawn` expression is dynamically ordered before any subexpression of its body. The body of the `spawn` is dynamically ordered before any point at which the spawned thread object is observed to have completed.

Only expressions whose evaluation completes normally occur in dynamic program order, unless the expression is "directly responsible" for generating abrupt termination. Examples of the latter case are `throw` and `exit` expressions and division by zero. In particular, when the evaluation of a subexpression of an expression completes abruptly, causing the expression itself to complete abruptly, the containing expression does not occur in dynamic program order. A `label` block is ordered after an `exit` that targets it. The expressions in a `catch` clause whose `try` block throws a matching exception are ordered after the `throw` and before any expression in the `finally` clause. If the `catch` completes normally, the `try` block as a whole is ordered after the expressions in the `finally` clause. For this reason, when we refer to the place of non-`spawn` expression in dynamic program order, we mean the expression or any expression it dynamically contains.

For any construct giving rise to implicit threads—tuple evaluation, function or method call, or the body of an expression with generators such as `for`—there is no ordering in dynamic program order between the expression executed in each thread in the group. These subexpressions are ordered with respect to expressions which precede or succeed the group.

When a function or method is called, the body of the function or method occurs dynamically after the arguments and function or receiver; the call expression is ordered after the body of the called function or method.

For conditional expressions such as `if`, `case`, and `typecase`, the expression being tested is ordered dynamically before any chosen branch. This branch is in turn ordered dynamically before the conditional expression itself.

Iterations of the body of a `while` loop are ordered by dynamic program order. Each evaluation of the guarding predicate is ordered after any previous iteration and before any succeeding iteration. The `while` loop as a whole is ordered after the final evaluation of the guarding predicate, which yields *false*.

An iteration of the body of a `for` loop, and each evaluation of the body expression in a comprehension or big operator, is ordered after the generator expressions.

### 19.4.2 Memory Order

*Memory order* gives a total order on all memory accesses in a program execution. A read obtains the value of the most recent prior write to the identical location in memory order. In this section we describe the constraints on memory order, guided by dynamic program order. We can think of these constraints as specifying a partial order which must be respected by memory order. The simplest constraint is that accesses certainly to the same location must respect dynamic program order. Apparently disjoint accesses need not respect dynamic program order, but an initializing write must be ordered before all other accesses to the identical location in program order.

Accesses in distinct (non-nested) `atomic` expressions respect dynamic program order. Given an `atomic` expression, we divide accesses into four classes:

1. Constituents, dynamically contained within the `atomic` expression.
2. Ancestors, dynamically ordered before the `atomic` expression.
3. Descendants, dynamically ordered after the `atomic` expression.
4. Peers, dynamically unordered with respect to operations dynamically contained within the `atomic` expression.

We say an `atomic` expression is *effective* if it contains an access to a location, there is a peer access to the identical location, and at least one of these accesses is a write. For an effective `atomic` expression, every peer access must either be a *predecessor* or a *successor*. A predecessor must occur before every constituent and every descendant in memory order. A successor must occur after every constituent and every ancestor in memory order. Every ancestor must occur before every descendant in memory order.

The above conditions guarantee that there is a single, global ordering for the effective `atomic` expressions in a Fortress program. This means that for any executions of `atomic` expressions $A$ and $B$ one of the following conditions holds:

- $A$ is dynamically contained inside $B$.
- $B$ is dynamically contained inside $A$.
- Every expression dynamically contained in $A$ precedes every expression dynamically contained in $B$ in memory order. This will always hold when $A$ is dynamically ordered before $B$.
- Every expression dynamically contained in $B$ precedes every expression dynamically contained in $A$ in memory order. This will always hold when $B$ is dynamically ordered before $A$.

The above rules are also sufficient to guarantee that `atomic` expressions nested inside an enclosing `atomic` behave with respect to one another just as if they had occurred at the top level in an un-nested context.

Any access preceding a `spawn` in dynamic program order will precede accesses in the spawned expression in memory order. Any access occurring after a spawned thread has been observed to complete in dynamic program order will occur after accesses in the spawned expression in memory order.

A reduction variable in a `for` loop does not have a single associated location; instead, there is a distinct location for each loop iteration, initialized by writing the identity of the reduction. These locations are distinct from the location associated with the reduction variable in the surrounding scope. In memory order there is a read of each of these locations each of which succeeds the last access to that variable in the loop iteration, along with a read of the location in the enclosing scope which succeeds all accesses to that location preceding the loop in dynamic program order. These reads are followed by a write of the location in the enclosing scope which in turn precedes all accesses to that location that succeed the loop in dynamic program order.

Finally, reads and writes in Fortress programs must respect dynamic program order for operations that are *semantically related*. If the read $A$ precedes the write $B$ in dynamic program order, and the value of $B$ can be determined in some fashion without recourse to $A$, then these operations are not semantically related. A simple example is if $A$ is a reference to variable $x$ and $B$ is the assignment $y := x \cdot 0$. Here it can be determined that $y := 0$ without recourse to $x$ and these variables are not semantically related. By contrast, the write $y := x$ is always semantically related to the read of $x$. Note that two operations can only be semantically related if a transitive data or control dependency exists between them.

# Chapter 20

# Components and APIs

Fortress programs are developed and compiled as *components*. The imported and exported references of a component are described with explicit *APIs*. [1]

## 20.1 Overview

Syntax:

| | | |
|---|---|---|
| *File* | ::= | *CompilationUnit* |
| | \| | *Imports*? *Exports* *Decls*? |
| | \| | *Imports*? *AbsDecls* |
| | \| | *Imports* *AbsDecls*? |
| *CompilationUnit* | ::= | *Component* |
| | \| | *Api* |

Components are the fundamental structure of Fortress programs. They export and import APIs, which serve as "interfaces" of the components. A key design choice we make is to require that components never refer to other components directly; all external references are to APIs. This requirement allows programmers to extend and test existing components more easily, swapping new implementations of libraries in and out of programs at will. External references are resolved by linking components together: the references of a component to an imported API are resolved to a component that exports that API. Linking components produces new components, whose *constituents* are the components that were linked together.

## 20.2 Components

Syntax:

| | | |
|---|---|---|
| *Component* | ::= | `native`? `component` *APIName* *Imports*? *Exports* *Decls*? `end` |
| *APIName* | ::= | *Id*( . *Id*)$^*$ |
| *Imports* | ::= | *Import*$^+$ |
| *Import* | ::= | `import` *ImportedNames* |
| *ImportedNames* | ::= | *APIName* . { ... } |
| | \| | *APIName* . { *SimpleNameList* ( , ... )? } |
| *SimpleNameList* | ::= | *SimpleName*( , *SimpleName*)$^*$ |

[1]The system described in this chapter is based on that described in [2].

```
SimpleName    ::=  Id
                |  opr Op
                |  opr EncloserPair
EncloserPair  ::=  (LeftEncloser | Encloser) ·? (RightEncloser | Encloser)
Exports       ::=  Export+
Export        ::=  export APINames
APINames      ::=  APIName
                |  { APINameList }
APINameList   ::=  APIName( , APIName)*
Decls         ::=  Decl+
```

In this specification, we will refer to components created by compiling a file as *simple components*, while components created by linking components together will be known as *compound components*.

The source code of a simple component definition begins with an optional modifier `native` followed by `component` followed by a *possibly qualified name* (an identifier or a sequence of identifiers separated by periods with no intervening whitespace), followed by a sequence of *import* statements, and a sequence of *export* statements, and finally a sequence of declarations, where all sequences are separated by newlines or semicolons.

### 20.2.1 Import Statements

There are two forms of import statements: *explicit import statements* and *on-demand import statements*.

An explicit import statement specifies a single API and a set of names; it imports the top-level and functional method declarations in the specified API. It allows the specified names to be used *unqualified* in the importing component or API:

`import` $\mathrm{APIName}.\{\,name(,name)^{+}\,\}$

An "ordinary" operator (i.e., operators other than enclosing or vertical-line operators) is a single token, and it is imported simply by putting `opr` before it in the import statement. This imports all of its prefix, postfix, infix, multifix and nofix declarations. A matched pair of enclosing operators is written `opr` the left encloser, the right encloser, with $\cdot$ optionally between the two enclosers. A vertical-line operator may be imported either as an infix operator or a bracketing operator, according to the rules above for ordinary and enclosing operators respectively.

For convenience, an on-demand import statement allows all names declared by imported declarations of the specified API to be referred to with unqualified names:

`import` $\mathrm{APIName}.\{\ldots\}$

A name $name$ is imported on demand in a component $C$ from an API $A$ only if all the following conditions hold:

1. $C$ contains an on-demand import statement.
2. Either 1) a top-level or functional method declaration of $name$ appears in $C$ or 2) a reference to $name$ appears in component $C$ and $C$ does not provide an explicit declaration of $name$.

The set of all declarations that are declared or imported (either explicitly or on demand) by a component must satisfy the overloading rules (and in particular, any nonfunctional declaration must not be overloaded).

If there is no imported declaration matching a reference, it is a static error. If there is more than one imported declaration that a reference may refer to, it is a static error. For example, it is a static error to have a reference $\mathrm{List}$ in the context of the following import statements:

`import` $\mathrm{List}.\{\ldots\}$
`import` $\mathrm{PureList}.\{\ldots\}$

A reference $\mathrm{List}$ may refer to the type $\mathrm{List}$ declared in the API $\mathrm{List}$ or the type $\mathrm{List}$ declared in the API $\mathrm{PureList}$.

### 20.2.2 Export Statements

Export statements specify the APIs that a component exports. One important restriction on components is that no API may be both imported and exported by the same component. This restriction helps to avoid some (but not all) accidental cyclic dependencies.

A component must provide a declaration, or a set of declarations, that *satisfies* every top-level declaration in any API that it exports, as described below. A component may include declarations that do not participate in satisfying any exported declaration (i.e., a declaration of any exported API).

A top-level variable declaration declaring a single variable is satisfied by any top-level variable declaration that declares the name with the same type (in the component, the type may be inferred). A top-level variable declaration declaring multiple variables is satisfied by a set of declarations (possibly just one) that declare all the names with their respective types (which again, may be inferred). In either case, the mutability of a variable must be the same in the exported and satisfying declarations.

A trait or object declaration is satisfied by a declaration that has the same header,[2] and contains, for each field declaration and non-abstract method declaration in the exported declaration, a satisfying declaration (or a set of declarations). When a trait has an abstract method declared, a satisfying trait declaration is allowed to provide a concrete declaration.

A satisfying trait or object declaration may contain method and field declarations not exported by the API but these might not be overloaded with method or field declarations provided by (contained in or inherited by) any declarations exported by the API.

For functional declarations, recall that several functional declarations may define the same entity (i.e., they may be overloaded). Given a set of overloaded declarations, it is not permitted to export some of them and not others.

### 20.2.3 Cross-Component Overloading

When a component imports a functional $f$ (either a top-level function or a functional method) by an import statement, the imported $f$ may be overloaded with a functional $f$ declared by the component. When a component imports a top-level declaration $f$ from an API $A$, all the relevant types to type check the uses of $f$ are implicitly imported from $A$. However, these implicitly imported types for type checking are not expressible by programmers; programmers must import the types by import statements to use them. For example, the two functional calls in the following component $C$ are valid:

```
api A
    trait T
        m() : ()
    end
end
api B
    import A.{...}
    f(): T
    g(T): ()
end
component C
    import B.{f, g}
    export Executable
    run(args) = do f().m(); g(f()) end
end
```

[2]The order of the modifiers, the clauses, and the types in the `extends`, `excludes` and `comprises` clauses may differ.

because $T$ is implicitly imported from $B$ to type check the functional calls. However, the programmers cannot write $T$ in $C$ because $T$ is not imported by an import statement.

### 20.2.4 Native Components

A component may have the modifier `native` to declare an "unsafe component". Within an unsafe component, the syntax and semantics are implementation dependent. However, an unsafe component can export an API, which can be imported by safe components and APIs as usual.

## 20.3 APIs

Syntax:

| | | |
|---|---|---|
| *Api* | ::= | `api` *APIName Imports*? *AbsDecls*? `end` |
| *AbsDecls* | ::= | *AbsDecl*$^+$ |
| *AbsDecl* | ::= | *AbsTraitDecl* |
| | \| | *AbsObjectDecl* |
| | \| | *AbsVarDecl* |
| | \| | *AbsFnDecl* |
| *AbsTraitDecl* | ::= | *AbsTraitMods*? *TraitHeaderFront AbsTraitClauses AbsGoInATrait*? `end` |
| *AbsTraitMods* | ::= | *AbsTraitMod*$^+$ |
| *AbsTraitMod* | ::= | `value` |
| *AbsTraitClauses* | ::= | *AbsTraitClause*$^*$ |
| *AbsTraitClause* | ::= | *Excludes* |
| | \| | *AbsComprises* |
| *AbsComprises* | ::= | `comprises` *ComprisingTypes* |
| *ComprisingTypes* | ::= | *TraitType* |
| | \| | { *ComprisingTypeList* } |
| *ComprisingTypeList* | ::= | ... |
| | \| | *TraitType*( , *TraitType*)* ( , ...)? |
| *AbsGoInATrait* | ::= | *AbsGoFrontInATrait AbsGoBackInATrait*? |
| | \| | *AbsGoBackInATrait* |
| *AbsGoFrontInATrait* | ::= | *AbsGoesFrontInATrait*$^+$ |
| *AbsGoesFrontInATrait* | ::= | *ApiFldDecl* |
| *AbsGoBackInATrait* | ::= | *AbsGoesBackInATrait*$^+$ |
| *AbsGoesBackInATrait* | ::= | *AbsMdDecl* |
| *AbsObjectDecl* | ::= | *AbsObjectMods*? *ObjectHeader AbsGoInAnObject*? `end` |
| *AbsObjectMods* | ::= | *AbsTraitMods* |
| *AbsGoInAnObject* | ::= | *AbsGoFrontInAnObject AbsGoBackInAnObject*? |
| | \| | *AbsGoBackInAnObject* |
| *AbsGoFrontInAnObject* | ::= | *AbsGoesFrontInAnObject*$^+$ |
| *AbsGoesFrontInAnObject* | ::= | *ApiFldDecl* |
| *AbsGoBackInAnObject* | ::= | *AbsGoesBackInAnObject*$^+$ |
| *AbsGoesBackInAnObject* | ::= | *AbsMdDecl* |
| *ApiFldDecl* | ::= | *BindId IsType* |
| *AbsVarDecl* | ::= | *AbsVarMods*? *VarWTypes* |
| | \| | *AbsVarMods*? *BindIdOrBindIdTuple* : *Type* ... |
| | \| | *AbsVarMods*? *BindIdOrBindIdTuple* : *TupleType* |
| *AbsVarMods* | ::= | *AbsVarMod*$^+$ |
| *AbsVarMod* | ::= | `var` |

APIs are compiled from special API definitions. These are source files which declare the entities declared by the API, the names of all APIs referred to by those declarations, and prose documentation. In short, the source code of an API must specify all the information that is traditionally provided for the published APIs of libraries in other languages.

The syntax of an API definition is identical to the syntax of a component definition, except that:

1. An API definition begins with `api` rather than `component`.
2. An API does not include `export` statements. (However, it does include `import` statements, which name the other APIs used in the API definition.)
3. Only declarations (not definitions!) are included in an API definition.

Import statements in APIs permit names declared in imported APIs to be used in the importing API just as in components. Those names are *not*, however, part of the importing API, and thus cannot be imported from that API by a component or another API.

For the sake of simplicity, every identifier reference in an API definition must refer either to a declaration in a used API (i.e., an API named in an import statement, or the Fortress core APIs, which are implicitly imported), or to a declaration in the API itself. In this way, APIs differ from signatures in most module systems: they are not parametric in their external dependencies.

## 20.4 Component and API Identity

Every component has a unique name, used for the purposes of component linking. This name includes a user-provided identifier. In the case of a simple component, the identifier is determined by a component name given at the top of the source file from which it is compiled. Component equivalence is determined nominally to allow mutually recursive linking of components.

Every API has a unique name that consists of a user-provided identifier. As with components, API equivalence is determined nominally. Component names must not conflict with API names.

# Part III

# Fortress for Library Writers

# Chapter 21

# Parallelism and Locality

Fortress is designed to make parallel programming as simple and as painless as possible. This chapter describes the internals of Fortress parallelism designed for use by authors of library code (such as generators and arrays). We adopt a multi-tiered approach to parallelism:

- At the highest level, we provide libraries that allocate shared arrays (Section 21.4) and implicitly parallel constructs such as tuples and loops. Synchronization is accomplished through the use of atomic sections (Section 13.22). More complex synchronization makes use of abortable atomicity, described in Section 21.5.
- Immediately below that, the `at` expression requests that a computation take place in a particular region of the machine (Section 21.2). We also provide a mechanism to terminate a spawned thread early (Section 21.6).
- Finally, there are mechanisms for constructing new generators via recursive subdivision into tree structures with individual elements at the leaves. Section 21.7 explains how iterative constructs such as `for` loops and comprehensions are desugared into calls to methods of trait $\mathrm{Generator}$, and how new instances of this trait may be defined.

We begin by describing the abstraction of *regions*, which Fortress uses to describe the machine on which a program is run.

## 21.1 Regions

Every thread (either explicit or implicit) and every object in Fortress, and every element of a Fortress array (the physical storage for that array element), has an associated *region*. The Fortress libraries provide a function $\mathit{region}$ which returns the region in which a given object resides. Regions abstractly describe the structure of the machine on which a Fortress program is running. They are organized hierarchically to form a tree, the *region hierarchy*, reflecting in an abstract way the degree of locality which those regions share. The distinguished region $\mathrm{Global}$ represents the root of the region hierarchy.[1] The different levels of this tree reflect underlying machine structure, such as execution engines within a CPU, memory shared by a group of processors, or resources distributed across the entire machine. The function $\mathit{here}()$ returns the region in which execution is currently occurring. Objects which reside in regions near the leaves of the tree are local entities; those which reside at higher levels of the region tree are logically spread out. The method call $r.\mathit{isLocalTo}(s)$ returns $\mathit{true}$ if $r$ is contained within the region tree rooted at $s$.

It is important to understand that regions and the structures built on top of them exist purely for performance purposes. The placement of a thread or an object does not have any semantic effect on the meaning of a program; it is simply an aid to enable the implementation to make informed decisions about data placement.

[1] Note: the initial implementation of the Fortress language assumes a single machine with shared memory and exposes only the $\mathrm{Global}$ region.

It might not be possible for an object or a thread to reside in a given region. Threads of execution reside at the *execution level* of the region hierarchy, generally the bottommost level in the region tree. Each thread is generally associated with some region at the execution level, indicating where it will preferentially be run. The programmer can affect the choice of region by using an `at` expression (Section 21.2) when the thread is created. A thread may have an associated region which is not at the execution level of the region hierarchy, either because a higher region was requested with an `at` expression or because scheduling decisions permit the thread to run in several possible execution regions. The region to which a thread is assigned may also change over time due to scheduling decisions. For the object associated with a spawned thread, the $\mathit{region}$ function provided by the Fortress libraries returns the region of the associated thread.

The *memory level* of the region hierarchy is where individual reference objects reside; on a machine with nodes composed of multiple processor cores sharing a single memory, this generally will not be a leaf of the region hierarchy. Imagine a constructor for a reference object is called by a thread residing in region $r$, yielding an object $o$. Except in very rare circumstances (for example when a local node is out of memory) either $r.\mathit{isLocalTo}(\mathit{region}(o))$ or $\mathit{region}(o).\mathit{isLocalTo}(r)$ ought to hold: data is allocated locally to the thread which runs the constructor. For a value object $v$ being manipulated by a thread residing in region $r$ either $\mathit{region}(v).\mathit{isLocalTo}(r)$ or $r.\mathit{isLocalTo}(\mathit{region}(v))$ (value objects always appear to be local).

Note that $\mathit{region}$ is a top-level function provided by the Fortress libraries and can be overridden by any functional method. The chief example of this is arrays, which are generally composed from many reference objects; the $\mathit{region}$ function can be overridden to return the location of the array as a whole—the region which contains all of its constituent reference objects.

## 21.2 Placing Threads

A thread can be placed in a particular region by using an `at` expression:

```
(v, w) := (a_i,
           at a.region(j) do
              a_j
           end)
```

In this example, two implicit threads are created; the first computes $a_i$ locally, the second computes $a_j$ in the region where the $j^{th}$ element of $a$ resides, specified by $a.\mathit{region}(j)$. The expression after `at` must return a value of type $\mathrm{Region}$, and the block immediately following `do` is run in that region; the result of the block is the result of the `at` expression as a whole. Often it is more graceful to use the `also do` construct (described inSection 13.11.3) in these cases:

```
do
   v := a_i
also at a.region(j) do
   w := a_j
end
```

We can also use `at` with a `spawn` expression:

```
s = spawn at a.region(i) do
           a_i
    end
t = spawn at region(s) do
           a_j
    end
```

Finally, note that it is possible to use an `at` expression within a block:

```
do
   $v := a_i$
   at $a.region(j)$ do
      $w := a_j$
   end
   $x = v + w$
end
```

We can think of this as the degenerate case of `also do`: a thread group is created with a single implicit thread running the contents of the `at` expression in the given region; when this thread completes control returns to the original location.

Note that the regions given in an `at` expression are non-binding: the Fortress implementation may choose to run the computations elsewhere—for example, thread migration might not be possible within an `atomic` expression, or load balancing might cause code to be executed in a different region. In general, however, implementations should attempt to respect thread placement annotations when they are given.

## 21.3 Shared and Local Data

Every object in a Fortress program is considered to be either *shared* or *local* (collectively referred to as the *sharedness* of the object). A local object must be transitively reachable (through zero or more object references) from the variables of at most one running thread. A local object may be accessed more cheaply than a shared object, particularly in the case of atomic reads and writes. Sharedness is ordinarily managed implicitly by the Fortress implementation. Control over sharedness is intended to be a performance optimization; however, functions such as $isShared$ and $localize$ provided by the Fortress libraries can affect program semantics, and must be used with care.

The sharedness of an object must be contrasted with its region. The region of an object describes where that object is located on the machine. The sharedness of an object describes whether the object is visible to one thread or to many. A local object need not actually reside in a region near the thread to which it is visible (though ordinarily it will).

The following rules govern sharedness:

- Reference objects are initially local when they are constructed.
- The sharedness of an object may change as the program executes.[2]
- If an object is currently transitively reachable from more than one running thread, it must be shared.
- When a reference to a local object is stored into a field of a shared object, the local object must be *published*: Its sharedness is changed to shared, and all of the data to which it refers is also published.
- The value of a local variable referenced by a thread must be published before that thread may be run in parallel with the thread which created it. Values assigned to the variable while the threads run in parallel must also be published.
- A field with value type is assigned by copying, and thus has the sharedness of the containing object or closure.

Publishing can be expensive, particularly if the structure being broadcast is large and heavily nested; this can cause an apparently short `atomic` expression (a single write, say) to run arbitrarily long. To avoid this, the library programmer can request that an object be published by calling the semantically transparent function $shared$ provided by the Fortress libraries:

[2]Note, for example, that the present Fortress implementation immediately makes every object shared after construction, so that $isShared()$ will always return $true$.

|  |  |
|---|---|
| $array[\![E]\!](size:I):\mathrm{Array}[\![E,I]\!]$ | Creates an uninitialized 0-indexed array of the specified runtime-determined size. The size is an integer for a 1-dimensional array, a 2-tuple for a two-dimensional array, and so forth. |
| $array_1[\![E,n]\!]()$ | Creates an uninitialized 0-indexed 1-dimensional array of statically determined size $n$. |
| $array_2[\![E,n,m]\!]()$ | Creates an uninitialized 0-indexed 2-dimensional array of statically determined size $n$ by $m$. |
| $array_3[\![E,n,m,p]\!]()$ | Creates an uninitialized 0-indexed 3-dimensional array of statically determined size $n$ by $m$ by $p$. |
| $a.\mathit{fill}(v:E)$ | Initializes all elements with value $v$. |
| $a.\mathit{fill}(f:I\rightarrow E)$ | Calls $f$ at each index and initializes the corresponding element with the result of the call. |
| $a.\mathit{init}(i:I,v:E)$ | Initializes element at index $i$ with value $v$. |

Figure 21.1: Factories for creating arrays and methods for initializing their elements

$y := \mathit{shared}\ \mathrm{Cons}(x, xs)$
$\mathit{shared}(y)$

Two additional functions are provided which permit different choices of program behavior based on the sharedness of objects:

- The function $\mathit{isShared}(o)$ returns $\mathit{true}$ when $o$ is shared, and $\mathit{false}$ when it is local. This permits the program to take different actions based on sharedness.
- The function $\mathit{localize}(o)$ is provided that attempts to make a local version of object $o$, by copying if necessary.

These functions must be used with extreme caution. For example, $\mathit{localize}$ should be used only when there is a unique reference to the object being localized. The $\mathit{localize}$ function can have unexpected behavior if there is a reference to $o$ from another local object $p$. Updates to $o$ will be visible through $p$; subsequent publication of $p$ will publish $o$. By contrast, if $o$ was already shared, and referred to by another shared object, the newly-localized copy will be entirely distinct; changes to the copy will not be visible through $p$, and publishing $p$ will not affect the locality of the copy.

## 21.4 Distributed Arrays

Arrays, vectors, and matrices in Fortress are assumed to be spread out across the machine. As in Fortran, Fortress arrays are complex data structures. The default distribution of an array is determined by the Fortress libraries; in general it depends on the size of the array, and on the size and locality characteristics of the machine running the program. Programmers must create arrays by using an aggregate expression (Section 13.27), or by using one of the factory functions at the top of Figure 21.1. After calling any of these factories, the elements of the resulting array must be initialized using either indexed assignment or using one of the initialization methods described at the bottom of the figure.

Each array element may be initialized at most once using the methods of Figure 21.1; the programmer must assure that this initialization completes before any other access to the corresponding array element. The programmer must also assure that an element is initialized before it is first read.[3] Note that these factories and methods are intended to be used to library programmers to build higher-level functionality (for example, the $\mathit{fill}$ methods themselves are defined in terms of calls to $\mathit{init}$, and the $array$ factory is defined in terms of $array_1$, $array_2$, and so forth).

Because the elements of a fortress array may reside in multiple regions of the machine, there is an additional method $a.\mathit{region}(i)$ which returns the region in which the array element $a_i$ resides. An element of an array is always local

[3]The present implementation signals a fatal error in case of duplicate initialization, or if an uninitialized element is read.

to the region in which the array as a whole is contained, so $(a.\mathit{region}(i)).\mathit{isLocalTo}(\mathit{region}(a))$ must always return $\mathit{true}$. When an array contains reference objects, the programmer must be careful to distinguish the region in which the array element $a_i$ resides, $a.\mathit{region}(i)$, from the region in which the object referred to by the array element resides, $\mathit{region}(a_i)$. The former describes the region of the array itself; the latter describes the region of the data referred to by the array. These may differ.

## 21.5 Abortable Atomicity

Fortress provides a user-level $\mathit{abort}()$ function which abandons execution of the innermost `atomic` expression and rolls back its changes, requiring the `atomic` expression to execute again from the beginning. This permits an atomic section to perform consistency checks as it runs. Invoking the $\mathit{abort}$ function not within an `atomic` expression has no effect. The functionality provided by $\mathit{abort}()$ can be abused; it is possible to induce deadlock or livelock by creating an atomic section that always fails. Here is a simple example of a program using $\mathit{abort}()$ which is incorrect because Fortress does not guarantee that the two implicit threads (created by evaluating the two elements of the tuple) will always run in parallel; it is possible for the first element of the tuple to continually abort without ever running the second element of the tuple:

```
r: ℤ64 := 0
(a, b) = (atomic if r = 1 then 17 else abort() end,
          do r := 1; r end)(* INCORRECT! *)
```

Fortress also includes a `tryatomic` expression, which attempts to run its body expression atomically. If it succeeds, the result is returned; if it aborts due to a call to $\mathit{abort}$ or due to conflict (as described in Section 13.22), the checked exception TryAtomicFailure is thrown. In addition, `tryatomic` is permitted to throw TryAtomicFailure in the absence of conflict; however, it is not permitted to fail unless some other thread performs an access to shared state while the body is being run. Conceptually `atomic` can be defined in terms of `tryatomic` as follows:

```
label AtomicBlock
  while true do
    try
      result = tryatomic body
      exit AtomicBlock with result
    catch e
      TryAtomicFailure ⇒ ()(* continue execution *)
    end
  end
  throw UnreachableCode(* inserted for type correctness *)
end AtomicBlock
```

Unlike the above definition, an implementation may choose to suspend a thread running an `atomic` expression which invokes $\mathit{abort}$, re-starting it at a later time when it may be possible to make further progress. The above definition restarts the body of the `atomic` expression immediately without suspending.

## 21.6 Early Termination of Threads

As noted in Section 5.4, an implicit thread can be terminated if its group is going to throw an exception. Similarly, a spawned thread $t$ may be terminated by calling $t.\mathit{stop}()$. A successful attempt to terminate a thread causes the thread to complete asynchronously. There is no guarantee that termination attempts will be prompt, or that they will occur at all; the implementation will make its best effort. If a thread completes normally or exceptionally before an attempt to terminate it succeeds, the result is retained and the termination attempt is simply dropped.

At present stopping a thread immediately causes it to cease execution; no outstanding `finally` blocks are run and the thread is not considered to return a result.

## 21.7 Use and Definition of Generators

Several expressions in Fortress make use of *generator clause lists* to express parallel iteration (see Section 13.14). A generator clause list binds a series of variables to the values produced by a series of objects that extend the trait $\mathrm{Generator}$. A generator clause list is simply syntactic sugar for a nested series of invocations of methods on these objects.

A type that extends $\mathrm{Generator}[\![E]\!]$ acts as a generator of elements of type $E$. An instance of $\mathrm{Generator}[\![E]\!]$ only needs to define the $\mathit{generate}$ method:

$$\mathit{generate}[\![R]\!](r : \mathrm{Reduction}[\![R]\!], \mathit{body}\colon E \to R)\colon R$$

The mechanics of object generation are embedded entirely in the $\mathit{generate}$ method. This method takes two arguments. The $\mathit{generate}$ method invokes $\mathit{body}$ once for each object which is to be generated, passing the generated object as an argument. Each call to $\mathit{body}$ returns a result of type $R$; these results are combined using the reduction $r$, which encapsulates a monoidal operator on objects of type $R$. *All the parallelism provided by a particular generator is specified by definition of the* $\mathit{generate}$ *method.*

In practice, calls to $\mathit{generate}$ are produced by desugaring expressions with generator clause lists, as described below (Section 21.7.1). However it is possible to call the $\mathit{generate}$ method directly, as in the following example:

```
object SumZZ32 extends Reduction[[ℤ32]]
    empty(): ℤ32 = 0
    join(a: ℤ32, b: ℤ32): ℤ32 = a + b
end
z = (1 # 100).generate[[ℤ32]](SumZZ32, fn (x) ⇒ 3x + 2)
```

Any reduction must define two methods: an associative binary operation $\mathit{join}$, and $\mathit{empty}$, a method that returns the identity of this operation. Here we define reduction $\mathrm{SumZZ32}$ representing integer addition. We use this to compute the sum of $3x + 2$ for $x$ drawn from the range $1 \# 100$, yielding the expected answer of 15350.

For non-commutative reductions such as the $\mathrm{Concat}$ reduction used for list comprehensions in Figure 21.4, it is important to note that results must be combined in the natural order of the generator. If $\mathit{join}$ is not associative, or $\mathit{empty}$ is not the identity of $\mathit{join}$, passing the reduction to $\mathit{generate}$ will produce unpredictable results. A generator is permitted to group reduction operations in any way it likes consistent with its natural order, and insert an arbitrary number of $\mathit{empty}$ elements.

Figure 21.2 defines a generator that generates the integers between $\mathit{lo}$ and $\mathit{hi}$ in sequential blocks of size at most $b$. In this example, we divide the range in half if it is larger than the block size $b$; these two halves are computed in parallel (recall that the arguments to the method call $\mathit{reduction.join}$ are evaluated in parallel). If the range is smaller than $b$, then it is enumerated serially using a `while` loop, accumulating the result $r$ as it goes. Observe that the parallelism obtained from a generator is *dictated by the code of the* $\mathit{generate}$ *method.* While programmers using a generator should assume that calls to $\mathit{body}$ may occur in parallel, the library programmer is free to choose the amount of parallelism that is actually exposed.

This example uses *recursive subdivision* to divide a blocked range into approximately equal-sized chunks of work. Recursive subdivision is the recommended technique for exposing large amounts of parallelism in a Fortress program because it adjusts easily to varying machine sizes and can be dynamically load balanced.

$\mathrm{Generator}$ defines the functional method $\mathit{seq}(\mathtt{self})$ that returns an equivalent generator that runs iterations of the body sequentially in natural order. In most cases (such as in this example), it is prudent to override the default definition

```
object BlockedRange($lo$: $\mathbb{Z}64$, $hi$: $\mathbb{Z}64$, $b$: $\mathbb{Z}64$) extends Generator$[\![\mathbb{Z}64]\!]$
    $size : \mathbb{Z}64 = hi - lo + 1$
    $seq$(self): SequentialGenerator$[\![\mathbb{Z}64]\!] = seq(lo : hi)$
    $generate[\![R]\!](reduction$: Reduction$[\![R]\!]$, $body$: $\mathbb{Z}64 \to R$): $R$ =
        if $size \leq (b$ MAX $1)$ then
            ($*$ Blocks smaller than b run sequentially $*$)
            $r : R := reduction.empty()$
            $i : \mathbb{Z}64 := lo$
            while $i \leq hi$ do
                $v : R = body(i)$
                $r := reduction.join(r, v)$
                $i$ += 1
            end
            $r$
        else
            ($*$ Blocks larger than b are split in half and generated in parallel. $*$)
            $mid = \lfloor (lo + hi)/2 \rfloor$
            $reduction.join$(BlockedRange$(lo, mid, b).generate[\![R]\!](reduction, body)$,
                            BlockedRange$(mid + 1, hi, b).generate[\![R]\!](reduction, body)$)
        end
end
$f() = \langle\, 2x \mid x \leftarrow$ BlockedRange$(1, 10, 3)\,\rangle$
```

Figure 21.2: Sample Generator definition: blocked integer ranges.

$$
\begin{array}{lll}
\mathcal{C}[\ ] \quad body & = & u(body) \\
\mathcal{C}[x \leftarrow g, gs]\ body & = & g.generate(r, \mathtt{fn}(x) \Rightarrow \mathcal{C}[gs]\ body) \\
\mathcal{C}[p, gs]\ body & = & p.generate(r, \mathtt{fn}() \Rightarrow \mathcal{C}[gs]\ body)
\end{array}
$$

Figure 21.3: Simple syntax-directed desugaring of a generator clause list. Here the reduction $r$ and unit $u$ are variables chosen by the desugarer to be distinct from the variables in *gs* and *body*.

of this method; the default implementation of $seq$ effectively collects the generated elements together in parallel and traverses the result sequentially.

The remainder of this section describes in detail the desugaring of expressions with generator clause lists into invocations of the $generate$ methods of the generators in the generator clause list.

### 21.7.1 Simple Desugaring of Expressions with Generators

An expression with a generator clause list *gs* and body expression *body* is desugared into the following general form:

$$wrapper(\mathtt{fn}\ (r, u) \Rightarrow \mathcal{C}[gs]\ body)$$

The generator clause list and body can be desugared using the syntax-directed desugaring $\mathcal{C}$ defined in Figure 21.3. This yields a function that is in turn passed as an argument to $wrapper$. The particular choice of the function $wrapper$ depends upon the construct that is being desugared. For a reduction or a comprehension, the wrapper function is the corresponding big operator definition; see Section 13.17 and Section 13.28. For a `for` loop (Section 13.15) or a generated expression (Section 13.11.2), a special built-in wrapper is used. Examples are shown in Figure 21.4. A wrapper function always has the following type:

$$wrapper(g : (\text{Reduction}[\![R_0]\!], T \to R_0) \to R_0) : R$$

| expr | type | *wrapper* | $u(body)$ | $r$ |
|---|---|---|---|---|
| $\sum_{gs} e$ | $N$ | `SUM`$[\![N]\!]$ | $identity[\![N]\!](e)$ | $\mathrm{SumReduction}[\![N]\!]$ |
| $\langle\, e \mid gs \,\rangle$ | List$[\![E]\!]$ | `BIG`$\langle[\![E]\!]\,\rangle$ | $singleton[\![E]\!](e)$ | $\mathrm{Concat}[\![E]\!]$ |
| $lv := e, gs$ | () | built in | $ignore(lv := e)$ | $\mathrm{NoReduction}$ |

Figure 21.4: Examples of wrappers for expressions with generator clause lists. Top to bottom: big operators (here $\sum$ is used as an example; the appropriate library function is called on the right-hand side), comprehensions (here list comprehensions are shown; other comprehensions are similar to list comprehensions) and generated assignment.

Here the type $T$ is the type of values returned by the body expression, and $R_0$ and $R$ are arbitrary types chosen by the wrapper function.

Note that the function $g$ passed to the wrapper has essentially the same type signature as the $generate$ method itself. It is instructive to think of $wrapper$ as having the following similar type signature:[4]

$wrapper'(g\colon \mathrm{Generator}[\![T]\!])\colon R(*$ NOT THE ACTUAL TYPE $*)$

[4]In future, it is likely that Fortress will use a desugaring that in fact yields a $\mathrm{Generator}$ rather than a higher-order function. This permits type-directed nesting and composition of generators.

# Chapter 22

# Operator Declarations

An operator declaration may appear anywhere a top-level function or method declaration may appear. Operator declarations are like other function or method declarations in all respects except that an operator declaration has `opr` and has an operator name (see Section 16.1 for a discussion of valid operator names) instead of an identifier. The precise placement of the operator name within the declaration depends on the fixity of the operator. Like other functionals, operators may have overloaded declarations (see Chapter 15 for a discussion of overloading). These overloadings may be of the same or differing fixities.

Syntax:

| | | |
|---|---|---|
| *FnDecl* | ::= | *FnMods*? *FnHeaderFront* *FnHeaderClause* ( = *Expr*)? |
| *FnHeaderFront* | ::= | *OpHeaderFront* |
| *OpHeaderFront* | ::= | `opr` `BIG` ? ({ $\mapsto$ \| *LeftEncloser* \| *Encloser*) *StaticParams*? *Params* |
| | | (*RightEncloser* \| *Encloser*) |
| | \| | `opr` *ValParam* (*Op* \| *ExponentOp*) *StaticParams*? |
| | \| | `opr` `BIG` ? (*Op* \| ˆ \| *Encloser* \| $\sum$ \| $\prod$) *StaticParams*? *ValParam* |
| *MdDecl* | ::= | *MdDef* |
| | \| | *MdMods*? *MdHeaderFront* *FnHeaderClause* |
| *MdHeaderFront* | ::= | *OpMdHeaderFront* |
| *OpMdHeaderFront* | ::= | `opr` `BIG` ? ({ $\mapsto$ \| *LeftEncloser* \| *Encloser*) *StaticParams*? *Params* |
| | | (*RightEncloser* \| *Encloser*) |
| | | ( := ( *SubscriptAssignParam* ) )? |
| | \| | `opr` *ValParam* (*Op* \| *ExponentOp*) *StaticParams*? |
| | \| | `opr` `BIG` ? (*Op* \| ˆ \| *Encloser* \| $\sum$ \| $\prod$) *StaticParams*? *ValParam* |
| *SubscriptAssignParam* | ::= | *Varargs* |
| | \| | *Param* |

An operator declaration has one of seven forms: infix operator declaration, prefix operator declaration, postfix operator declaration, nofix operator declaration, bracketing operator declaration, subscripting operator method declaration, and subscripted assignment operator method declaration. Each is invoked according to specific rules of syntax. An operator method declaration must be a functional method declaration, a subscripting operator method declaration, or a subscripted assignment operator method declaration.

## 22.1 Infix Operator Declarations

An infix operator declaration has `opr` and then an operator name where a functional declaration would have an identifier. The declaration must have two value parameter, which must not be a keyword parameter or varargs parameter. Static parameters may also be present, between the operator and the parameter list.

An expression consisting of an infix operator applied to an expression will invoke an infix operator declaration. The compiler considers all infix operator declarations for that operator that are both accessible and applicable, and the most specific operator declaration is chosen according to the usual rules for overloaded functionals.

Note that superscripting (ˆ) may be defined using an infix operator declaration even though it has very high precedence.

Example:

$\texttt{opr}\ \texttt{MAX}[\![T\ \texttt{extends}\ \text{String}]\!](x\colon T, y\colon T) : T = \texttt{if}\ \ x > y\ \texttt{then}\ \ x\ \texttt{else}\ \ y\ \texttt{end}$

## 22.2 Prefix Operator Declarations

A prefix operator declaration has `opr` and then an operator name where a functional declaration would have an identifier. The declaration must have one value parameter, which must not be a keyword parameter or varargs parameter. Static parameters may also be present, between the operator and the parameter list.

An expression consisting of a prefix operator applied to an expression will invoke a prefix operator declaration. The compiler considers all prefix operator declarations for that operator that are both accessible and applicable, and the most specific operator declaration is chosen according to the usual rules for overloaded functionals.

Example:

$\texttt{opr}\ \texttt{INV}(x\colon \text{Widget})\colon \text{Widget} = x.\mathit{invert}()$

## 22.3 Postfix Operator Declarations

A postfix operator declaration has `opr` where a functional declaration would have an identifier; the operator name itself *follows* the parameter list. The declaration must have one value parameter, which must not be a keyword parameter or varargs parameter. Static parameters may also be present, between `opr` and the parameter list.

An expression consisting of a postfix operator applied to an expression will invoke a postfix operator declaration. The compiler considers all postfix operator declarations for that operator that are both accessible and applicable, and the most specific operator declaration is chosen according to the usual rules for overloaded functionals.

Example:

$\texttt{opr}\ (n\colon \mathbb{Z}32)! = \prod_{i \leftarrow 1:n} i (*\ \text{factorial}\ *)$

## 22.4 Nofix Operator Declarations

A nofix operator declaration has `opr` and then an operator name where a functional declaration would have an identifier. The declaration must have no parameters.

An expression consisting only of a nofix operator will invoke a nofix operator declaration. The compiler considers all nofix operator declarations for that operator that are both accessible and applicable, and the most specific operator declaration is chosen according to the usual rules for overloaded functionals.

Uses for nofix operators are rare, but those rare examples are very useful. For example, if the @ operator is used to construct subscripting ranges, and it is the nofix declaration of @ that allows a lone @ to be used as a subscript:

$\texttt{opr}\ @() \colon \mathrm{ImplicitRange} = \mathrm{ImplicitRange}$

## 22.5 Bracketing Operator Declarations

A bracketing operator declaration has `opr` where a functional declaration would have an identifier. The value parameter list, rather than being surrounded by parentheses, is surrounded by the brackets being defined. A bracketing operator declaration may have any number of parameters, keyword parameters, and varargs parameters in the value parameter list. Static parameters may also be present, between `opr` and the parameter list. Any paired Unicode brackets may be so defined *except* ordinary parentheses and white square brackets.

An expression consisting of zero or more comma-separated expressions surrounded by a bracket pair will invoke a bracketing operator declaration. The compiler considers all bracketing operator declarations for that type of bracket pair that are both accessible and applicable, and the most specific operator declaration is chosen according to the usual rules for overloaded functionals. For example, the expression $\langle p, q\rangle$ might invoke the following bracketing method declaration:

$(*\ \text{angle bracket notation for inner product}\ *)$
$\texttt{opr}\ \langle [\![T\ \texttt{extends}\ \mathrm{Number}, \texttt{nat}\ k]\!] x \colon \mathrm{Vector}[\![T,k]\!], y \colon \mathrm{Vector}[\![T,k]\!]\rangle =$
$\qquad \sum_{i \leftarrow x.indices()} x_i \cdot y_i$

## 22.6 Subscripting Operator Method Declarations

A subscripting operator method declaration has `opr` where a method declaration would have an identifier. The value parameter list, rather than being surrounded by parentheses, is surrounded by a pair of brackets. A subscripting operator method declaration may have any number of value parameters and varargs parameters in that value parameter list. Static parameters may also be present, between `opr` and the left bracket. Any paired Unicode brackets may be so defined *except* ordinary parentheses and white square brackets; in particular, the square brackets ordinarily used for indexing may be used.

An expression consisting of a subexpression immediately followed (with no intervening whitespace) by zero or more comma-separated expressions surrounded by brackets will invoke a subscripting operator method declaration. Methods for the expression preceding the bracketed expression list are considered. The compiler considers all subscripting operator method declarations that are both accessible and applicable, and the most specific method declaration is chosen according to the usual overloading rules. For example, the expression $\mathit{foo}_p$ might invoke the following subscripting method declaration because expressions in the square brackets are rendered as subscripts:

$(*\ \text{subscripting method}\ *)$
$\texttt{opr}\ [x \colon \mathrm{BizarroIndex}] = \texttt{self}.\mathit{bizarroFetch}(x)$

## 22.7 Subscripted Assignment Operator Method Declarations

A subscripted assignment operator method declaration has `opr` where a method declaration would have an identifier. The value parameter list, rather than being surrounded by parentheses, is surrounded by a pair of brackets; this is then followed by the operator `:=` and then a second value parameter list in parentheses, which must contain exactly one value parameter. A subscripted assignment operator method declaration may have any number of value parameters within the brackets; these value parameters may include varargs parameters. A result type may appear after the second value parameter list, but it must be $()$. Static parameters may also be present, between `opr` and the left bracket. Any paired Unicode brackets may be so defined *except* ordinary parentheses and white square brackets; in particular, the square brackets ordinarily used for indexing may be used.

An assignment expression consisting of an expression immediately followed (with no intervening whitespace) by zero or more comma-separated expressions surrounded by brackets, followed by the assignment operator `:=`, followed by another expression, will invoke a subscripted assignment operator method declaration. Methods for the expression preceding the bracketed expression list are considered. The compiler considers all subscript operator method declarations that are both accessible and applicable, and the most specific method declaration is chosen according to the usual overloading rules. When a compound assignment operator (described in Section 13.7) is used with a subscripting operator and a subscripted assignment operator, for example $a_3 \mathrel{+}= k$, both a subscripting operator declaration and a subscripted assignment operator declaration are required. For example, the assignment $foo_p := myWidget$ might invoke the following subscripted assignment method declaration:

$(*$ subscripted assignment method $*)$
$\texttt{opr}\ [x\colon \text{BizarroIndex}] := (newValue\colon \text{Widget}) = \texttt{self}.bizarroInstall(x, newValue)$

## 22.8 Conditional Operator Declarations

A *conditional operator* is a binary operator (other than '`:`') that is immediately followed by '`:`'; see Section 16.6. A conditional operator expression $x@\!:y$ is syntactic sugar for $x@(\texttt{fn}\ () \Rightarrow y)$; that is, the right-hand operand is converted to a "thunk" (zero-parameter function) that then becomes the right-hand operand of the corresponding unconditional operator. Therefore a conditional operator is simply implemented as an overloading of the operator that accepts a thunk.

It is also permitted for a conditional operator to have a preceding as well as a following colon. A conditional operator expression $x:@:y$ is syntactic sugar for $(\texttt{fn}\ () \Rightarrow x)@(\texttt{fn}\ () \Rightarrow y)$; that is, each operand is converted to a thunk. This mechanism is used, for example, to define the results-comparison operator `:∼:`, which takes exceptions into account.

The conditional $\wedge$ and $\vee$ operators for boolean values, for example, are implemented as follows:

$\texttt{opr}\ \vee(a:\text{Boolean}, b:\text{Boolean}):\text{Boolean} = \texttt{if}\ a\ \texttt{then}\ true\ \texttt{else}\ b\ \texttt{end}$
$\texttt{opr}\ \wedge(a:\text{Boolean}, b:\text{Boolean}):\text{Boolean} = \texttt{if}\ a\ \texttt{then}\ b\ \texttt{else}\ false\ \texttt{end}$
$\texttt{opr}\ \vee(a:\text{Boolean}, b:() \rightarrow \text{Boolean}):\text{Boolean} = \texttt{if}\ a\ \texttt{then}\ true\ \texttt{else}\ b()\ \texttt{end}$
$\texttt{opr}\ \wedge(a:\text{Boolean}, b:() \rightarrow \text{Boolean}):\text{Boolean} = \texttt{if}\ a\ \texttt{then}\ b()\ \texttt{else}\ false\ \texttt{end}$

## 22.9 Big Operator Declarations

A *big operator* such as $\sum$ or $\prod$ is declared as a usual operator declaration. See Section 21.7 for an example declaration of a big operator. A big operator application is either a *reduction expression* described in Section 13.17 or a *comprehension* described in Section 13.28.

## Part IV

# Fortress Library APIs and Documentation

# Chapter 23

# Structure of the Fortress Libraries

The Fortress libraries are divided into two basic categories. The first, covered in Chapter 24, are the libraries that are automatically imported by every Fortress component and API, chiefly $\mathrm{FortressLibrary}$ and $\mathrm{FortressBuiltin}$. These libraries provide the basic numeric types, generators, arrays, booleans, exceptions, and the like. The second, covered in Chapter 25, are libraries that must be explicitly imported by the programmer. These include $\mathrm{List}$, $\mathrm{Set}$, and $\mathrm{Map}$.

The documentation in Part IV is largely automatically generated from the API code for the libraries themselves, and describes the state of the libraries as of the release date of this specification. The libraries are presently in a state of flux, and programmers using a more recent version of the Fortress implementation may find differences between the APIs presented here and those found in the actual implementation.

# Chapter 24

# Default Libraries

Two sets of libraries are imported into every Fortress program by default. The first, $\mathrm{FortressLibrary}$ (Section 24.1), implements the high-level functionality that will be used by most Fortress programmers. It will eventually be possible to selectively exclude portions of this library from a component if desired. By contrast, the second set of libraries are the *builtins* (Section 24.2). These libraries are intended to encompass primitive functionality that must be visible in every Fortress component.

## 24.1 FortressLibrary

`api` $\mathrm{FortressLibrary}$

The $\mathit{builtinPrimitive}$ function is actually recognized as a special piece of built-in magic by the Fortress interpreter. The $\mathit{javaClass}$ argument names a Java Class which is a subclass of `com.sun.fortress.interpreter.glue.NativeApp`, which provides code for the closure which is used in place of the call to $\mathit{builtinPrimitive}$. Meanwhile all the necessary type information, argument names, etc. must be declared here in Fortress-land. For examples, see the end of this file.

In practice, if you are extending the interpreter you will probably want to extend `com.sun.fortress.interpreter.glue.NativeFn0,1,2,3,4` or one of their subclasses defined in `com.sun.fortress.interpreter.glue.primitive`. These types are generally easier to work with, and the boilerplate packing and unpacking of values is done for you.

$\mathit{builtinPrimitive}[\![T]\!](\mathit{javaClass}:\mathrm{String}):T$

### Simple Combinators

Casting

$\mathit{cast}[\![T\ \texttt{extends}\ \mathrm{Any}]\!](x:\mathrm{Any}):T$

$\mathit{instanceOf}[\![T\ \texttt{extends}\ \mathrm{Any}]\!](x:\mathrm{Any}):\mathrm{Boolean}$

Useful functions

$ignore(\_ : \text{Any}) : ()$

$identity[\![T$ `extends` $\text{Any}]\!](x : T) : T$

$(*$ Should we depracate tuple and use identity instead? Decision: no. $*)$
$tuple[\![T]\!](x : T) : T$

$(*$ Function composition $*)$
`opr` $\circ[\![A, B, C]\!](f\colon B \to C, g\colon A \to B)\colon A \to C$

$fail(s : \text{String})$

## Control over locality and location

At the moment, all Fortress objects are immediately shared by default.
$shared[\![T$ `extends` $\text{Any}]\!](x : T)\colon T$

$isShared(x : \text{Any})\colon \text{Boolean}$

$localize[\![T$ `extends` $\text{Any}]\!](x : T)\colon T$

$(*copy\ is\ presently\ unimplemented.$
$copy[\![T$ `extends` $\text{Any}]\!](x : T)\colon T$
$*)$

```
trait Region extends Equality⟦Region⟧
      getter toString(): String
      isLocalTo(r: Region): Boolean
end
```

```
object Global extends Region end
```

$region(a : \text{Any})\colon \text{Region}$

$here()\colon \text{Region}$

## Equality and ordering

`opr` $=(a : \text{Any}, b : \text{Any}) : \text{Boolean}$

`opr` $\neq(a : \text{Any}, b : \text{Any}) : \text{Boolean}$

```
trait Equality⟦Self extends Equality⟦Self⟧⟧
            excludes { Number }(* Until Number is an actual type. *)
      abstract opr =(self, other : Self): Boolean
end
```

Total ordering

```
object LexicographicPartialReduction extends Reduction⟦Comparison⟧
      empty(): Comparison
      join(a : Comparison, b : Comparison) : Comparison
end
```

```
object LexicographicReduction extends Reduction⟦TotalComparison⟧
      empty(): TotalComparison
      join(a : TotalComparison, b : TotalComparison) : TotalComparison
end
```

```
trait Comparison
```

```
        extends { StandardPartialOrder⟦Comparison⟧ }
        comprises { Unordered, TotalComparison }
    abstract getter toString(): String
    opr =(self, other : Comparison): Boolean
    opr LEXICO(self, other : Comparison): Comparison
    abstract opr INVERSE(self): Comparison
end
```

Unordered is the outcome of $a$ `CMP` $b$ when $a$ and $b$ are partially ordered and no ordering relationship exists between them.

```
object Unordered extends Comparison
    getter toString(): String
    opr =(self, other : Unordered): Boolean
    opr <(self, other : Comparison): Boolean
    opr INVERSE(self): Comparison
end
```

TotalComparison is both a partial order (including Unordered) and a total order (TotalComparison alone). Its method definitions avoid ambiguities between these orderings.

```
trait TotalComparison
        extends { Comparison, StandardTotalOrder⟦TotalComparison⟧ }
        comprises { LessThan, EqualTo, GreaterThan }
    opr =(self, other : Comparison): Boolean
    opr CMP(self, other : Unordered): Boolean
    opr ≥(self, other : Unordered): Boolean
    opr ≥(self, other : Comparison): Boolean
    opr LEXICO(self, other : TotalComparison): TotalComparison
    opr LEXICO(self, other : () → TotalComparison): TotalComparison
    abstract opr INVERSE(self): TotalComparison
end
object LessThan extends TotalComparison
    getter toString(): String
    opr =(self, other : LessThan): Boolean
    opr CMP(self, other : LessThan): Comparison
    opr CMP(self, other : TotalComparison): TotalComparison
    opr <(self, other : LessThan): Boolean
    opr <(self, other : TotalComparison): Boolean
    opr INVERSE(self): TotalComparison
end
object GreaterThan extends TotalComparison
    getter toString(): String
    opr =(self, other : GreaterThan): Boolean
    opr CMP(self, other : GreaterThan): Comparison
    opr CMP(self, other : TotalComparison): TotalComparison
    opr <(self, other : TotalComparison): Boolean
    opr INVERSE(self): TotalComparison
end
object EqualTo extends TotalComparison
    getter toString(): String
    opr =(self, other : EqualTo): Boolean
    opr CMP(self, other : TotalComparison): TotalComparison
    opr <(self, other : LessThan): Boolean
    opr <(self, other : TotalComparison): Boolean
    opr LEXICO(self, other : TotalComparison): TotalComparison
    opr LEXICO(self, other : () → TotalComparison): TotalComparison
    opr INVERSE(self): TotalComparison
end
```

StandardPartialOrder is partial ordering using $<$, $>$, $\leq$, $\geq$, $=$, and CMP. This is primarily for floating-point values. Minimal complete definition: CMP or $\{<, =\}$.

```
trait StandardPartialOrder⟦Self extends StandardPartialOrder⟦Self⟧⟧
        excludes { Number }(* Until Number is an actual type. *)
    opr CMP(self, other : Self): Comparison
    opr <(self, other : Self): Boolean
    opr >(self, other : Self): Boolean
    opr =(self, other : Self): Boolean
    opr ≤(self, other : Self): Boolean
    opr ≥(self, other : Self): Boolean
end
```

StandardTotalOrder is the usual total order using $<$, $>$, $\leq$, $\geq$, $=$, and CMP. Most values that define a comparison should do so using this. Minimal complete definition: either CMP or $<$ (it is advisable to define $=$ in the latter case).

```
trait StandardTotalOrder⟦Self extends StandardTotalOrder⟦Self⟧⟧
        extends StandardPartialOrder⟦Self⟧
        excludes { Number }(* Until Number is an actual type. *)
    opr CMP(self, other : Self): Comparison
    opr ≥(self, other : Self): Boolean
end
```

## Assertions

*assert*(*flag* : Boolean): ()

*assert*(*flag*: Boolean, *failMsg*: String): ()

This version of *assert* checks the equality of its first two arguments; if unequal it includes the remaining arguments in its error indication.

*assert*($x$ : Any, $y$ : Any, *failMsg*: Any ...): ()

## Generator support

We say an object which extends Generator⟦$T$⟧ "generates objects of type $T$."

Generators are used to express iteration in Fortress. Every generated expression in Fortress (eg. for loop and comprehension) is desugared into calls to methods of Generator, chiefly the *generate* method.

Every generator has a notion of a "natural order" (which by default is unspecified), which describes the ordering of reduction operations, and also describes the order in which elements are produced by the sequential version of the same generator (given by the *seq*(self) method). The default implementation of *seq*(self) guarantees that these orders will match.

Note in particular that the natural order of a Generator must be consistent; that is, if $a$ `SEQV` $b$ then $a$ and $b$ must yield `SEQV` elements in the same natural order. However, note that unless a type specifically documents otherwise, no particular element ordering is guaranteed, nor is it necessary to guarantee that $a = b$ have the same natural order when equality is defined.

Note that more complex derived generators are specified further down in the definition of Generator. These have the same notions of natural order and by default are defined in terms of the $\mathit{generate}$ method.

Minimal complete definition of a $\text{Generator}$ is the $\mathit{generate}$ method.

`trait` $\text{Generator}[\![E]\!]$
    `excludes` $\{\,\text{Number}\,\}$

$\mathit{generate}$ is the core of $\text{Generator}$. It generates elements of type $E$ and passes them to the $\mathit{body}$ function. This generation can occur using any mixture of serial and parallel execution desired by the author of the generator; by default uses of a generator must assume every call to $\mathit{body}$ occurs in parallel.

The results of generation are combined using the reduction object $R$, which specifies a monoidal operation (associative and with an identity). Body results must be combined together following the natural order of the generator. The author of the generator is free to use the identity element anywhere desired in this computation, and to group reductions in any way desired; if no elements are generated, the identity must be returned.

`abstract` $\mathit{generate}[\![R]\!](r\colon \text{Reduction}[\![R]\!], \mathit{body}\colon E \rightarrow R)\colon R$

**Transforming generators into new generators**

$\mathit{map}$ applies a function $f$ to each element generated and yields the result. The resulting generator must have the same ordering and cross product properties as the generator from which it is derived.

$\mathit{map}[\![G]\!](f\colon E \rightarrow G)\colon \text{Generator}[\![G]\!]$

$\mathit{seq}$ produces a sequential version of the same generator, in which elements are produced in the generator's natural order.

$\mathit{seq}($`self`$)\colon \text{SequentialGenerator}[\![E]\!]$

Nesting of two generators; the innermost is data-dependent upon the outer one. This is specifically designed to be overloaded so that the combined generators have properties appropriate to the pairing. Because of the data dependency, the natural order of the nesting is the natural order of the inner generators, in the natural order the outer nesting produces them. So, for example, if we write:

$(0 \# 3).\mathit{nest}[\![\mathbb{Z}32]\!]($`fn` $(n : \mathbb{Z}32) : \text{Generator}[\![\mathbb{Z}32]\!] \Rightarrow ((n100) \# 4))$

then the natural order is 0,1,2,3,100,101,102,103,200,201,202,203.

$\mathit{nest}[\![G]\!](f\colon E \rightarrow \text{Generator}[\![G]\!])\colon \text{Generator}[\![G]\!]$

Cross product of two generators. This is specifically designed to be overloaded, such that pairs of independent generators can be combined to produce a generator which possibly interleaves the iteration spaces of the two generators. For example, we might combine

$(0 \# 16).\mathit{cross}(0 \# 32)$

such that it first splits the second range in half, then the first range in half, then the second, and so forth.

Consider a grid for which the rows are labeled from top to bottom with the elements of $a$ in natural order, and the columns are labeled from left to right with the elements of $g$ in natural order. Each point in the grid corresponds to a pair $(a, b)$ that must be generated by `self`$.\mathit{cross}(g)$. In the natural order of the cross product, an element must occur after those that lie above and to the left of it in the grid. By default the natural order of the cross product is left-to-right, top to bottom. Programmers must not rely on the default order, except that cross products involving one or more sequential generators are always performed in the default order. Note that this means that the following have the same natural order:

$\mathit{seq}(a).\mathit{cross}(b)$
$a.\mathit{cross}(\mathit{seq}(b))$
$\mathit{seq}(a).\mathit{cross}(\mathit{seq}(b))$

But $\mathit{seq}(a.\mathit{cross}(b))$ may have a different natural order.

$\mathit{cross}[\![G]\!](g\colon \mathrm{Generator}[\![G]\!])\colon \mathrm{Generator}[\![(E, G)]\!]$

**Derived generation operations**

$\mathit{mapReduce}$ is equivalent to $\mathit{generate}$, but takes an explicit $\mathit{join}$ and $\mathit{zero}$ which can have any type. It still assumes $\mathit{join}$ is associative and that $\mathit{zero}$ is the identity of $\mathit{join}$.

$\mathit{mapReduce}[\![R]\!](\mathit{body}\colon E \to R, \mathit{join} : (R, R) \to R, \mathit{zero} : R)\colon R$

$\mathit{reduce}$ works much like $\mathit{generate}$ or $\mathit{mapReduce}$, but has no body expression.

$\mathit{reduce}(j : (E, E) \to E, z : E) : E$
$\mathit{reduce}(r\colon \mathrm{Reduction}[\![E]\!]) : E$

$\mathit{loop}$ is a version of $\mathit{generate}$ which discards the $()$ results of the body computation. It can be used to translate reduction-variable-free `for` loops.

$\mathit{loop}(f : E \to ())\colon ()$

$x \in$ `self` holds if $x$ is generated by this generator. By default this is implemented using the naive $O(n)$ algorithm.

`opr` $\in(x : E,$ `self`$)\colon \mathrm{Boolean}$
`end`

The following stubs exist as a temporary workaround to shortcomings in interpreter type inference, and are intended for use by reduction desugaring.

$\_\_\mathit{generate}[\![E, R]\!](g : \mathrm{Generator}[\![E]\!], r\colon \mathrm{Reduction}[\![R]\!], b : E \to R)\colon R$

$\_\_\mathit{nest}[\![E_1, E_2]\!](g : \mathrm{Generator}[\![E_1]\!], f : E_1 \to \mathrm{Generator}[\![E_2]\!]) : \mathrm{Generator}[\![E_2]\!]$

$\_\_\mathit{map}[\![E, R]\!](g : \mathrm{Generator}[\![E]\!], f : E \to R)\colon \mathrm{Generator}[\![R]\!]$

`trait` $\mathrm{SequentialGenerator}[\![E]\!]$ `extends` $\{\ \mathrm{Generator}[\![E]\!]\ \}$
$\mathit{seq}($`self`$)$
$\mathit{map}[\![G]\!](f\colon E \to G)\colon \mathrm{SequentialGenerator}[\![G]\!]$
$\mathit{nest}[\![G]\!](f\colon E \to \mathrm{Generator}[\![G]\!])\colon \mathrm{Generator}[\![G]\!]$
$\mathit{cross}[\![F]\!](g : \mathrm{Generator}[\![F]\!])\colon \mathrm{Generator}[\![(E, F)]\!]$
`end`

`opr` $\in$ returns true if any element generated by its second argument is $=$ to its first argument. $x \notin g$ is simply $\neg(x \in g)$.

`opr` $\notin[\![E]\!](x\colon E, \mathit{this}\colon \mathrm{Generator}[\![E]\!])\colon \mathrm{Boolean}$

$\mathit{sequential}[\![T]\!](g : \mathrm{Generator}[\![T]\!]) : \mathrm{SequentialGenerator}[\![T]\!]$

### 24.1.1 The Maybe type

This trait makes excludes work without where clauses, and allows opr = to remain non-parametric.

```
value trait MaybeType extends Equality⟦MaybeType⟧ excludes Number


          not yet: “comprises Maybe⟦T⟧ where ⟦T⟧”
     abstract getter isJust() : Boolean
     opr =(self, other : MaybeType): Boolean
end
```

Maybe represents either Nothing or a single element of type $T$ ( Just⟦T⟧ ), which may be retrieved by calling *unJust*. An object of type Maybe⟦T⟧ can be used as a generator; it is either empty (Nothing) or generates the single element yielded by *unJust*, so there is no issue of canonical order or parallelism.

Thus, Just⟦T⟧ can be used as a single-element generator, and Nothing can be used as an empty generator.

```
value trait Maybe⟦T⟧
          extends { MaybeType, SequentialGenerator⟦T⟧, ZeroIndexed⟦T⟧ }
          comprises { Nothing⟦T⟧, Just⟦T⟧ }
     abstract getter unJust() : T throws NotFound
     abstract unJust(t : T): T
     abstract maybe⟦R⟧(nothingAction: () → R, justAction: T → R): R
end
value object Just⟦T⟧(x : T) extends Maybe⟦T⟧
     getter size()
     getter toString() : String
     getter isJust()
     getter unJust()
     unJust(_ : T): T
     generate⟦R⟧(_ : Reduction⟦R⟧, m : T → R): R
     opr [i : ℤ32] : T
     opr [r : Range⟦ℤ32⟧] : Maybe⟦T⟧
     map⟦G⟧(f: T → G): Just⟦G⟧
     cross⟦G⟧(g: Generator⟦G⟧): Generator⟦(T, G)⟧
     mapReduce⟦R⟧(m: T → R, _ : (R, R) → R, _ : R): R
     reduce(_ : (T, T) → T, _ : T) : T
     reduce(_: Reduction⟦T⟧) : T
     loop(f : T → ()): ()
     maybe⟦R⟧(_: () → R, justAction: T → R): R
     opr =(self, o : Just⟦T⟧): Boolean
end
```

Nothing will become a non-parametric singleton when we get where clauses working.

```
value object Nothing⟦T⟧ extends Maybe⟦T⟧
     getter size()
     getter isJust()
     getter unJust()
     getter toString() : String
     unJust(t : T) : T
     generate⟦R⟧(r : Reduction⟦R⟧, _ : T → R): R
     opr [_ : ℤ32]: T
```

```
    opr $[r : \mathrm{Range}[\![\mathbb{Z}32]\!]]$: $\mathrm{Nothing}[\![T]\!]$
    $map[\![G]\!](f\colon T \to G)\colon \mathrm{Nothing}[\![G]\!]$
    $cross[\![G]\!](g\colon \mathrm{Generator}[\![G]\!])\colon \mathrm{Generator}[\![(T, G)]\!]$
    $mapReduce[\![R]\!](\_\colon T \to R, \_ : (R, R) \to R, z : R)\colon R$
    $reduce(\_ : (T, T) \to T, z : T) : T$
    $reduce(r\colon \mathrm{Reduction}[\![T]\!]) : T$
    $loop(f : T \to ())\colon ()$
    $maybe[\![R]\!](nothingAction\colon () \to R, \_\colon T \to R)\colon R$
    opr =(self, $\_ : \mathrm{Nothing}[\![T]\!]$)
end
```

## Exception hierarchy

```
trait Exception comprises { UncheckedException, CheckedException }
end
```

(∗ Exceptions which are not checked ∗)

```
trait UncheckedException extends Exception excludes CheckedException
end

object FailCalled($s$ : String) extends UncheckedException
  $toString()$: String
end

object DivisionByZero extends UncheckedException
end

object UnpastingError extends UncheckedException
end

object CallerViolation extends UncheckedException
end

object CalleeViolation extends UncheckedException
end

object TestFailure extends UncheckedException
end

object ContractHierarchyViolation extends UncheckedException
end

object NoEqualityOnFunctions extends UncheckedException
end

object InvalidRange extends UncheckedException
end

object ForbiddenException($chain$ : Exception) extends UncheckedException
end
```

(∗ Should this be called “IndexNotFound” instead? ∗)

```
object NotFound extends UncheckedException
end

object IndexOutOfBounds extends UncheckedException
end

object NegativeLength extends UncheckedException
end
```

```
object IntegerOverflow extends UncheckedException
end
object RationalComparisonError extends UncheckedException
end
object FloatingComparisonError extends UncheckedException
end
(* Checked Exceptions *)
trait CheckedException extends Exception excludes UncheckedException
end
object CastError extends CheckedException
end
object IOFailure extends CheckedException
end
object MatchFailure extends CheckedException
end
(* SetsNotDisjoint? *)
object DisjointUnionError extends CheckedException
end
object APIMissing extends CheckedException
end
object APINameCollision extends CheckedException
end
object ExportedAPIMissing extends CheckedException
end
object HiddenAPIMissing extends CheckedException
end
object TryAtomicFailure extends CheckedException
end
(* Should take a spawned thread as an argument *)
object AtomicSpawnSynchronization extends {UncheckedException}
end
```

## Array support

```
trait HasRank extends Equality⟦HasRank⟧ excludes { Number, MaybeType }

   not yet: “ comprises Array⟦T, E, I⟧ where ⟦T, E, I⟧{ T extends Array⟦T, E, I⟧ } ”
   abstract rank() : ℤ32
   opr =(self, other : HasRank): Boolean
end
(* Declared Rank-n-ness *)
trait Rank⟦ nat n ⟧ extends HasRank
   rank() : ℤ32
end
```

Potemkin exclusion traits. Really we just want to say that "$\text{Rank}[\![n]\!]$ `excludes` $\text{Rank}[\![m]\!]$ `where` $\{\, m \neq n \,\}$", but we cannot yet.

`trait` $\text{Rank1}$ `extends` $\{\,\text{Rank}[\![1]\!]\}$ `excludes` $\{\,\text{Rank2}, \text{Rank3}, \text{Number}, \text{String}\,\}$
`end`

`trait` $\text{Rank2}$ `extends` $\{\,\text{Rank}[\![2]\!]\}$ `excludes` $\{\,\text{Rank3}, \text{Number}, \text{String}\,\}$
`end`

`trait` $\text{Rank3}$ `extends` $\{\,\text{Rank}[\![3]\!]\}$ `excludes` $\{\,\text{Number}, \text{String}\,\}$
`end`

The trait $\mathit{Indexed\_i}[\![n]\!]$ indicates that something has an $i^{th}$ dimension of size $n$. In general, anything which extends $\mathit{Indexed\_i}$ must also extend $\mathit{Indexed\_j}$ for $j < i$.

`trait` $\text{Indexed1}[\![$ `nat` $n\,]\!]$ `end`

`trait` $\text{Indexed2}[\![$ `nat` $n\,]\!]$ `end`

`trait` $\text{Indexed3}[\![$ `nat` $n\,]\!]$ `end`

The indexed trait indicates that an object of type $T$ can be indexed using type $I$ to obtain elements with type $E$.

An object $i$ that is an instance of $\text{Indexed}$ defines three basic things:

The indexing operator `opr` $[\,]$, which must be defined for every instance of the type.

A suite of generators: $i.\mathit{indices}$ generates the index space of the array. $i$ itself generates the values contained at those indices. $i.\mathit{indexValuePairs}$ yields pairs of $(\mathit{index}, \mathit{val})$. All of these share the same natural order. It is necessary to define one of $\mathit{indices}()$ and $\mathit{indexValuePairs}()$, in addition to $\mathit{generate}()$ (but the latter requirement can be dispensed by instead extending $\text{DelegatedIndexed}$).

A set of utility functions, $\mathit{assign}$, $\mathit{fill}$, and $\mathit{copy}$. Only $\mathit{fill}$ and $\mathit{copy}$ need to be defined.

`trait` $\text{Indexed}[\![E, I]\!]$ `extends` $\text{Generator}[\![E]\!]$

> $\mathit{isEmpty}()$ indicates whether there are any valid indices. It is defined as $\mathit{size}() = 0$.
> `getter` $\mathit{isEmpty}()\colon \text{Boolean}$
>
> $\mathit{size}()$ indicates the number of distinct valid indices that may be passed to indexing operations.
> `abstract getter` $\mathit{size}()\colon \mathbb{Z}32$
>
> $\mathit{bounds}()$ yields a range of indices that are valid for the indexed generator.
> `abstract getter` $\mathit{bounds}()\colon \text{FullRange}[\![I]\!]$
>
> $\mathit{indexValuePairs}()$ generates the elements of the indexed object (exactly those elements that are generated by the object itself), but each element is paired with its index. When we obtain $(i, v)$ from $\mathit{indexValuePairs}()$ we know that:
>
> - $\texttt{self}_i = v$
> - the $i$ are distinct and $i \in \mathit{bounds}()$
> - stripping away the $i$ yields exactly the results of $v \leftarrow \texttt{self}$
>
> This generator attempts to follow the structure of the underlying object as closely as possible.
> `getter` $\mathit{indexValuePairs}()\colon \text{Indexed}[\![(I, E), I]\!]$

$\mathit{indices}()$ yields the indices corresponding to the elements of the indexed object—it corresponds to the index component of $\mathit{indexValuePairs}()$ . This may in general be a subset of all the valid indices represented by $\mathit{bounds}()$ . This generator attempts to follow the structure of the underlying object as closely as possible.

```
    getter $\mathit{indices}()\colon \mathrm{Indexed}[\![I, I]\!]$
```

Indexing. $i \in \mathit{bounds}()$ must hold.

```
    abstract opr $[i\colon I]\colon\ E$
```

Indexing by ranges. The results are 0-based when the underlying index type has a notion of 0. This ensures consistency of behavior between types such as vectors that “only” support 0-indexing and types such as arrays that permit other choices of lower bounds. The easiest way to write the index by ranges operation for an instance of $\mathrm{Indexed}$ is to take advantage of indexing on the ranges themselves by writing $(\mathit{bounds}())[r]$ in order to narrow and bounds check the range $r$ and obtain a closed range of indices on the underlying data.

```
    abstract opr $[r\colon \mathrm{Range}[\![I]\!]]\colon\ \mathrm{Indexed}[\![E, I]\!]$
    opr $[\_\colon \mathrm{OpenRange}[\![\mathrm{Any}]\!]]\colon\ \mathrm{Indexed}[\![E, I]\!]$
```

Roughly speaking, $\mathit{ivmap}(f)$ is equivalent to $\mathit{indexValuePairs}.\mathit{map}(f)$ . However $\mathit{ivmap}$ is not merely a convenient shortcut. It is actually intended to create a copy of the underlying indexed structure when that is appropriate.

The usual $\mathit{map}$ function in $\mathrm{Generator}$ should do the same (and does for the instances in this library). Copying can be bad for space, but is complexity-preserving if the mapped generator is used more than once.

```
    $\mathit{ivmap}[\![R]\!](f\colon (I, E) \rightarrow R)\colon \mathrm{Indexed}[\![R, I]\!]$
    $\mathit{map}[\![R]\!](f\colon E \rightarrow R)\colon \mathrm{Indexed}[\![R, I]\!]$
end
trait $\mathrm{ZeroIndexed}[\![E]\!]$ extends $\mathrm{Indexed}[\![E, \mathbb{Z}32]\!]$
    $\mathit{bounds}()\colon \mathrm{FullRange}[\![\mathbb{Z}32]\!]$
    $\mathit{zip}[\![F]\!](g\colon \mathrm{ZeroIndexed}[\![F]\!])\colon \mathrm{ZeroIndexed}[\![(E, F)]\!]$
end
object $\mathrm{DefaultZip}[\![E, F]\!](e\colon \mathrm{ZeroIndexed}[\![E]\!], f\colon \mathrm{ZeroIndexed}[\![F]\!])$
        extends $\{\, \mathrm{ZeroIndexed}[\![(E, F)]\!], \mathrm{DelegatedIndexed}[\![(E, F), \mathbb{Z}32]\!] \,\}$
    getter $\mathit{size}()\colon \mathbb{Z}32$
    getter $\mathit{indices}()\colon \mathrm{Generator}[\![\mathbb{Z}32]\!]$
    opr $[i\colon \mathbb{Z}32]\colon (E, F)$
    opr $[r\colon \mathrm{Range}[\![\mathbb{Z}32]\!]]\colon\ \mathrm{ZeroIndexed}[\![(E, F)]\!]$
end
trait $\mathrm{LexicographicOrder}[\![T$ extends $\mathrm{LexicographicOrder}[\![T, E]\!], E]\!]$
        extends $\{\, \mathrm{StandardTotalOrder}[\![T]\!], \mathrm{ZeroIndexed}[\![E]\!] \,\}$
    opr CMP(self, $\mathit{other}\colon T)\colon \mathrm{TotalComparison}$
```

We give a specialized version of $=$ because it can fail faster than `CMP` by checking sizes early.

```
    opr $=($self$, \mathit{other}\colon T)\colon \mathrm{Boolean}$
end
$\mathit{toArray}[\![E]\!](g\colon \mathrm{Indexed}[\![E, \mathbb{Z}32]\!])\colon \mathrm{Array}[\![E, \mathbb{Z}32]\!]$
```

$\mathrm{DelegatedIndexed}$ is an indexed generator that has recourse to another indexed generator internally. By default, this in turn is defined in terms of $\mathit{indexValuePairs}()$ . Thus, it is only necessary to define either $\mathit{indexValuePairs}()$ or $\mathit{indices}()$ .

This class is designed for convenience; it should not be used as a type in running code, but only as a supertype in lieu of $\mathrm{Indexed}$.

```
trait $\mathrm{DelegatedIndexed}[\![E, I]\!]$ extends $\mathrm{Indexed}[\![E, I]\!]$
    getter $\mathit{generator}()\colon \mathrm{Generator}[\![E]\!]$
    getter $\mathit{size}()\colon \mathbb{Z}32$
    $\mathit{generate}[\![R]\!](r\colon \mathrm{Reduction}[\![R]\!], \mathit{body}\colon E \to R)\colon R$
    $\mathit{seq}(\mathtt{self})\colon \mathrm{SequentialGenerator}[\![E]\!]$
    $\mathit{cross}[\![G]\!](g\colon \mathrm{Generator}[\![G]\!])\colon \mathrm{Generator}[\![(E, G)]\!]$
    $\mathit{mapReduce}[\![R]\!](\mathit{body}\colon E \to R, \mathit{join} : (R, R) \to R, \mathit{zero} : R)\colon R$
    $\mathit{reduce}(j : (E, E) \to E, z : E) : E$
    $\mathit{reduce}(r\colon \mathrm{Reduction}[\![E]\!]) : E$
    $\mathit{loop}(f : E \to ())\colon ()$
end
```

The $\mathrm{MutableIndexed}$ trait is an indexed trait whose elements can be mutated using indexed assignment. Right now, we are using this type in a somewhat dangerous way, since, for example, $\mathrm{Array1}[\![E, b_0, s_0]\!]$ extends both $\mathrm{Indexed}[\![\mathrm{Array1}[\![E, b_0, s_0]\!], E, \mathbb{Z}32]\!]$ and $\mathrm{Indexed}[\![\mathrm{Array}[\![E, \mathbb{Z}32]\!], E, \mathbb{Z}32]\!]$. We will need to find a solution to this at some point.

```
trait $\mathrm{MutableIndexed}[\![E, I]\!]$
        extends $\{ \mathrm{Indexed}[\![E, I]\!] \}$
    abstract opr $[i : I] := (v : E) : ()$
```

For $\mathrm{Ranged}$ assignment, the extents of $r$ and $v.\mathit{bounds}()$ must match, but the lower bounds need not.

```
    abstract opr $[r : \mathrm{Range}[\![I]\!]] := (v : \mathrm{Indexed}[\![E, I]\!]) : ()$
    opr $[\_ : \mathrm{OpenRange}[\![\mathrm{Any}]\!]] := (v : \mathrm{Indexed}[\![E, I]\!]) : ()$
end
```

Array whose bounds are implicit rather than static, and which may be either mutable or immutable.

```
trait $\mathrm{ReadableArray}[\![E, I]\!]$
        extends $\{ \mathrm{HasRank}, \mathrm{Indexed}[\![E, I]\!] \}$
        comprises $\{ \mathrm{Array}[\![E, I]\!], \mathrm{ImmutableArray}[\![E, I]\!] \}$
```

Indexed functionality with more specific type information.

```
    abstract opr $[r : \mathrm{Range}[\![I]\!]] :\ \mathrm{ReadableArray}[\![E, I]\!]$
    abstract opr $[\_ : \mathrm{OpenRange}[\![\mathrm{Any}]\!]] :\ \mathrm{ReadableArray}[\![E, I]\!]$
    abstract $\mathit{ivmap}[\![R]\!](f : (I, E) \to R)\colon \mathrm{ReadableArray}[\![R, I]\!]$
    abstract $\mathit{map}[\![R]\!](f : E \to R)\colon \mathrm{ReadableArray}[\![R, I]\!]$
```

Shift the origin of an array. This should yield a new view of the same array; that is, initialization and/or update to either will be reflected in the other.

```
    abstract $\mathit{shift}(\mathit{newOrigin} : I) : \mathrm{ReadableArray}[\![E, I]\!]$
```

Initialize element at index $i$ with value $v$. This should occur once, before any other access or assignment occurs to element $i$. An error will be signaled if an uninitialized element is read or an initialized element is re-initialized.

```
    abstract $\mathit{init}(i : I, v : E)\colon ()$
```

Bulk initialization of an array using a given function or value.

```
    abstract fill(f : I → E) : ReadableArray⟦E, I⟧
    abstract fill(v : E) : ReadableArray⟦E, I⟧
    abstract copy() : ReadableArray⟦E, I⟧
```

Create a fresh array structurally identical to the present one, but holding elements of type $U$.

```
    abstract replica⟦U⟧() : ReadableArray⟦U, I⟧
    opr =(self, other : HasRank): Boolean
end
trait ImmutableArray⟦E, I⟧ extends { ReadableArray⟦E, I⟧ }
        excludes { Array⟦E, I⟧ }
    abstract opr [r : Range⟦I⟧] : ImmutableArray⟦E, I⟧
    abstract opr [_ : OpenRange⟦Any⟧] : ImmutableArray⟦E, I⟧
    abstract ivmap⟦R⟧(f : (I, E) → R): ImmutableArray⟦R, I⟧
    abstract map⟦R⟧(f : E → R): ImmutableArray⟦R, I⟧
    abstract shift(newOrigin : I) : ImmutableArray⟦E, I⟧
    abstract init(i : I, v : E): ()
    abstract fill(f : I → E) : ImmutableArray⟦E, I⟧
    abstract fill(v : E) : ImmutableArray⟦E, I⟧
    abstract copy() : ImmutableArray⟦E, I⟧
    abstract replica⟦U⟧() : ImmutableArray⟦U, I⟧
```

Thaw array (return mutable copy).

```
    abstract thaw() : Array⟦E, I⟧
end
trait Array⟦E, I⟧ extends { ReadableArray⟦E, I⟧, MutableIndexed⟦E, I⟧ }
    abstract opr [r : Range⟦I⟧] : Array⟦E, I⟧
    abstract opr [_ : OpenRange⟦Any⟧] : Array⟦E, I⟧
    abstract ivmap⟦R⟧(f : (I, E) → R): Array⟦R, I⟧
    abstract map⟦R⟧(f : E → R): Array⟦R, I⟧
    abstract shift(newOrigin : I) : Array⟦E, I⟧
    abstract init(i : I, v : E): ()
    abstract fill(f : I → E) : Array⟦E, I⟧
    abstract fill(v : E) : Array⟦E, I⟧
    abstract assign(f : I → E) : Array⟦E, I⟧
    abstract copy() : Array⟦E, I⟧
    abstract replica⟦U⟧() : Array⟦U, I⟧
```

Freeze array (return mutable copy).

```
    abstract freeze(): ImmutableArray⟦E, I⟧
end
```

Factory for arrays that returns an empty 0-indexed array of a given run-time-determined size.

```
array⟦E⟧(x : ℤ32) : Array⟦E, ℤ32⟧
array⟦E⟧(x : ℤ32, y : ℤ32) : Array⟦E, (ℤ32, ℤ32)⟧
array⟦E⟧(x : ℤ32, y : ℤ32, z : ℤ32) : Array⟦E, (ℤ32, ℤ32, ℤ32)⟧
```

Factory for immutable arrays that returns an empty 0-indexed array of a given run-time-determined size.

$\mathit{immutableArray}⟦E⟧(x:\mathbb{Z}32):\mathrm{ImmutableArray}⟦E,\mathbb{Z}32⟧$
$\mathit{immutableArray}⟦E⟧(x:\mathbb{Z}32, y:\mathbb{Z}32):\mathrm{ImmutableArray}⟦E,(\mathbb{Z}32,\mathbb{Z}32)⟧$
$\mathit{immutableArray}⟦E⟧(x:\mathbb{Z}32, y:\mathbb{Z}32, z:\mathbb{Z}32):\mathrm{ImmutableArray}⟦E,(\mathbb{Z}32,\mathbb{Z}32,\mathbb{Z}32)⟧$

$\mathit{primitiveArray}⟦E⟧(x:\mathbb{Z}32):\mathrm{Array}⟦E,\mathbb{Z}32⟧$

$\mathit{primitiveImmutableArray}⟦E⟧(x:\mathbb{Z}32):\mathrm{ImmutableArray}⟦E,\mathbb{Z}32⟧$

Array type supporting un-bounds-checked 0-based indexing. Useful for the internals of all the array functionality.

`trait` $\mathrm{ArrayTypeWith0}⟦E,I⟧$
    `extends` $\{\,\mathrm{ReadableArray}⟦E,I⟧, \mathrm{DelegatedIndexed}⟦E,I⟧\,\}$

  0-based non-bounds-checked indexing.
  `abstract` $\mathit{get}(i:I):E$
  `abstract` $\mathit{init}_0(i:I, e:E):()$
  `abstract` $\mathit{zeroIndices}():\mathrm{FullRange}⟦I⟧$

  Convert from $\mathit{base}$-based indexing to 0-based indexing, performing bounds checking.
  `abstract` $\mathit{offset}(i:I):I$

  Convert from 0-based indexing to $\mathit{base}$-based indexing.
  `abstract` $\mathit{toIndex}(i:I):I$
`end`

NOTE: StandardImmutableArrayType is a parent of StandardMutableArrayType. It therefore does not extend ImmutableArrayType as you might expect. Other types that extend it should also extend ImmutableArrayType explicitly.

`trait` $\mathrm{StandardImmutableArrayType}⟦T$ `extends` $\mathrm{StandardImmutableArrayType}⟦T,E,I⟧, E, I⟧$
    `extends` $\{\,\mathrm{ArrayTypeWith0}⟦E,I⟧\,\}$

  CONCRETE GETTERS: Default implementations of getters based on abstract methods in StandardArrayType.
  `getter` $\mathit{indices}():\mathrm{Indexed}⟦I,I⟧$
  `getter` $\mathit{indexValuePairs}():\mathrm{Indexed}⟦(I,E),I⟧$
  `getter` $\mathit{generator}():\mathrm{Indexed}⟦E,I⟧$

  CONCRETE METHODS: Default implementations of most array stuff based on the above. The things we cannot provide are anything involving replica.
  `opr` $[i:I]:E$
  $\mathit{init}(i:I, v:E)$
  $\mathit{generate}⟦R⟧(r:\mathrm{Reduction}⟦R⟧, \mathit{body}:E\to R):R$
  $\mathit{seq}($`self`$):\mathrm{SequentialGenerator}⟦E⟧$
  $\mathit{fill}(f:I\to E):T$
  $\mathit{fill}(v:E):T$
  `abstract` $\mathit{copy}():T$
`end`

`trait` $\mathrm{StandardMutableArrayType}⟦T$ `extends` $\mathrm{StandardMutableArrayType}⟦T,E,I⟧, E, I⟧$
  `extends` $\{\,\mathrm{StandardImmutableArrayType}⟦T,E,I⟧, \mathrm{Array}⟦E,I⟧\,\}$

0-based non-bounds-checked indexing.

```
    abstract $put(i:I, e:E):()$
    opr $[i:I] := (v:E):()$
    opr $[r:\mathrm{Range}[\![I]\!]] := (a:\mathrm{Indexed}[\![E,I]\!]):()$
    $assign(v:T):T$
    $assign(f:I \to E):T$
end
```

Canonical partitioning of a positive number $x$ into two pieces. If $(a,b) = partition(n)$ and $n > 0$ then $0 < a <= b$, $n = a + b$. As it turns out we choose $a$ to be the largest power of $2 < n$.

$partition(x:\mathbb{Z}32):(\mathbb{Z}32, \mathbb{Z}32)$

A $\mathrm{ReadableArray1}[\![T, b_0, s_0]\!]$ is an arbitrary 1-dimensional array whose $s_0$ elements are of type $T$, and whose lowest index is $b_0$ .

The natural order of all generators is from $b_0$ to $b_0 + s_0 - 1$ .

```
trait $\mathrm{ReadableArray1}[\![T, \texttt{nat}\ b_0, \texttt{nat}\ s_0]\!]$
        extends { $\mathrm{Indexed1}[\![s_0]\!], \mathrm{Rank1}, \mathrm{ArrayTypeWith0}[\![T, \mathbb{Z}32]\!]$ }
        comprises { $\mathrm{ImmutableArray1}[\![T, b_0, s_0]\!], \mathrm{Array1}[\![T, b_0, s_0]\!]$ }
    getter $size():\mathbb{Z}32$
    getter $bounds():\mathrm{FullRange}[\![\mathbb{Z}32]\!]$
    abstract getter $mutability():\mathrm{String}$
    getter $toString()$
    $subarray[\![\texttt{nat}\ b, \texttt{nat}\ s, \texttt{nat}\ o]\!]():\mathrm{ReadableArray1}[\![T, b, s]\!]$

    Offset converts from $b_0$ -indexing to 0-indexing, bounds checking en route.
    $offset(i:\mathbb{Z}32):\mathbb{Z}32$
    $toIndex(i:\mathbb{Z}32):\mathbb{Z}32$
    $zeroIndices():\mathrm{FullRange}[\![\mathbb{Z}32]\!]$
end
trait $\mathrm{ImmutableArray1}[\![T, \texttt{nat}\ b_0, \texttt{nat}\ s_0]\!]$
    extends { $\mathrm{StandardImmutableArrayType}[\![\mathrm{ImmutableArray1}[\![T, b_0, s_0]\!], T, \mathbb{Z}32]\!]$,
              $\mathrm{ImmutableArray}[\![T, \mathbb{Z}32]\!], \mathrm{ReadableArray1}[\![T, b_0, s_0]\!]$ }
    getter $mutability():\mathrm{String}$
    $shift(o:\mathbb{Z}32):\mathrm{ImmutableArray}[\![T, \mathbb{Z}32]\!]$
    opr $[r:\mathrm{Range}[\![\mathbb{Z}32]\!]] : \mathrm{ImmutableArray}[\![T, \mathbb{Z}32]\!]$
    opr $[\_:\mathrm{OpenRange}[\![\mathbb{Z}32]\!]] : \mathrm{ImmutableArray1}[\![T, 0, s_0]\!]$
    opr $[\_:\mathrm{OpenRange}[\![\mathrm{Any}]\!]] : \mathrm{ImmutableArray1}[\![T, 0, s_0]\!]$

    $subarray$ selects a subarray of this array based on static parameters. $b \# s$ are the new bounds of the array; $o$
    is the index of the subarray within the current array.
    $subarray[\![\texttt{nat}\ b, \texttt{nat}\ s, \texttt{nat}\ o]\!]():\mathrm{ImmutableArray1}[\![T, b, s]\!]$

    The $replica$ method returns a replica of the array (similar layout etc.) but with a different element type.
    $replica[\![U]\!]():\mathrm{ImmutableArray1}[\![U, b_0, s_0]\!]$
    $copy():\mathrm{ImmutableArray1}[\![T, b_0, s_0]\!]$
    $thaw():\mathrm{Array1}[\![T, b_0, s_0]\!]$
```

$\quad map[\![R]\!](f : T \to R)$: $\mathrm{ImmutableArray1}[\![R, b_0, s_0]\!]$
$\quad ivmap[\![R]\!](f : (\mathbb{Z}32, T) \to R)$: $\mathrm{ImmutableArray1}[\![R, b_0, s_0]\!]$
`end`

$\mathrm{Array1}[\![T, b_0, s_0]\!]$ is a 1-dimension array whose $s_0$ elements are of type $T$, and whose lowest index is $b_0$.
`trait` $\mathrm{Array1}[\![T,$ `nat` $b_0,$ `nat` $s_0]\!]$
$\quad$ `extends` $\{ \mathrm{ReadableArray1}[\![T, b_0, s_0]\!],$
$\qquad \mathrm{StandardMutableArrayType}[\![\mathrm{Array1}[\![T, b_0, s_0]\!], T, \mathbb{Z}32]\!] \}$
$\quad$ `excludes` $\{\mathrm{Number}, \mathrm{String}\}$
$\quad$ `getter` $mutability() : \mathrm{String}$
$\quad shift(o : \mathbb{Z}32)$: $\mathrm{Array}[\![T, \mathbb{Z}32]\!]$
$\quad$ `opr` $[r : \mathrm{Range}[\![\mathbb{Z}32]\!]] :\ \mathrm{Array}[\![T, \mathbb{Z}32]\!]$
$\quad$ `opr` $[\_ : \mathrm{OpenRange}[\![\mathbb{Z}32]\!]] :\ \mathrm{Array1}[\![T, 0, s_0]\!]$
$\quad$ `opr` $[\_ : \mathrm{OpenRange}[\![\mathrm{Any}]\!]] :\ \mathrm{Array1}[\![T, 0, s_0]\!]$
$\quad subarray[\![$`nat` $b,$ `nat` $s,$ `nat` $o]\!]() : \mathrm{Array1}[\![T, b, s]\!]$
$\quad replica[\![U]\!]() : \mathrm{Array1}[\![U, b_0, s_0]\!]$
$\quad copy() : \mathrm{Array1}[\![T, b_0, s_0]\!]$
$\quad freeze() : \mathrm{ImmutableArray1}[\![T, b_0, s_0]\!]$
$\quad map[\![R]\!](f : T \to R)$: $\mathrm{Array1}[\![R, b_0, s_0]\!]$
$\quad ivmap[\![R]\!](f : (\mathbb{Z}32, T) \to R)$: $\mathrm{Array1}[\![R, b_0, s_0]\!]$
`end`

`trait` $\mathrm{Vector}[\![T$ `extends` $\mathrm{Number},$ `nat` $s_0]\!]$ `extends` $\mathrm{Array1}[\![T, 0, s_0]\!]$
$\quad add(v : \mathrm{Vector}[\![T, s_0]\!])$: $\mathrm{Vector}[\![T, s_0]\!]$
$\quad subtract(v : \mathrm{Vector}[\![T, s_0]\!])$: $\mathrm{Vector}[\![T, s_0]\!]$
$\quad negate()$: $\mathrm{Vector}[\![T, s_0]\!]$
$\quad scale(t : T)$: $\mathrm{Vector}[\![T, s_0]\!]$
$\quad pmul(v : \mathrm{Vector}[\![T, s_0]\!])$: $\mathrm{Vector}[\![T, s_0]\!]$
$\quad dot(v : \mathrm{Vector}[\![T, s_0]\!])$: $T$
`end`

$\_\_builtinFactory1$ must be a non-overloaded 0-parameter factory for 1-D arrays. The type parameters are enshrined in `LHSEvaluator.java` and `NonPrimitive.java`; the factory name is enshrined in `WellKnownNames.java`. There must be some factory, named in this file, with this type signature. A similar thing is true for $k$-dimensional array types.
$\_\_builtinFactory1[\![T,$ `nat` $b_0,$ `nat` $s_0]\!]() : \mathrm{Array1}[\![T, b_0, s_0]\!]$

$\_\_immutableFactory1$ is a non-overloaded 0-parameter factory for 0-indexed 1-D write-once arrays. It is also mentioned in `WellKnownNames` as it is used to allocate storage for varargs.
$\_\_immutableFactory1[\![T,$ `nat` $b_0,$ `nat` $s_0]\!]() : \mathrm{Array1}[\![T, b_0, s_0]\!]$

$array_1[\![T,$ `nat` $s_0]\!]() : \mathrm{Array1}[\![T, 0, s_0]\!]$
$array_1[\![T,$ `nat` $s_0]\!](v : T) : \mathrm{Array1}[\![T, 0, s_0]\!]$
$array_1[\![T,$ `nat` $s_0]\!](f : \mathbb{Z}32 \to T) : \mathrm{Array1}[\![T, 0, s_0]\!]$
$immutableArray_1[\![T,$ `nat` $s_0]\!]()$: $\mathrm{ImmutableArray1}[\![T, 0, s_0]\!]$

$vector$ is the same as $array_1$, but specialized to numeric type arguments.
$vector[\![T$ `extends` $\mathrm{Number},$ `nat` $s_0]\!]() : \mathrm{Vector}[\![T, s_0]\!]$
$vector[\![T$ `extends` $\mathrm{Number},$ `nat` $s_0]\!](v : T) : \mathrm{Vector}[\![T, s_0]\!]$

$vector[\![T\ \texttt{extends}\ \text{Number}, \texttt{nat}\ s_0]\!](f : \mathbb{Z}32 \to T) : \text{Vector}[\![T, s_0]\!]$

$\texttt{opr}\ +[\![T\ \texttt{extends}\ \text{Number}, \texttt{nat}\ n, \texttt{nat}\ m]\!]$
$\quad(me : \text{Vector}[\![T, n]\!], other : \text{Vector}[\![T, n]\!]) : \text{Vector}[\![T, n]\!]$

$\texttt{opr}\ -[\![T\ \texttt{extends}\ \text{Number}, \texttt{nat}\ n, \texttt{nat}\ m]\!]$
$\quad(me : \text{Vector}[\![T, n]\!], other : \text{Vector}[\![T, n]\!]) : \text{Vector}[\![T, n]\!]$

$\texttt{opr}\ -[\![T\ \texttt{extends}\ \text{Number}, \texttt{nat}\ n, \texttt{nat}\ m]\!]$
$\quad(me : \text{Vector}[\![T, n]\!]) : \text{Vector}[\![T, n]\!]$

$pmul[\![T\ \texttt{extends}\ \text{Number}, \texttt{nat}\ k]\!]$
$\quad(a : \text{Vector}[\![T, k]\!], b : \text{Vector}[\![T, k]\!]) : \text{Vector}[\![T, k]\!]$

$\texttt{opr}\ \cdot[\![T\ \texttt{extends}\ \text{Number}, \texttt{nat}\ n, \texttt{nat}\ m, \texttt{nat}\ p]\!]$
$\quad(me : \text{Vector}[\![T, n]\!], other : \text{Vector}[\![T, n]\!]) : T$

$\texttt{opr juxtaposition}\ [\![T\ \texttt{extends}\ \text{Number}, \texttt{nat}\ n, \texttt{nat}\ m, \texttt{nat}\ p]\!]$
$\quad(me : \text{Vector}[\![T, n]\!], other : \text{Vector}[\![T, n]\!]) : T$

$\texttt{opr}\ \cdot[\![T\ \texttt{extends}\ \text{Number}, \texttt{nat}\ n, \texttt{nat}\ m, \texttt{nat}\ p]\!]$
$\quad(me : \text{Vector}[\![T, n]\!], other : T) : \text{Vector}[\![T, n]\!]$

$\texttt{opr juxtaposition}\ [\![T\ \texttt{extends}\ \text{Number}, \texttt{nat}\ n, \texttt{nat}\ m, \texttt{nat}\ p]\!]$
$\quad(me : \text{Vector}[\![T, n]\!], other : T) : \text{Vector}[\![T, n]\!]$

$\texttt{opr}\ \cdot[\ T\ \texttt{extends}\ \text{Number}, \texttt{nat}\ n, \texttt{nat}\ m, \texttt{nat}\ p]\!]$
$\quad(other : T, me : \text{Vector}[\![T, n]\!]) : \text{Vector}[\![T, n]\!]$

$\texttt{opr juxtaposition}\ [\![T\ \texttt{extends}\ \text{Number}, \texttt{nat}\ n, \texttt{nat}\ m, \texttt{nat}\ p]\!]$
$\quad(other : T, me : \text{Vector}[\![T, n]\!]) : \text{Vector}[\![T, n]\!]$

$squaredNorm[\![T\ \texttt{extends}\ \text{Number}, \texttt{nat}\ s_0]\!](a : \text{Vector}[\![T, s_0]\!]) : T$

$\texttt{opr}\ \|[\![T\ \texttt{extends}\ \text{Number}, \texttt{nat}\ k]\!] me : \text{Vector}[\![T, k]\!]\| : \mathbb{R}64$

$\text{Array2}[\![T, b_0, s_0, b_1, s_1]\!]$ is the type of 2-dimensional arrays of element type $T$, with size $s_0$ in the first dimension and $s_1$ in the second dimension and lowest index $(b_0, b_1)$. Natural order for all generators in each dimension is from $b$ to $b + s - 1$; the overall order of elements need only be consistent with the cross product of these orderings (see Generator.$cross()$).

$\texttt{trait}\ \text{Array2}[\![T, \texttt{nat}\ b_0, \texttt{nat}\ s_0, \texttt{nat}\ b_1, \texttt{nat}\ s_1]\!]$
$\quad\texttt{extends}\ \{\ \text{Indexed1}[\![s_0]\!], \text{Indexed2}[\![s_1]\!], \text{Rank2},$
$\qquad\text{StandardMutableArrayType}[\![\text{Array2}[\![T, b_0, s_0, b_1, s_1]\!], T, (\mathbb{Z}32, \mathbb{Z}32)]\!]\ \}$
$\quad\texttt{excludes}\ \{\ \text{Number}, \text{String}\ \}$
$\texttt{getter}\ size() : \mathbb{Z}32$
$\texttt{getter}\ bounds() : \text{FullRange}[\![(\mathbb{Z}32, \mathbb{Z}32)]\!]$
$\texttt{getter}\ toString()$

Translate from $b_0$, $b_1$ -indexing to 0-indexing, checking bounds.

$offset(t : (\mathbb{Z}32, \mathbb{Z}32)) : (\mathbb{Z}32, \mathbb{Z}32)$
$toIndex(t : (\mathbb{Z}32, \mathbb{Z}32)) : (\mathbb{Z}32, \mathbb{Z}32)$
$\texttt{opr}\ [x : \mathbb{Z}32, y : \mathbb{Z}32] := (v : T) : ()$
$\texttt{opr}\ [r : \text{Range}[\![(\mathbb{Z}32, \mathbb{Z}32)]\!]] : \text{Array}[\![T, (\mathbb{Z}32, \mathbb{Z}32)]\!]$
$\texttt{opr}\ [\_ : \text{OpenRange}[\![\mathbb{Z}32]\!]] : \text{Array2}[\![T, 0, s_0, 0, s_1]\!]$
$\texttt{opr}\ [\_ : \text{OpenRange}[\![\text{Any}]\!]] : \text{Array2}[\![T, 0, s_0, 0, s_1]\!]$
$shift(t : (\mathbb{Z}32, \mathbb{Z}32)) : \text{Array}[\![T, (\mathbb{Z}32, \mathbb{Z}32)]\!]$

2-D subarray given static subarray parameters. $(bo_1, bo_2)$ # $(so_1, so_2)$ are output bounds. The result is the subarray starting at $(o_0, o_1)$ in the original array.

$\mathit{subarray}[\![\texttt{nat}\ bo_0, \texttt{nat}\ so_0, \texttt{nat}\ bo_1, \texttt{nat}\ so_1, \texttt{nat}\ o_0, \texttt{nat}\ o_1]\!]$
$\quad (): \mathrm{Array2}[\![T, bo_0, so_0, bo_1, so_1]\!]$

$\mathit{zeroIndices}() : \mathrm{FullRange}[\![(\mathbb{Z}32, \mathbb{Z}32)]\!]$

$\mathit{replica}[\![U]\!]() : \mathrm{Array2}[\![U, b_0, s_0, b_1, s_1]\!]$
$\mathit{copy}() : \mathrm{Array2}[\![T, b_0, s_0, b_1, s_1]\!]$
$\mathit{put}(t : (\mathbb{Z}32, \mathbb{Z}32), v : T) : ()$
$\mathit{get}(t : (\mathbb{Z}32, \mathbb{Z}32)) : T$
$t() : \mathrm{Array2}[\![T, b_1, s_1, b_0, s_0]\!]$
(∗ Copied here for better return type information. ∗)
$\mathit{map}[\![R]\!](f : T \rightarrow R): \mathrm{Array2}[\![R, b_0, s_0, b_1, s_1]\!]$
$\mathit{ivmap}[\![R]\!](f : ((\mathbb{Z}32, \mathbb{Z}32), T) \rightarrow R): \mathrm{Array2}[\![R, b_0, s_0, b_1, s_1]\!]$

$\mathit{freeze}() : \mathrm{ImmutableArray}[\![T, (\mathbb{Z}32, \mathbb{Z}32)]\!]$
`end`

`trait` $\mathrm{Matrix}[\![T$ `extends` $\mathrm{Number}, \texttt{nat}\ s_0, \texttt{nat}\ s_1]\!]$ `extends` $\mathrm{Array2}[\![T, 0, s_0, 0, s_1]\!]$
  `abstract` $\mathit{add}(v : \mathrm{Matrix}[\![T, s_0, s_1]\!]): \mathrm{Matrix}[\![T, s_0, s_1]\!]$
  `abstract` $\mathit{subtract}(v : \mathrm{Matrix}[\![T, s_0, s_1]\!]): \mathrm{Matrix}[\![T, s_0, s_1]\!]$
  `abstract` $\mathit{negate}(): \mathrm{Matrix}[\![T, s_0, s_1]\!]$
  `abstract` $\mathit{scale}(t : T): \mathrm{Matrix}[\![T, s_0, s_1]\!]$
  `abstract` $\mathit{mul}[\![\texttt{nat}\ s_2]\!](\mathit{other}: \mathrm{Matrix}[\![T, s_1, s_2]\!]): \mathrm{Matrix}[\![T, s_0, s_2]\!]$
  `abstract` $\mathit{rmul}(v: \mathrm{Vector}[\![T, s_1]\!]): \mathrm{Vector}[\![T, s_0]\!]$
  `abstract` $\mathit{lmul}(v: \mathrm{Vector}[\![T, s_0]\!]): \mathrm{Vector}[\![T, s_1]\!]$
  `abstract` $t(): \mathrm{Matrix}[\![T, s_1, s_0]\!]$
`end`

$\_\_\mathit{builtinFactory2}[\![T, \texttt{nat}\ b_0, \texttt{nat}\ s_0, \texttt{nat}\ b_1, \texttt{nat}\ s_1]\!]() : \mathrm{Array2}[\![T, b_0, s_0, b_1, s_1]\!]$

$array_2$ is a factory for 0-based 2-D arrays.

$array_2[\![T, \texttt{nat}\ s_0, \texttt{nat}\ s_1]\!]() : \mathrm{Array2}[\![T, 0, s_0, 0, s_1]\!]$
$array_2[\![T, \texttt{nat}\ s_0, \texttt{nat}\ s_1]\!](v : T) : \mathrm{Array2}[\![T, 0, s_0, 0, s_1]\!]$
$array_2[\![T, \texttt{nat}\ s_0, \texttt{nat}\ s_1]\!](f : (\mathbb{Z}32, \mathbb{Z}32) \rightarrow T) : \mathrm{Array2}[\![T, 0, s_0, 0, s_1]\!]$

$\mathit{matrix}$ is the same as $array_2$, but specialized to numeric type arguments, except that the default value (if given) is used to construct a multiple of the identity matrix.

$\mathit{matrix}[\![T$ `extends` $\mathrm{Number}, \texttt{nat}\ s_0, \texttt{nat}\ s_1]\!]() : \mathrm{Matrix}[\![T, s_0, s_1]\!]$
$\mathit{matrix}[\![T$ `extends` $\mathrm{Number}, \texttt{nat}\ s_0, \texttt{nat}\ s_1]\!](v : T) : \mathrm{Matrix}[\![T, s_0, s_1]\!]$

`opr` $+[\![T$ `extends` $\mathrm{Number}, \texttt{nat}\ n, \texttt{nat}\ m]\!]$
  $(\mathit{me} : \mathrm{Matrix}[\![T, n, m]\!], \mathit{other} : \mathrm{Matrix}[\![T, n, m]\!]): \mathrm{Matrix}[\![T, n, m]\!]$

`opr` $-[\![T$ `extends` $\mathrm{Number}, \texttt{nat}\ n, \texttt{nat}\ m]\!]$
  $(\mathit{me} : \mathrm{Matrix}[\![T, n, m]\!], \mathit{other} : \mathrm{Matrix}[\![T, n, m]\!]) :\ \mathrm{Matrix}[\![T, n, m]\!]$

`opr` $-[\![T$ `extends` $\mathrm{Number}, \texttt{nat}\ n, \texttt{nat}\ m]\!]$
  $(\mathit{me} : \mathrm{Matrix}[\![T, n, m]\!]): \mathrm{Matrix}[\![T, n, m]\!]$

Matrix multiplication.

`opr` $\cdot[\![T$ `extends` $\mathrm{Number}, \texttt{nat}\ n, \texttt{nat}\ m, \texttt{nat}\ p]\!]$
  $(\mathit{me} : \mathrm{Matrix}[\![T, n, m]\!], \mathit{other} : \mathrm{Matrix}[\![T, m, p]\!]): \mathrm{Matrix}[\![T, n, p]\!]$

`opr juxtaposition` $[\![T$ `extends` $\mathrm{Number}, \texttt{nat}\ n, \texttt{nat}\ m, \texttt{nat}\ p]\!]$

$(\mathit{me}:\text{Matrix}[\![T,n,m]\!], \mathit{other}:\text{Matrix}[\![T,m,p]\!]):\text{Matrix}[\![T,n,p]\!]$

Matrix-vector multiplication.

`opr` $\cdot[\![T$ `extends` $\text{Number}$, `nat` $n$, `nat` $m$, `nat` $p]\!]$
    $(\mathit{me}:\text{Matrix}[\![T,n,m]\!], v:\text{Vector}[\![T,m]\!]):\text{Vector}[\![T,n]\!]$

`opr juxtaposition` $[\![T$ `extends` $\text{Number}$, `nat` $n$, `nat` $m$, `nat` $p]\!]$
  $(\mathit{me}:\text{Matrix}[\![T,n,m]\!], v:\text{Vector}[\![T,m]\!]):\text{Vector}[\![T,n]\!]$

Vector-matrix multiplication.

`opr` $\cdot[\![T$ `extends` $\text{Number}$, `nat` $n$, `nat` $m$, `nat` $p]\!]$
    $(v:\text{Vector}[\![T,n]\!], \mathit{me}:\text{Matrix}[\![T,n,m]\!]):\text{Vector}[\![T,m]\!]$

`opr juxtaposition` $[\![T$ `extends` $\text{Number}$, `nat` $n$, `nat` $m$, `nat` $p]\!]$
  $(v:\text{Vector}[\![T,n]\!], \mathit{me}:\text{Matrix}[\![T,n,m]\!]):\text{Vector}[\![T,m]\!]$

`opr` $\cdot[\![T$ `extends` $\text{Number}$, `nat` $n$, `nat` $m$, `nat` $p]\!]$
    $(\mathit{me}: \text{Matrix}[\![T,n,m]\!], \mathit{other}: T): \text{Matrix}[\![T,n,m]\!]$

`opr juxtaposition` $[\![T$ `extends` $\text{Number}$, `nat` $n$, `nat` $m$, `nat` $p]\!]$
  $(\mathit{me}: \text{Matrix}[\![T,n,m]\!], \mathit{other}: T): \text{Matrix}[\![T,n,m]\!]$

`opr` $\cdot[\![T$ `extends` $\text{Number}$, `nat` $n$, `nat` $m$, `nat` $p]\!]$
    $(\mathit{other}: T, \mathit{me}: \text{Matrix}[\![T,n,m]\!]): \text{Matrix}[\![T,n,m]\!]$

`opr juxtaposition` $[\![T$ `extends` $\text{Number}$, `nat` $n$, `nat` $m$, `nat` $p]\!]$
  $(\mathit{other}: T, \mathit{me}: \text{Matrix}[\![T,n,m]\!]): \text{Matrix}[\![T,n,m]\!]$

$\text{Array3}[\![T,b_0,s_0,b_1,s_1,b_2,s_2]\!]$ is the type of 3-dimensional arrays of element type $T$, with size $s_i$ in the $i^{th}$ dimension and lowest index $(b_0,b_1,b_2)$. Natural order for all generators in each dimension is from $b$ to $b+s-1$; the overall order of elements need only be consistent with the cross product of these orderings (see $\text{Generator}.\mathit{cross}()$).

`trait` $\text{Array3}[\![T$, `nat` $b_0$, `nat` $s_0$, `nat` $b_1$, `nat` $s_1$, `nat` $b_2$, `nat` $s_2]\!]$
  `extends` $\{\text{Indexed1}[\![s_0]\!], \text{Indexed2}[\![s_1]\!], \text{Indexed3}[\![s_2]\!], \text{Rank3},$
      $\text{StandardMutableArrayType}[\![\text{Array3}[\![T,b_0,s_0,b_1,s_1,b_2,s_2]\!], T,$
        $(\mathbb{Z}32,\mathbb{Z}32,\mathbb{Z}32)]\!]\}$
  `excludes` $\{\text{Number}, \text{String}\}$
  `getter` $\mathit{size}():\mathbb{Z}32$
  `getter` $\mathit{bounds}():\text{FullRange}[\![(\mathbb{Z}32,\mathbb{Z}32,\mathbb{Z}32)]\!]$
  `getter` $\mathit{toString}():\text{String}$

  Again, $\mathit{offset}$ performs bounds checking and shifts to 0-indexing.
  $\mathit{offset}(t:(\mathbb{Z}32,\mathbb{Z}32,\mathbb{Z}32)):(\mathbb{Z}32,\mathbb{Z}32,\mathbb{Z}32)$
  $\mathit{toIndex}(t:(\mathbb{Z}32,\mathbb{Z}32,\mathbb{Z}32)):(\mathbb{Z}32,\mathbb{Z}32,\mathbb{Z}32)$

  And $\mathit{get}$ and $\mathit{put}$ are 0-indexed without bounds checks.
  `abstract` $\mathit{put}(t:(\mathbb{Z}32,\mathbb{Z}32,\mathbb{Z}32), v:T):()$
  `abstract` $\mathit{get}(t:(\mathbb{Z}32,\mathbb{Z}32,\mathbb{Z}32)):T$
  `opr` $[i:\mathbb{Z}32, j:\mathbb{Z}32, k:\mathbb{Z}32] := (v:T)$
  `opr` $[r:\text{Range}[\![(\mathbb{Z}32,\mathbb{Z}32,\mathbb{Z}32)]\!]]:\text{Array}[\![T,(\mathbb{Z}32,\mathbb{Z}32,\mathbb{Z}32)]\!]$
  `opr` $[\_:\text{OpenRange}[\![\mathbb{Z}32]\!]]: \text{Array3}[\![T,0,s_0,0,s_1,0,s_2]\!]$
  `opr` $[\_:\text{OpenRange}[\![\text{Any}]\!]]: \text{Array3}[\![T,0,s_0,0,s_1,0,s_2]\!]$

```
        $shift(t : (\mathbb{Z}32, \mathbb{Z}32, \mathbb{Z}32))\colon \mathrm{Array}[\![T, (\mathbb{Z}32, \mathbb{Z}32)]\!]$
```

2-D subarray given static subarray parameters. $(bo_1, bo_2)$ # $(so_1, so_2)$ are output bounds. The result is the subarray starting at $(o_0, o_1)$ in the original array.

```
        $subarray[\![$nat $bo_0$, nat $so_0$, nat $bo_1$, nat $so_1$, nat $bo_2$, nat $so_2$,
                  nat $o_0$, nat $o_1$, nat $o_2]\!]$
                    $()\colon \mathrm{Array3}[\![T, bo_0, so_0, bo_1, so_1, bo_2, so_2]\!]$
        $zeroIndices() : \mathrm{FullRange}[\![(\mathbb{Z}32, \mathbb{Z}32, \mathbb{Z}32)]\!]$
        $replica[\![U]\!]() : \mathrm{Array3}[\![U, b_0, s_0, b_1, s_1, b_2, s_2]\!]$
        $copy() : \mathrm{Array3}[\![T, b_0, s_0, b_1, s_1, b_2, s_2]\!]$
        $map[\![R]\!](f : T \to R)\colon \mathrm{Array3}[\![R, b_0, s_0, b_1, s_1, b_2, s_2]\!]$
        $ivmap[\![R]\!](f : ((\mathbb{Z}32, \mathbb{Z}32, \mathbb{Z}32), T) \to R)\colon \mathrm{Array3}[\![R, b_0, s_0, b_1, s_1, b_2, s_2]\!]$
        $freeze() : \mathrm{ImmutableArray}[\![T, (\mathbb{Z}32, \mathbb{Z}32, \mathbb{Z}32)]\!]$
end
$\_\_builtinFactory3[\![T,$ nat $b_0$, nat $s_0$, nat $b_1$, nat $s_1$, nat $b_2$, nat $s_2]\!]()$ :
            $\mathrm{Array3}[\![T, b_0, s_0, b_1, s_1, b_2, s_2]\!]$
$array_3[\![T,$ nat $s_0$, nat $s_1$, nat $s_2]\!]() : \mathrm{Array3}[\![T, 0, s_0, 0, s_1, 0, s_2]\!]$
```

## Reductions

```
trait Reduction[\![ R ]\!]
        abstract getter $toString() : \mathrm{String}$
        abstract $empty()\colon R$
        abstract $join(a\colon R, b\colon R)\colon R$
end
object VoidReduction extends Reduction[\![()]\!]
        getter $toString()$
        $empty()\colon ()$
        $join(a\colon (), b\colon ())\colon ()$
end
```

Hack to permit any Number to work non-parametrically.

```
object SumReduction extends Reduction[\![Number]\!]
        getter $toString()$
        $empty()\colon \mathrm{Number}$
        $join(a\colon \mathrm{Number}, b\colon \mathrm{Number})\colon \mathrm{Number}$
end
opr $\sum[\![T]\!](g : (\mathrm{Reduction}[\![\mathrm{Number}]\!], T \to \mathrm{Number}) \to \mathrm{Number})\colon \mathrm{Number}$
object ProdReduction extends Reduction[\![Number]\!]
        getter $toString()$
        $empty()\colon \mathrm{Number}$
        $join(a : \mathrm{Number}, b : \mathrm{Number})\colon \mathrm{Number}$
end
opr $\prod[\![T]\!](g : (\mathrm{Reduction}[\![\mathrm{Number}]\!], T \to \mathrm{Number}) \to \mathrm{Number})\colon \mathrm{Number}$
```

Hack to permit both Number and TotalOrder to work.

```
object MinReduction extends Reduction⟦Any⟧
      getter toString()
      empty(): Any
      join(a: Any, b: Any): Any
end
```

Again, type information is notoriously non-specific to permit either TotalOrder or Number types.

```
opr BIG MIN ⟦T⟧(g : (Reduction⟦Any⟧, T → Any) → Any): Any
object NoMax extends UncheckedException end
```

Hack to permit both Number and TotalOrder to work.

```
object MaxReduction extends Reduction⟦Any⟧
      getter toString()
      empty(): Any
      join(a: Any, b: Any): Any
end
opr BIG MAX ⟦T⟧(g : (Reduction⟦Any⟧, T → Any) → Any): Any
```

AndReduction and OrReduction take advantage of natural zeroes for early exit.

```
object AndReduction extends Reduction⟦Boolean⟧
      getter toString()
      empty(): Boolean
      join(a: Boolean, b: Boolean): Boolean
end
opr BIG ∧ ⟦T⟧(g : (Reduction⟦Boolean⟧, T → Boolean) → Boolean) : Boolean
object OrReduction extends Reduction⟦Boolean⟧
      getter toString()
      empty(): Boolean
      join(a: Boolean, b: Boolean): Boolean
end
opr BIG ∨ ⟦T⟧(g : (Reduction⟦Boolean⟧, T → Boolean) → Boolean) : Boolean
object StringReduction extends Reduction⟦String⟧
      getter toString()
      empty(): Boolean
      join(a : String, b : String): String
end
opr BIG STRING (g : (Reduction⟦String⟧, Any → String) → String): String
```

## Ranges

Ranges in general represent uses of the `#` and `:` operators. It is mostly subtypes of Range that are interesting.

The partial order on ranges describes containment: $a < b$ if and only if all points in $a$ are strictly contained in $b$.

```
trait Range⟦T⟧
      extends StandardPartialOrder⟦Range⟦T⟧⟧
      comprises { RangeWithLower⟦T⟧, RangeWithUpper⟦T⟧,
                  RangeWithExtent⟦T⟧, PartialRange⟦T⟧ }
```

```
    excludes { Number }
    opr =(self, _: Range⟦T⟧): Boolean
end
trait PartialRange⟦T⟧ extends Range⟦T⟧
    comprises { OpenRange⟦T⟧,
                LowerRange⟦T⟧, UpperRange⟦T⟧, ExtentRange⟦T⟧ }
    excludes { FullRange⟦T⟧ }
end
object OpenRange⟦T⟧ extends { Range⟦T⟧, PartialRange⟦T⟧ }
    toString(): String
    opr =(self, _: OpenRange⟦T⟧): Boolean
    opr CMP(self, other: Range⟦T⟧): Comparison
end
opr PARTIAL_LEXICO(a: Comparison, b: Comparison)
opr PARTIAL_LEXICO(a: Comparison, b: () → Comparison)
trait RangeWithLower⟦T⟧ extends Range⟦T⟧
        comprises { LowerRange⟦T⟧, FullRange⟦T⟧ }
    abstract getter lower(): T
end
object LowerRange⟦T⟧(lo: T) extends { RangeWithLower⟦T⟧, PartialRange⟦T⟧ }
    getter lower(): T
    toString(): String
    opr =(self, x: LowerRange⟦T⟧): Boolean
    opr CMP(self, other: Range⟦T⟧): Comparison
end
trait RangeWithUpper⟦T⟧ extends Range⟦T⟧
        comprises { UpperRange⟦T⟧, FullRange⟦T⟧ }
    abstract getter upper(): T
end
object UpperRange⟦T⟧(up: T) extends { RangeWithUpper⟦T⟧, PartialRange⟦T⟧ }
    getter upper(): T
    toString(): String
    opr =(self, x: UpperRange⟦T⟧): Boolean
    opr CMP(self, other: Range⟦T⟧): Comparison
end
trait RangeWithExtent⟦T⟧ extends Range⟦T⟧
        comprises { ExtentRange⟦T⟧, FullRange⟦T⟧ }
    abstract getter extent(): T
    toString(): String
end
object ExtentRange⟦T⟧(ex: T) extends { RangeWithExtent⟦T⟧, PartialRange⟦T⟧ }
    getter extent(): T
    opr =(self, x: ExtentRange⟦T⟧): Boolean
    opr CMP(self, other: Range⟦T⟧): Comparison
end
trait FullRange⟦T⟧
        extends { RangeWithLower⟦T⟧, RangeWithUpper⟦T⟧,
                  RangeWithExtent⟦T⟧, Indexed⟦T, T⟧ }
```

    `comprises` $\{\ldots\}$
  `getter` $\mathit{indices}()\colon \mathrm{FullRange}[\![T]\!]$

  `opr` $[r : \mathrm{Range}[\![T]\!]]\colon \mathrm{FullRange}[\![T]\!]$
  `opr` $[\_ : \mathrm{OpenRange}[\![T]\!]]\colon \mathrm{FullRange}[\![T]\!]$

Square-bracket indexing on a FullRange restricts that range to the range provided. Restriction should behave as follows:

- Restriction to an OpenRange is the identity.
- An UpperRange or ExtentRange restrict the upper bound and extent of the range.
- A LowerRange restricts the lower bound and extent of the range.

Note that this makes it compatible with the square-bracket indexing of the Indexed trait.

  `opr` $[r : \mathrm{LowerRange}[\![T]\!]]\colon \mathrm{FullRange}[\![T]\!]$
  `opr` $[r : \mathrm{UpperRange}[\![T]\!]]\colon \mathrm{FullRange}[\![T]\!]$
  `opr` $[r : \mathrm{ExtentRange}[\![T]\!]]\colon \mathrm{FullRange}[\![T]\!]$
  `opr` $[r : \mathrm{FullRange}[\![T]\!]]\colon \mathrm{FullRange}[\![T]\!]$

  $\mathit{toString}() : \mathrm{String}$

  `opr` $=($`self`$, \mathit{other} : \mathrm{FullRange}[\![T]\!])\colon \mathrm{Boolean}$
  `opr CMP`$($`self`$, \mathit{other} : \mathrm{Range}[\![T]\!])\colon \mathrm{Comparison}$
`end`

The # and : operators serve as factories for parallel ranges.

`opr #` $[\![I$ `extends` $\mathrm{Integral}]\!](\mathit{lo} : I, \mathit{ex} : I)\colon \mathrm{Range}[\![I]\!]$
`opr #` $(\mathit{lo} : \mathrm{IntLiteral}, \mathit{ex} : \mathrm{IntLiteral})\colon \mathrm{Range}[\![\mathbb{Z}32]\!]$
`opr #` $[\![I$ `extends` $\mathrm{Integral}, J$ `extends` $\mathrm{Integral}]\!]$
    $(\mathit{lo} : (I, J), \mathit{ex} : (I, J))\colon \mathrm{Range}[\![(I, J)]\!]$
`opr #` $[\![I$ `extends` $\mathrm{Integral}, J$ `extends` $\mathrm{Integral}, K$ `extends` $\mathrm{Integral}]\!]$
    $(\mathit{lo} : (I, J, K), \mathit{ex} : (I, J, K))\colon \mathrm{Range}[\![(I, J, K)]\!]$
`opr:` $[\![I$ `extends` $\mathrm{Integral}]\!](\mathit{lo} : I, \mathit{hi} : I)\colon \mathrm{FullRange}[\![I]\!]$
`opr:` $(\mathit{lo} : \mathrm{IntLiteral}, \mathit{ex} : \mathrm{IntLiteral})\colon \mathrm{FullRange}[\![\mathbb{Z}32]\!]$
`opr:` $[\![I$ `extends` $\mathrm{Integral}, J$ `extends` $\mathrm{Integral}]\!]$
    $(\mathit{lo} : (I, J), \mathit{hi} : (I, J))\colon \mathrm{Range}[\![(I, J)]\!]$
`opr:` $[\![I$ `extends` $\mathrm{Integral}, J$ `extends` $\mathrm{Integral}, K$ `extends` $\mathrm{Integral}]\!]$
    $(\mathit{lo} : (I, J, K), \mathit{hi} : (I, J, K))\colon \mathrm{Range}[\![(I, J, K)]\!]$

Factories for incomplete ranges.

`opr` $(x : T)$ `#` $[\![T]\!]$ : $\mathrm{LowerRange}[\![T]\!]$
`opr` $(x : T)$ `:` $[\![T]\!]$ : $\mathrm{LowerRange}[\![T]\!]$
`opr #` $[\![T]\!](x : T)$ : $\mathrm{ExtentRange}[\![T]\!]$
`opr:` $[\![T]\!](x : T)$ : $\mathrm{UpperRange}[\![T]\!]$

`opr #` $()\colon \mathrm{OpenRange}[\![\mathrm{Any}]\!]$
`opr:` $()\colon \mathrm{OpenRange}[\![\mathrm{Any}]\!]$

### 24.1.2 Top-level primitives

`opr` $|[\![\, N$ `extends` $\mathrm{Integral}\,]\!] x : N|$

`opr` $-(a : \mathbb{Z}32) : \mathbb{Z}32$

`opr` $+(a : \mathbb{Z}32, b : \mathbb{Z}32) : \mathbb{Z}32$
`opr` $-(a : \mathbb{Z}32, b : \mathbb{Z}32) : \mathbb{Z}32$
`opr` $\cdot(a : \mathbb{Z}32, b : \mathbb{Z}32) : \mathbb{Z}32$
`opr juxtaposition`
    $(a : \mathbb{Z}32, b : \mathbb{Z}32) : \mathbb{Z}32$
`opr` $\div(a : \mathbb{Z}32, b : \mathbb{Z}32) : \mathbb{Z}32$
`opr REM`$(a : \mathbb{Z}32, b : \mathbb{Z}32) : \mathbb{Z}32$
`opr MOD`$(a : \mathbb{Z}32, b : \mathbb{Z}32) : \mathbb{Z}32$
`opr GCD`$(a : \mathbb{Z}32, b : \mathbb{Z}32) : \mathbb{Z}32$
`opr LCM`$(a : \mathbb{Z}32, b : \mathbb{Z}32) : \mathbb{Z}32$
`opr CHOOSE`$(a : \mathbb{Z}32, b : \mathbb{Z}32) : \mathbb{Z}32$
`opr` $⩓(a : \mathbb{Z}32, b : \mathbb{Z}32) : \mathbb{Z}32$
`opr` $⩔(a : \mathbb{Z}32, b : \mathbb{Z}32) : \mathbb{Z}32$
`opr` $\underline{⩔}(a : \mathbb{Z}32, b : \mathbb{Z}32) : \mathbb{Z}32$
`opr LSHIFT`$(a : \mathbb{Z}32, b : \text{Integral}) : \mathbb{Z}32$
`opr RSHIFT`$(a : \mathbb{Z}32, b : \text{Integral}) : \mathbb{Z}32$
`opr` $⫬(a : \mathbb{Z}32) : \mathbb{Z}32$
`opr` $=(a : \mathbb{Z}32, b : \mathbb{Z}32) : \text{Boolean}$
`opr` $\leq(a : \mathbb{Z}32, b : \mathbb{Z}32) : \text{Boolean}$
`opr` $\hat{}\,(a : \mathbb{Z}32, b : \text{Integral}) : \text{Number}$

$\mathit{widen}$ converts a $\mathbb{Z}32$ into a valid $\mathbb{Z}64$ quantity.
$\mathit{widen}(a : \mathbb{Z}32) : \mathbb{Z}64$

$\mathit{partitionL}$ returns the highest power of $2 < a$, used to partition iteration spaces for arrays and ranges.
$\mathit{partitionL}(a : \mathbb{Z}32) : \mathbb{Z}32$

$\mathit{nanoTime}$ returns the current time in nanoseconds. Currently, this only supports taking differences of results of $\mathit{nanoTime}$ to produce an elapsed time interval.
$\mathit{nanoTime}() : \mathbb{Z}64$

$\mathit{printTaskTrace}$ dumps some internal error state.
$\mathit{printTaskTrace}() : ()$

$\mathit{recordTime}(\mathit{dummy}\colon \text{Any})\colon ()$
$\mathit{printTime}(\mathit{dummy}\colon \text{Any})\colon ()$

`opr` $-(a\colon \text{IntLiteral})\colon \text{IntLiteral}$
`opr` $+(a\colon \text{IntLiteral}, b\colon \text{IntLiteral})\colon \text{IntLiteral}$
`opr` $-(a\colon \text{IntLiteral}, b\colon \text{IntLiteral})\colon \text{IntLiteral}$
`opr` $\cdot(a\colon \text{IntLiteral}, b\colon \text{IntLiteral})\colon \text{IntLiteral}$
`opr juxtaposition`
    $(a\colon \text{IntLiteral}, b\colon \text{IntLiteral})\colon \text{IntLiteral}$
`opr` $\div(a\colon \text{IntLiteral}, b\colon \text{IntLiteral})\colon \text{IntLiteral}$
`opr REM`$(a\colon \text{IntLiteral}, b\colon \text{IntLiteral})\colon \text{IntLiteral}$
`opr MOD`$(a\colon \text{IntLiteral}, b\colon \text{IntLiteral})\colon \text{IntLiteral}$
`opr GCD`$(a\colon \text{IntLiteral}, b\colon \text{IntLiteral})\colon \text{IntLiteral}$
`opr LCM`$(a\colon \text{IntLiteral}, b\colon \text{IntLiteral})\colon \text{IntLiteral}$
`opr CHOOSE`$(a\colon \text{IntLiteral}, b\colon \text{IntLiteral})\colon \text{IntLiteral}$
`opr` $⩓(a\colon \text{IntLiteral}, b\colon \text{IntLiteral})\colon \text{IntLiteral}$
`opr` $⩔(a\colon \text{IntLiteral}, b\colon \text{IntLiteral})\colon \text{IntLiteral}$

$\mathtt{opr}\ \underline{⩔}(a\colon \text{IntLiteral}, b\colon \text{IntLiteral})\colon \text{IntLiteral}$
$\mathtt{opr}\ \mathtt{LSHIFT}(a\colon \text{IntLiteral}, b : \text{Integral})\colon \text{IntLiteral}$
$\mathtt{opr}\ \mathtt{RSHIFT}(a\colon \text{IntLiteral}, b : \text{Integral})\colon \text{IntLiteral}$
$\mathtt{opr}\ ⫬(a\colon \text{IntLiteral})\colon \text{IntLiteral}$
$\mathtt{opr}\ =(a\colon \text{IntLiteral}, b\colon \text{IntLiteral}) : \text{Boolean}$
$\mathtt{opr}\ \leq(a\colon \text{IntLiteral}, b\colon \text{IntLiteral}) : \text{Boolean}$
$\mathtt{opr}\ \hat{}\ (a\colon \text{IntLiteral}, b : \text{Integral}) : \text{Number}$

$\mathtt{opr}\ -[\![ T\ \mathtt{extends}\ \text{Number}, \mathtt{nat}\ n, \mathtt{nat}\ m ]\!]$
$\quad (a : \text{Integral}) : \mathbb{Z}64$
$\mathtt{opr}\ +[\![ T\ \mathtt{extends}\ \text{Number}, \mathtt{nat}\ n, \mathtt{nat}\ m ]\!]$
$\quad (a : \text{Integral}, b : \text{Integral}) : \mathbb{Z}64$
$\mathtt{opr}\ -[\![ T\ \mathtt{extends}\ \text{Number}, \mathtt{nat}\ n, \mathtt{nat}\ m ]\!]$
$\quad (a : \text{Integral}, b : \text{Integral}) : \mathbb{Z}64$
$\mathtt{opr}\ \cdot[\![ T\ \mathtt{extends}\ \text{Number}, \mathtt{nat}\ n, \mathtt{nat}\ m, \mathtt{nat}\ p ]\!]$
$\quad (a : \text{Integral}, b : \text{Integral}) : \mathbb{Z}64$
$\mathtt{opr\ juxtaposition}\ [\![ T\ \mathtt{extends}\ \text{Number}, \mathtt{nat}\ n, \mathtt{nat}\ m, \mathtt{nat}\ p ]\!]$
$\quad (a : \text{Integral}, b : \text{Integral}) : \mathbb{Z}64$
$\mathtt{opr}\ \div(a : \text{Integral}, b : \text{Integral}) : \mathbb{Z}64$
$\mathtt{opr}\ \mathtt{REM}(a : \text{Integral}, b : \text{Integral}) : \mathbb{Z}64$
$\mathtt{opr}\ \mathtt{MOD}(a : \text{Integral}, b : \text{Integral}) : \mathbb{Z}64$
$\mathtt{opr}\ \mathtt{GCD}(a : \text{Integral}, b : \text{Integral}) : \mathbb{Z}64$
$\mathtt{opr}\ \mathtt{LCM}(a : \text{Integral}, b : \text{Integral}) : \mathbb{Z}64$
$\mathtt{opr}\ \mathtt{CHOOSE}(a : \text{Integral}, b : \text{Integral}) : \mathbb{Z}64$
$\mathtt{opr}\ ⩓(a : \text{Integral}, b : \text{Integral}) : \mathbb{Z}64$
$\mathtt{opr}\ ⩔(a : \text{Integral}, b : \text{Integral}) : \mathbb{Z}64$
$\mathtt{opr}\ \underline{⩔}(a : \text{Integral}, b : \text{Integral}) : \mathbb{Z}64$
$\mathtt{opr}\ \mathtt{LSHIFT}(a : \text{Integral}, b : \text{Integral}) : \mathbb{Z}64$
$\mathtt{opr}\ \mathtt{RSHIFT}(a : \text{Integral}, b : \text{Integral}) : \mathbb{Z}64$
$\mathtt{opr}\ ⫬(a : \text{Integral}) : \mathbb{Z}64$
$\mathtt{opr}\ =(a : \text{Integral}, b : \text{Integral}) : \text{Boolean}$
$\mathtt{opr}\ \leq(a : \text{Integral}, b : \text{Integral}) : \text{Boolean}$
$\mathtt{opr}\ \hat{}\ (a : \mathbb{Z}64, b : \text{Integral}) : \text{Number}$
$\mathit{narrow}(a : \mathbb{Z}64) : \mathbb{Z}32$

$\mathtt{opr}\ <(a : \text{Integral}, b : \text{Integral}) : \text{Boolean}$
$\mathtt{opr}\ >(a : \text{Integral}, b : \text{Integral}) : \text{Boolean}$
$\mathtt{opr}\ \geq(a : \text{Integral}, b : \text{Integral}) : \text{Boolean}$
$\mathtt{opr}\ \mathtt{CMP}(a : \text{Integral}, b : \text{Integral}) : \text{TotalComparison}$
$\mathtt{opr}\ \mathtt{MIN}[\![ I\ \mathtt{extends}\ \text{Integral} ]\!](a : I, b : I) : I$
$\mathtt{opr}\ \mathtt{MAX}[\![ I\ \mathtt{extends}\ \text{Integral} ]\!](a : I, b : I) : I$

$\mathtt{opr}\ -(a : \mathbb{R}64) : \mathbb{R}64$
$\mathtt{opr}\ +(a : \text{Number}, b : \text{Number}) : \mathbb{R}64$
$\mathtt{opr}\ -(a : \text{Number}, b : \text{Number}) : \mathbb{R}64$
$\mathtt{opr}\ \cdot(a : \text{Number}, b : \text{Number}) : \mathbb{R}64$
$\mathtt{opr\ juxtaposition}$
$\quad (a : \text{Number}, b : \text{Number}) : \mathbb{R}64$
$\mathtt{opr}\ /(a : \text{Number}, b : \text{Number}) : \mathbb{R}64$
$\mathtt{opr}\ =(a : \text{Number}, b : \text{Number}) : \text{Boolean}$
$\mathtt{opr}\ \neq(a : \text{Number}, b : \text{Number}) : \text{Boolean}$
$\mathtt{opr}\ <(a : \text{Number}, b : \text{Number}) : \text{Boolean}$
$\mathtt{opr}\ \leq(a : \text{Number}, b : \text{Number}) : \text{Boolean}$
$\mathtt{opr}\ >(a : \text{Number}, b : \text{Number}) : \text{Boolean}$

`opr` $\geq(a : \text{Number}, b : \text{Number}) : \text{Boolean}$
`opr` `CMP`$(a : \text{Number}, b : \text{Number}) : \text{Comparison}$
`opr` `MIN`$(a : \text{Number}, b : \text{Number}) : \text{Boolean}$
`opr` `MAX`$(a : \text{Number}, b : \text{Number}) : \text{Boolean}$
`opr` $|a : \mathbb{R}64| : \mathbb{R}64$
`opr` $\hat{}\,(a : \text{Number}, b : \text{Number}) : \mathbb{R}64$
`opr` $\surd(a : \text{Number}) : \mathbb{R}64$
$\sin(a : \text{Number}) : \mathbb{R}64$
$\cos(a : \text{Number}) : \mathbb{R}64$
$\tan(a : \text{Number}) : \mathbb{R}64$
$asin(a : \text{Number}) : \mathbb{R}64$
$acos(a : \text{Number}) : \mathbb{R}64$
$atan(a : \text{Number}) : \mathbb{R}64$
$atan_2(y : \text{Number}, x : \text{Number}) : \mathbb{R}64$
$\log(a : \text{Number}) : \mathbb{R}64$
$e(a : \text{Number}) : \mathbb{R}64$
$floor(a : \text{Number}) : \mathbb{R}64$
`opr` $\lfloor a : \text{Number} \rfloor : \mathbb{Z}64$
$ceiling(a : \text{Number}) : \mathbb{R}64$
`opr` $\lceil a : \text{Number} \rceil : \mathbb{Z}64$
$truncate(a : \text{Number}) : \mathbb{Z}64$
$random(a : \text{Number}) : \mathbb{R}64$

`opr` $=(a : \text{Char}, b : \text{Char}) : \text{Boolean}$

`opr` $\cdot(a : \text{String}, b : \text{String}) : \text{String}$
`opr juxtaposition`
        $(a : \text{String}, b : \text{String}) : \text{String}$
`opr` $\cdot(a : \text{Number}, b : \text{String}) : \text{String}$
`opr juxtaposition`
        $(a : \text{Number}, b : \text{String}) : \text{String}$
`opr` $\cdot(a : \text{String}, b : \text{Number}) : \text{String}$
`opr juxtaposition`
        $(a : \text{String}, b : \text{Number}) : \text{String}$
`opr` $\cdot(a : \text{Boolean}, b : \text{String}) : \text{String}$
`opr juxtaposition`
        $(a : \text{Boolean}, b : \text{String}) : \text{String}$
`opr` $\cdot(a : \text{String}, b : \text{Boolean}) : \text{String}$
`opr juxtaposition`
        $(a : \text{String}, b : \text{Boolean}) : \text{String}$
`opr` $\cdot(a : \text{String}, c : \text{Char}) : \text{String}$
`opr juxtaposition`
        $(a : \text{String}, c : \text{Char}) : \text{String}$
`opr` $\cdot(c : \text{Char}, a : \text{String}) : \text{String}$
`opr juxtaposition`
        $(c : \text{Char}, a : \text{String}) : \text{String}$
`opr` $\cdot(a : \text{String}, b : ()) : \text{String}$
`opr juxtaposition` $(a : \text{String}, b : ()) : \text{String}$
`opr` $\cdot(a : \text{String}, b : (\text{Any}, \text{Any})) : \text{String}$
`opr juxtaposition` $(a : \text{String}, b : (\text{Any}, \text{Any})) : \text{String}$
`opr juxtaposition` $(a : \text{String}, b : (\text{Any}, \text{Any}, \text{Any})) : \text{String}$
`opr` $\cdot(a : (), b : \text{String}) : \text{String}$
`opr juxtaposition` $(a : (), b : \text{String}) : \text{String}$

`opr` $\cdot(a\,{:}\,\mathrm{Any}, b\,{:}\,\mathrm{String})\,{:}\,\mathrm{String}$
`opr juxtaposition` $(a\,{:}\,\mathrm{Any}, b\,{:}\,\mathrm{String})\,{:}\,\mathrm{String}$
`opr` $\cdot(a\,{:}\,\mathrm{String}, b\,{:}\,\mathrm{Any})\,{:}\,\mathrm{String}$
`opr juxtaposition` $(a\,{:}\,\mathrm{String}, b\,{:}\,\mathrm{Any})\,{:}\,\mathrm{String}$

`opr` $=(a\,{:}\,\mathrm{String}, b\,{:}\,\mathrm{String})\,{:}\,\mathrm{Boolean}$
`opr` $<(a\,{:}\,\mathrm{String}, b\,{:}\,\mathrm{String})\,{:}\,\mathrm{Boolean}$
`opr` $\leq(a\,{:}\,\mathrm{String}, b\,{:}\,\mathrm{String})\,{:}\,\mathrm{Boolean}$
`opr` $>(a\,{:}\,\mathrm{String}, b\,{:}\,\mathrm{String})\,{:}\,\mathrm{Boolean}$
`opr` $\geq(a\,{:}\,\mathrm{String}, b\,{:}\,\mathrm{String})\,{:}\,\mathrm{Boolean}$
`opr CMP`$(a\,{:}\,\mathrm{String}, b\,{:}\,\mathrm{String})\,{:}\,\mathrm{TotalComparison}$

$\mathit{outFileOpen}(\mathit{name}\,{:}\,\mathrm{String})\,{:}\,\mathrm{BufferedWriter}$
$\mathit{outFileWrite}(\mathit{file}{:}\,\mathrm{BufferedWriter}, \mathit{str}{:}\,\mathrm{String})\,{:}\,()$
$\mathit{outFileClose}(\mathit{file}{:}\,\mathrm{BufferedWriter})\,{:}\,()$

$\mathit{substring}(\mathit{str}{:}\,\mathrm{String}, \mathit{beginIndex}{:}\,\mathbb{Z}32, \mathit{endIndex}{:}\,\mathbb{Z}32)\,{:}\,\mathrm{String}$
$\mathit{length}(\mathit{str}{:}\,\mathrm{String})\,{:}\,\mathbb{Z}32$

$\mathit{print}(a\,{:}\,\mathrm{String})\,{:}\,()$
$\mathit{println}(a\,{:}\,\mathrm{String})\,{:}\,()$
$\mathit{print}(a\,{:}\,\mathrm{Number})\,{:}\,()$
$\mathit{println}(a\,{:}\,\mathrm{Number})\,{:}\,()$
$\mathit{print}(a\,{:}\,\mathrm{Boolean})\,{:}\,()$
$\mathit{println}(a\,{:}\,\mathrm{Boolean})\,{:}\,()$
$\mathit{println}(a\,{:}\,\mathrm{Char})\,{:}\,()$
$\mathit{print}(a\,{:}\,\mathrm{Any})\,{:}\,()$
$\mathit{println}(a\,{:}\,\mathrm{Any})\,{:}\,()$

0-argument versions handle passing of () to single-argument versions.
$\mathit{print}()\,{:}\,()$
$\mathit{println}()\,{:}\,()$

The following three functions are useful temporary hacks for debugging multi-threaded programs.
$\mathit{printThreadInfo}(a\,{:}\,\mathrm{String})\,{:}\,()$
$\mathit{printThreadInfo}(a\,{:}\,\mathrm{Number})\,{:}\,()$
$\mathit{throwError}(a\,{:}\,\mathrm{String})\,{:}\,()$

`opr SEQV`$(a\,{:}\,\mathrm{Any}, b\,{:}\,\mathrm{Any})\,{:}\,\mathrm{Boolean}$

`opr` $\vee(a\,{:}\,\mathrm{Boolean}, b\,{:}\,\mathrm{Boolean})\,{:}\,\mathrm{Boolean}$
`opr` $\wedge(a\,{:}\,\mathrm{Boolean}, b\,{:}\,\mathrm{Boolean})\,{:}\,\mathrm{Boolean}$
`opr` $\vee(a\,{:}\,\mathrm{Boolean}, b\,{:}\,() \to \mathrm{Boolean})\,{:}\,\mathrm{Boolean}$
`opr` $\wedge(a\,{:}\,\mathrm{Boolean}, b\,{:}\,() \to \mathrm{Boolean})\,{:}\,\mathrm{Boolean}$
`opr` $\neg(a\,{:}\,\mathrm{Boolean})\,{:}\,\mathrm{Boolean}$
`opr` $\rightarrow(a{:}\,\mathrm{Boolean}, b\,{:}\,\mathrm{Boolean})\,{:}\,\mathrm{Boolean}$
`opr` $\rightarrow(a{:}\,\mathrm{Boolean}, b\,{:}\,() \to \mathrm{Boolean})\,{:}\,\mathrm{Boolean}$
`opr` $\leftrightarrow(a{:}\,\mathrm{Boolean}, b\,{:}\,\mathrm{Boolean})\,{:}\,\mathrm{Boolean}$

$\mathit{true}$ : Boolean
$\mathit{false}$ : Boolean

`opr` $+[\![T$ `extends` $\mathrm{Number}]\!](x\,{:}\,T)\,{:}\,T$

`opr` $=[\![A, B]\!](t_1\,{:}\,(A, B), t_2\,{:}\,(A, B)){:}\,\mathrm{Boolean}$
`opr` $<[\![A, B]\!](t_1\,{:}\,(A, B), t_2\,{:}\,(A, B)){:}\,\mathrm{Boolean}$
`opr` $\leq[\![A, B]\!](t_1\,{:}\,(A, B), t_2\,{:}\,(A, B)){:}\,\mathrm{Boolean}$

opr $>[\![A,B]\!](t_1 : (A,B), t_2 : (A,B))$: Boolean
opr $\geq[\![A,B]\!](t_1 : (A,B), t_2 : (A,B))$: Boolean
opr CMP$[\![A,B]\!](t_1 : (A,B), t_2 : (A,B))$: Boolean
opr $=[\![A,B,C]\!](t_1 : (A,B,C), t_2 : (A,B,C))$: Boolean
opr $<[\![A,B,C]\!](t_1 : (A,B,C), t_2 : (A,B,C))$: Boolean
opr $\leq[\![A,B,C]\!](t_1 : (A,B,C), t_2 : (A,B,C))$: Boolean
opr $>[\![A,B,C]\!](t_1 : (A,B,C), t_2 : (A,B,C))$: Boolean
opr $\geq[\![A,B,C]\!](t_1 : (A,B,C), t_2 : (A,B,C))$: Boolean
opr CMP$[\![A,B,C]\!](t_1 : (A,B,C), t_2 : (A,B,C))$: Boolean
end

## 24.2 Builtins

At present, there are two libraries containing builtins: FortressBuiltin (Section 24.2.1) and NativeSimpleTypes (Section 24.2.2). This reflects an artifact of the current implementation state; the intention is to eventually provide a single library called FortressBuiltin. Every Fortress API and component will unconditionally see the names exported by these builtin APIs.

### 24.2.1 FortressBuiltin

```
api FortressBuiltin
(* import NativeSimpleTypes.{Boolean} *)
trait Any end
trait String extends { Any }
      (* excludes { IntLiteral, FloatLiteral, Boolean } *)
end
trait Char extends { Any }
end
trait Number extends { Any }
      (* excludes { String, Boolean } *)
end
trait Integral extends { Number }
      (* excludes { String, Boolean, RR64, FloatLiteral } *)
end
trait BufferedWriter extends { Any } end
trait ℤ32 extends { Integral }
      (* excludes { String, Boolean, RR64, FloatLiteral } *)
end
trait ℤ64 extends { Integral }
      (* excludes { String, Boolean, RR64, FloatLiteral } *)
end
trait ℝ64 extends { Number }
      (* excludes { String, Boolean } *)
end
trait IntLiteral extends { ℤ32, ℤ64, ℝ64 } end
trait FloatLiteral extends { ℝ64 } end
end
```

### 24.2.2 NativeSimpleTypes

```
api NativeSimpleTypes
object Boolean
     extends { SequentialGenerator⟦()⟧, StandardTotalOrder⟦Boolean⟧ }
end
object Thread⟦T⟧(fcn : () → T)
     getter val() : T
     getter ready() : Boolean
     wait() : ()
     stop() : ()
end
abort() : ()
end
```

# Chapter 25

# Additional Libraries Available to Fortress Programmers

The libraries described in this chapter must be explicitly imported into Fortress programs.

## 25.1 Constants

```
api Constants
```

Useful floating-point constants.
$\pi :$ FloatLiteral
$e :$ FloatLiteral
$\mathit{infinity} : \mathbb{R}64$
`end`

## 25.2 List

```
api List
```

Array lists, immutable style (not the mutable Java `ArrayList` style).

A List is an immutable segment of an immutable (really write-once) array. The rest of the array may contain elements of lists which overlap this list, or may be free for future use. Every List includes two internal flags *canExtendLeft* and *canExtendRight*; if a flag is true we are permitted to add additional elements to the List in place by initializing additional elements of the array. At most one instance sharing the same backing array will obtain permission to extend the array in this way; we atomically check and update the flag to guarantee this. Having obtained permission to extend the list, that permission may be extended to future attempts to extend.

Eventually the backing array fills and we must allocate a new backing array to accept new elements. At the moment, we are not particularly careful to avoid stealing permission to extend for overflowing *append* operations.

Note that because of this implementation, a List can be efficiently extended on either side, but only in a non-persistent way; if a single list is extended by two different calls to *addRight* or *append* then one of them must pay the cost of copying the list elements.

Note also that the implementation has not yet been carefully checked for amortization, so it is quite likely there are a number of asymptotic infelicities.

Finally, note that this is an efficient *amortized* structure. An individual operation may be quite slow due to copying work.

Baking these off vs PureLists (which have good persistent behavior and non-amortized worst case behavior), they look very good in practice.

Lists of some item type. Used to collect elements of unknown type into a list whose element type is as specific as possible. This should not be necessary in the presence of true type inference.

```
trait SomeList excludes { Number, HasRank }
          Not yet: “ comprises List⟦E⟧ where ⟦E⟧ ”
      append(f : SomeList): SomeList
      addLeft(e : Any): SomeList
      addRight(e : Any): SomeList
end
```

List. We return a Generator for non-list-specific operations for which reuse of the Generator will not increase asymptotic complexity, but return a List in cases (such as *map* and *filter*) where it will.

```
trait List⟦E⟧ extends { Equality⟦E⟧, ZeroIndexed⟦E⟧ }
          excludes { Number, HasRank }
   getter left() : Maybe⟦E⟧
   getter right() : Maybe⟦E⟧
   getter extractLeft(): Maybe⟦(E, List⟦E⟧)⟧
   getter extractRight(): Maybe⟦(List⟦E⟧, E)⟧
   append(f : List⟦E⟧): List⟦E⟧
   addLeft(e : E) : List⟦E⟧
   addRight(e : E) : List⟦E⟧
   take(n : ℤ32): List⟦E⟧
   drop(n : ℤ32): List⟦E⟧
   split(n : ℤ32): (List⟦E⟧, List⟦E⟧)
   split(): (List⟦E⟧, List⟦E⟧)
   reverse(): List⟦E⟧
   zip⟦F⟧(other: List⟦F⟧): Generator⟦(E, F)⟧
   filter(p: E → Boolean): List⟦E⟧
   toString() : String
   concatMap⟦G⟧(f: E → List⟦G⟧): List⟦G⟧
end
```

Vararg factory for lists; provides aggregate list constants:

```
opr ⟨⟦E⟧xs: E . . . ⟩: List⟦E⟧
```

List comprehensions:

```
opr BIG⟨⟦T, U⟧g: (Reduction⟦SomeList⟧, T → SomeList) → SomeList ⟩: List⟦U⟧
```

Convert generator into list (simpler type than comprehension above):

```
list⟦E⟧(g : Generator⟦E⟧) : List⟦E⟧
```

Flatten a list of lists

$concat⟦E⟧(x : \text{List}⟦\text{List}⟦E⟧⟧) : \text{List}⟦E⟧$

$emptyList⟦E⟧(): \text{List}⟦E⟧$

$emptyList⟦E⟧(n)$ allocates an empty list that can accept $n$ $addRight$ operations without reallocating the underlying storage.

$emptyList⟦E⟧(n : \mathbb{Z}32): \text{List}⟦E⟧$

$singleton⟦E⟧(e : E): \text{List}⟦E⟧$

A reduction object for concatenating lists.

```
object Concat⟦E⟧ extends Reduction⟦ List⟦E⟧ ⟧
   empty(): List⟦E⟧
   join(a : List⟦E⟧, b : List⟦E⟧): List⟦E⟧
end
```

Covariant Singleton function, for use with CVConcat:

$cvSingleton(e : \text{Any}): \text{SomeList}$

A reduction object for concatenating SomeLists covariantly.

```
object CVConcat extends Reduction⟦SomeList⟧
   empty(): SomeList
   join(a : SomeList, b : SomeList): SomeList
end

end
```

## 25.3 PureList

```
api PureList
import List.{SomeList}
```

Finger trees, based on Ralf Hinze and Ross Paterson's article, Journal of Funtional Programming 16:2 2006 [9].

These are API-compatible with the List library, except that they do not support covariant construction. In most cases, you should be able to replace an import of List by PureList or vice versa and see only performance differences between the two.

Why finger trees? They are balanced and support nearly any operation we care to think of in optimal asymptotic time and space. The code is niggly due to lots of cases, but fast in practice.

It is also a trial for encoding type-based invariants in Fortress. Can we represent "array of size at most $n$"? Not yet, but we ought to be able to do so. This involves questions about the encoding of existentials, especially constrained existentials. If you are curious about the details of type-based invariants, the source code may prove instructive.

List. We return a Generator for non-list-specific operations for which reuse of the Generator will not increase asymptotic complexity, but return a List in cases (such as *map* and *filter*) where it will.

```
trait List⟦E⟧ extends { Equality⟦E⟧, ZeroIndexed⟦E⟧ }
          excludes { Number, HasRank }
   getter left() : Maybe⟦E⟧
   getter right() : Maybe⟦E⟧
```

    `getter` $\mathit{extractLeft}()\colon \mathrm{Maybe}[\![(E, \mathrm{List}[\![E]\!])]\!]$
    `getter` $\mathit{extractRight}()\colon \mathrm{Maybe}[\![(\mathrm{List}[\![E]\!], E)]\!]$
    $\mathit{append}(f : \mathrm{List}[\![E]\!])\colon \mathrm{List}[\![E]\!]$
    $\mathit{addLeft}(e : E) : \mathrm{List}[\![E]\!]$
    $\mathit{addRight}(e : E) : \mathrm{List}[\![E]\!]$
    $\mathit{take}(n : \mathbb{Z}32)\colon \mathrm{List}[\![E]\!]$
    $\mathit{drop}(n : \mathbb{Z}32)\colon \mathrm{List}[\![E]\!]$
    $\mathit{split}(n : \mathbb{Z}32)\colon (\mathrm{List}[\![E]\!], \mathrm{List}[\![E]\!])$
    $\mathit{split}()\colon (\mathrm{List}[\![E]\!], \mathrm{List}[\![E]\!])$
    $\mathit{reverse}()\colon \mathrm{List}[\![E]\!]$
    $\mathit{zip}[\![F]\!](\mathit{other}\colon \mathrm{List}[\![F]\!])\colon \mathrm{Generator}[\![(E, F)]\!]$
    $\mathit{filter}(p\colon E \to \mathrm{Boolean})\colon \mathrm{List}[\![E]\!]$
    $\mathit{toString}() : \mathrm{String}$
    $\mathit{concatMap}[\![G]\!](f\colon E \to \mathrm{List}[\![G]\!])\colon \mathrm{List}[\![G]\!]$
`end`

Vararg factory for lists; provides aggregate list constants:
`opr` $\langle[\![E]\!]xs\colon E\ldots\rangle\colon \mathrm{List}[\![E]\!]$

List comprehensions:
`opr BIG`$\langle[\![T, U]\!]g\colon (\mathrm{Reduction}[\![\mathrm{SomeList}]\!], T \to \mathrm{SomeList}) \to \mathrm{SomeList}\rangle\colon \mathrm{List}[\![U]\!]$

Convert generator into list (simpler type than comprehension above):
$\mathit{list}[\![E]\!](g : \mathrm{Generator}[\![E]\!]) : \mathrm{List}[\![E]\!]$

Flatten a list of lists.
$\mathit{concat}[\![E]\!](x : \mathrm{List}[\![\mathrm{List}[\![E]\!]]\!]) : \mathrm{List}[\![E]\!]$
$\mathit{emptyList}[\![E]\!]()\colon \mathrm{List}[\![E]\!]$
$\mathit{singleton}[\![E]\!](e : E)\colon \mathrm{List}[\![E]\!]$
`object` $\mathrm{Concat}[\![E]\!]$ `extends` $\mathrm{Reduction}[\![\,\mathrm{List}[\![E]\!]\,]\!]$
    $\mathit{empty}()\colon \mathrm{List}[\![E]\!]$
    $\mathit{join}(a : \mathrm{List}[\![E]\!], b : \mathrm{List}[\![E]\!])\colon \mathrm{List}[\![E]\!]$
`end`
`end`

## 25.4 File

`api` File
`import` FileSupport.{. . .}
`object` FileReadStream(`transient` $\mathit{filename} : \mathrm{String}$) `extends` { FileStream, ReadStream }
        `getter` $\mathit{fileName}() : \mathrm{String}$
        `getter` $\mathit{toString}() : \mathrm{String}$

        *eof* returns true if an end-of-file (EOF) condition has been encountered on the stream.
        `getter` $\mathit{eof}() : \mathrm{Boolean}$

        *ready* returns true if there is currently input from the stream available to be consumed.

`getter` $\mathit{ready}() : \mathrm{Boolean}$

close the stream.
$\mathit{close}() : ()$

$\mathit{readLine}$ returns the next available line from the stream, discarding line termination characters. Returns “” on EOF.
$\mathit{readLine}() : \mathrm{String}$

Returns the next available character from the stream, or “” on EOF.
$\mathit{readChar}() : \mathrm{Char}$

$\mathit{read}$ returns the next $k$ characters from the stream. It will block until at least one character is available, and will then return as many characters as are ready. Will return “” on end of file. If $k <= 0$ or absent a default value is chosen.
$\mathit{read}(k : \mathbb{Z}32) : \mathrm{String}$
$\mathit{read}() : \mathrm{String}$

All file generators yield file contents in parallel by default, with a natural ordering corresponding to the order data occurs in the underlying file. The file is closed if all its contents have been read.

These generators pull in multiple chunks of data (where a “chunk” is a line, a character, or a block) before it is processed. The maximum number of chunks to pull in before processing is given by the last optional argument in every case; if it is $<= 0$ or absent a default value is chosen.

It is possible to read from a $\mathrm{ReadStream}$ before invoking any of the following methods. Previously-read data is ignored, and only the remaining data is provided by the generator.

At the moment, it is illegal to read from a $\mathrm{ReadStream}$ once any of these methods has been called; we do not check for this condition. However, the sequential versions of these generators may use label/exit or throw in order to escape from a loop, and the $\mathrm{ReadStream}$ will remain open and ready for reading.

Once the $\mathrm{ReadStream}$ has been completely consumed it is closed.

$\mathit{lines}$ yields the lines found in the file a la $\mathit{readLine}$.
$\mathit{lines}(n : \mathbb{Z}32) : \mathrm{Generator}[\![\mathrm{String}]\!]$
$\mathit{lines}() : \mathrm{Generator}[\![\mathrm{String}]\!]$

$\mathit{characters}$ yields the characters found in the file a la $\mathit{readChar}$.
$\mathit{characters}(n : \mathbb{Z}32) : \mathrm{Generator}[\![\mathrm{String}]\!]$
$\mathit{characters}() : \mathrm{Generator}[\![\mathrm{String}]\!]$

$\mathit{chunks}$ returns chunks of characters found in the file, in the sense of $\mathit{read}$. The first argument is equivalent to the argument $k$ to read, the second (if present) is the number of chunks at a time.
$\mathit{chunks}(n : \mathbb{Z}32, m : \mathbb{Z}32) : \mathrm{Generator}[\![\mathrm{String}]\!]$
$\mathit{chunks}(n : \mathbb{Z}32)$: $\mathrm{Generator}[\![\mathrm{String}]\!]$
$\mathit{chunks}()$: $\mathrm{Generator}[\![\mathrm{String}]\!]$

`end`

`end`

## 25.5 Set

**api** Set
**import** List.{SomeList}

Thrown when taking big intersection of no sets.
**object** EmptyIntersection **extends** UncheckedException **end**

Sets represented using a (size-balanced) tree structure. The underlying type $E$ must support comparison using $<$ and $=$. When generated, these sets produce their elements in sorted order.

```
trait Set⟦E⟧
        extends { ZeroIndexed⟦E⟧, Equality⟦Set⟦E⟧⟧ }
        comprises { ... }
    printTree() : ()
    toString() : String
    minimum() : E
    maximum() : E
    deleteMinimum() : Set⟦E⟧
    deleteMaximum() : Set⟦E⟧
    removeMinimum() : (E, Set⟦E⟧)
    removeMaximum() : (E, Set⟦E⟧)
    add(x : E) : Set⟦E⟧
    delete(x : E) : Set⟦E⟧
    opr ∪(self, t_2 : Set⟦E⟧) : Set⟦E⟧
    opr ∩(self, t_2 : Set⟦E⟧) : Set⟦E⟧
    opr DIFFERENCE(self, t_2 : Set⟦E⟧) : Set⟦E⟧

    Symmetric difference: all elements in exactly one of the two sets.
    opr SYMDIFF(self, t_2 : Set⟦E⟧) : Set⟦E⟧
    splitAt(e : E) : (Set⟦E⟧, Boolean, Set⟦E⟧)
    opr ⊂(self, other : Set⟦E⟧): Boolean
    opr ⊆(self, other : Set⟦E⟧): Boolean
    opr ⊃(self, other : Set⟦E⟧): Boolean
    opr ⊇(self, other : Set⟦E⟧): Boolean
    opr SETCMP(self, other : Set⟦E⟧): Comparison

    Ordered concatenation; use only if you know what you are doing.
    concat(t_2 : Set⟦E⟧) : Set⟦E⟧
    concat_3(v : E, t_2 : Set⟦E⟧)
end
singleton⟦E⟧(x : E): Set⟦E⟧
set⟦E⟧(): Set⟦E⟧
set⟦E⟧(g: Generator⟦E⟧): Set⟦E⟧
opr {⟦E⟧es: E ...}: Set⟦E⟧
opr BIG{⟦T, U⟧g: (Reduction⟦SomeList⟧, T → SomeList) → SomeList } : Set⟦U⟧
opr BIG ∪ ⟦R⟧(g : (Reduction⟦Set⟦R⟧⟧, Set⟦R⟧ → Set⟦R⟧) → Set⟦R⟧) :
                                Set⟦R⟧
object Union⟦E⟧ extends Reduction⟦Set⟦E⟧⟧ end
opr BIG ∩ ⟦R⟧(g : (Reduction⟦Maybe⟦Set⟦R⟧⟧⟧,
                    Set⟦R⟧ → Maybe⟦Set⟦R⟧⟧) →
                                        Maybe⟦Set⟦R⟧⟧): Set⟦R⟧
end
```

## 25.6 Map

```
api Map
import List.{SomeList}
import Set.{Set}
object KeyOverlap⟦Key, Val⟧(key : Key, val_1 : Val, val_2 : Val)
        extends UncheckedException
    getter toString(): String
end
```

Note that the map interface is purely functional; methods return a fresh map rather than updating the receiving map in place. Methods that operate on a particular key leave the rest of the map untouched unless otherwise specified.

```
trait Map⟦Key, Val⟧
     extends { Generator⟦(Key, Val)⟧, Equality⟦Map⟦Key, Val⟧⟧ }
     comprises { . . . }
    getter isEmpty() : Boolean
    getter showTree() : String
    getter toString() : String
    dom(self) : Set⟦Key⟧
    opr |self |: ℤ32
    opr [k : Key]: Val throws NotFound
    member(x : Key): Maybe⟦Val⟧
```

The two-argument version of *member* returns the default value $v$ if the key is absent from the map.

```
    member(x : Key, v : Val): Val
```

*minimum* and *maximum* refer to the key.

```
    minimum() : (Key, Val) throws NotFound
    deleteMinimum() : Map⟦Key, Val⟧ throws NotFound
    removeMinimum() : ((Key, Val), Map⟦Key, Val⟧) throws NotFound
    maximum(): (Key, Val) throws NotFound
    deleteMaximum() : Map⟦Key, Val⟧ throws NotFound
    removeMaximum() : ((Key, Val), Map⟦Key, Val⟧) throws NotFound
```

If no mapping presently exists, maps $k$ to $v$.

```
    add(k : Key, v : Val) : Map⟦Key, Val⟧
```

Maps $k$ to $v$.

```
    update(k : Key, v : Val) : Map⟦Key, Val⟧
```

Eliminate any mapping for key $k$.

```
    delete(k : Key) : Map⟦Key, Val⟧
```

Process mapping for key $k$ with function $f$:

- If no mapping exists, $f$ is passed $\mathrm{Nothing}[\![\mathrm{Val}]\!]$
- If $k$ maps to value $v$, $f$ is passed $\mathrm{Just}[\![\mathrm{Val}]\!](v)$

If $f$ returns $\mathrm{Nothing}$, any mapping for $k$ is deleted; otherwise, $k$ is mapped to the value contained in the result.
$\mathit{updateWith}(f : \mathrm{Maybe}[\![\mathrm{Val}]\!] \rightarrow \mathrm{Maybe}[\![\mathrm{Val}]\!], k : \mathrm{Key}) \colon \mathrm{Map}[\![\mathrm{Key}, \mathrm{Val}]\!]$

$\cup$ favors the leftmost value when a key occurs in both maps.
`opr` $\cup($`self`$, \mathit{other} \colon \mathrm{Map}[\![\mathrm{Key}, \mathrm{Val}]\!]) \colon \mathrm{Map}[\![\mathrm{Key}, \mathrm{Val}]\!]$

$\uplus$ (disjoint union) throws the $\mathrm{KeyOverlap}$ exception when a key occurs in both maps.
`opr` $\uplus($`self`$, \mathit{other} \colon \mathrm{Map}[\![\mathrm{Key}, \mathrm{Val}]\!]) \colon \mathrm{Map}[\![\mathrm{Key}, \mathrm{Val}]\!]$

The $\mathit{union}$ method takes a function $f$ used to combine the values of keys that overlap.
$\mathit{union}(f : (\mathrm{Key}, \mathrm{Val}, \mathrm{Val}) \rightarrow \mathrm{Val}, \mathit{other} \colon \mathrm{Map}[\![\mathrm{Key}, \mathrm{Val}]\!]) \colon \mathrm{Map}[\![\mathrm{Key}, \mathrm{Val}]\!]$

$\mathit{combine}$ is the "swiss army knife" combinator on pairs of maps. We call $f$ on keys present in both input maps. We call $\mathit{doThis}$ on keys present in `self` but not in $\mathit{that}$. We call $\mathit{doThat}$ on keys present in $\mathit{that}$ but not in `self`. When any of these functions returns $r = \mathrm{Just}[\![\mathrm{Result}]\!]$, the key is mapped to $r.\mathit{unJust}()$ in the result. When any of these functions returns $\mathrm{Nothing}[\![\mathrm{Result}]\!]$, there is no mapping for the key in the result.

$\mathit{mapThis}$ must be equivalent to $\mathit{mapFilter}(\mathit{doThis})$ and $\mathit{mapThat}$ must be equivalent to $\mathit{mapFilter}(\mathit{doThat})$; they are included because often they can do their jobs without traversing their argument (eg. for union and interesection operations we can pass through or discard whole submaps without traversing them).
$\mathit{combine}[\![\mathrm{That}, \mathrm{Result}]\!](f : (\mathrm{Key}, \mathrm{Val}, \mathrm{That}) \rightarrow \mathrm{Maybe}[\![\mathrm{Result}]\!],$
$\mathit{doThis} : (\mathrm{Key}, \mathrm{Val}) \rightarrow \mathrm{Maybe}[\![\mathrm{Result}]\!],$
$\mathit{doThat} : (\mathrm{Key}, \mathrm{That}) \rightarrow \mathrm{Maybe}[\![\mathrm{Result}]\!],$
$\mathit{mapThis} : \mathrm{Map}[\![\mathrm{Key}, \mathrm{Val}]\!] \rightarrow \mathrm{Map}[\![\mathrm{Key}, \mathrm{Result}]\!],$
$\mathit{mapThat} : \mathrm{Map}[\![\mathrm{Key}, \mathrm{Val}]\!] \rightarrow \mathrm{Map}[\![\mathrm{Key}, \mathrm{Result}]\!],$
$\mathit{that} \colon \mathrm{Map}[\![\mathrm{Key}, \mathrm{That}]\!]) \colon \mathrm{Map}[\![\mathrm{Key}, \mathrm{Result}]\!]$

`self`$.\mathit{mapFilter}(f)$ is equivalent to:

$\{\, k \mapsto v' \mid (k, v) \leftarrow$ `self`$, v' \leftarrow f(k, v) \,\}$

It fuses generation, mapping, and filtering.
$\mathit{mapFilter}[\![\mathrm{Result}]\!](f : (\mathrm{Key}, \mathrm{Val}) \rightarrow \mathrm{Maybe}[\![\mathrm{Result}]\!]) \colon \mathrm{Map}[\![\mathrm{Key}, \mathrm{Result}]\!]$

`end`

$\mathit{mapping}[\![\mathrm{Key}, \mathrm{Val}]\!]() \colon \mathrm{Map}[\![\mathrm{Key}, \mathrm{Val}]\!]$
$\mathit{mapping}[\![\mathrm{Key}, \mathrm{Val}]\!](g \colon \mathrm{Generator}[\![(\mathrm{Key}, \mathrm{Val})]\!]) \colon \mathrm{Map}[\![\mathrm{Key}, \mathrm{Val}]\!]$

`opr` $\{\mapsto [\![\mathrm{Key}, \mathrm{Val}]\!] \mathit{xs} : (\mathrm{Key}, \mathrm{Val}) \ldots \} \colon \mathrm{Map}[\![\mathrm{Key}, \mathrm{Val}]\!]$

`opr BIG`$\{\mapsto [\![\mathrm{Key}, \mathrm{Val}]\!] g \colon (\mathrm{Reduction}[\![\mathrm{SomeList}]\!], (\mathrm{Key}, \mathrm{Val}) \rightarrow \mathrm{SomeList}) \rightarrow$
$\mathrm{SomeList}\,\} : \ \mathrm{Map}[\![\mathrm{Key}, \mathrm{Val}]\!]$

`opr BIG` $\cup\, [\![\mathrm{Key}, \mathrm{Val}]\!](g \colon (\mathrm{Reduction}[\![\mathrm{Map}[\![\mathrm{Key}, \mathrm{Val}]\!]]\!],$
$\mathrm{Map}[\![\mathrm{Key}, \mathrm{Val}]\!] \rightarrow \mathrm{Map}[\![\mathrm{Key}, \mathrm{Val}]\!]) \rightarrow$
$\mathrm{Map}[\![\mathrm{Key}, \mathrm{Val}]\!]) : \ \mathrm{Map}[\![\mathrm{Key}, \mathrm{Val}]\!]$

`opr BIG` $\uplus\, [\![\mathrm{Key}, \mathrm{Val}]\!](g \colon (\mathrm{Reduction}[\![\mathrm{Map}[\![\mathrm{Key}, \mathrm{Val}]\!]]\!],$

$\text{Map}[\![\text{Key},\text{Val}]\!] \rightarrow \text{Map}[\![\text{Key},\text{Val}]\!]) \rightarrow$
$\text{Map}[\![\text{Key},\text{Val}]\!]) : \text{Map}[\![\text{Key},\text{Val}]\!]$

`end`

## 25.7 Sparse

`api` Sparse

Trim array $v$ to length $l$.
$\mathit{trim}[\![T]\!](v : \text{Array}[\![T, \mathbb{Z}32]\!], l : \mathbb{Z}32) : \text{Array}[\![T, \mathbb{Z}32]\!]$

`object` $\text{SparseVector}[\![T,$ `nat` $n]\!](\mathit{mem} : \text{Array}[\![(\mathbb{Z}32, T), \mathbb{Z}32]\!])$
  `extends` $\text{Vector}[\![T, n]\!]$
`end`

*sparse* constructs a sparse vector from dense vector.
$\mathit{sparse}[\![T$ `extends` $\text{Number},$ `nat` $n]\!](\mathit{me} : \text{Array1}[\![\mathbb{R}64, 0, n]\!]) : \text{SparseVector}[\![\mathbb{R}64, n]\!]$

Compressed sparse row matrix.
`object` $\text{Csr}[\![N$ `extends` $\text{Number},$ `nat` $n,$ `nat` $m]\!]$
    $(\mathit{rows} : \text{Array1}[\![\text{SparseVector}[\![N, m]\!], 0, n]\!])$
  `extends` $\text{Matrix}[\![N, n, m]\!]$
`end`

Compressed sparse column matrix.
`object` $\text{Csc}[\![N$ `extends` $\text{Number},$ `nat` $n,$ `nat` $m]\!]$
    $(\mathit{cols} : \text{Array1}[\![\text{SparseVector}[\![N, n]\!], 0, m]\!])$
  `extends` $\text{Matrix}[\![N, n, m]\!]$
`end`

`end`

## 25.8 Heap

`api` Heap

Mergeable, pure priority queues (heaps).

At the moment, we are baking off several potential priority queue implementations, based on part on some advice from “Purely Functional Data Structures” by Okasaki [19].

- Pairing heaps, the current default: $O(1)$ merge; $O(\lg n)$ amortized *extractMin*, with $O(n)$ worst case. The worst case is proportional to the number of deferred merge operations performed with the min; if you merge $n$ entries tree-fashion rather than one at a time you should obtain $O(\lg n)$ worst case performance as well. The $\mathit{heap}(\mathit{gen})$ function will follow the merge structure of the underlying generator; for a well-written generator this will be sufficient to give good performance.
- Lazy-esque pairing heaps (supposedly slower but actually easier in some ways to implement than pairing heaps, and avoiding a potential stack overflow in the latter). These do not seem to be quite as whizzy in this setting as ordinary Pairing heaps: $O(\lg n)$ merge, $O(\lg n)$ worst-case *extractMin*.

- Splay heaps (noted as "fastest in practice", borne out by other experiments). Problem: $O(n)$ merge operation, vs $O(\lg n)$ for everything else. If we build heaps by performing reductions over a generator, with each iteration generating a few elements, this will be a problem. This is not yet implemented.

Minimum complete implementation of $\mathrm{Heap}[\![K,V]\!]$:

```
    $empty$
    $singleton$
    $isEmpty$
    $extractMin$
    $merge(\mathrm{Heap}[\![K,V]\!])$
```

```
trait $\mathrm{Heap}[\![K,V]\!]$ extends $\mathrm{Generator}[\![(K,V)]\!]$
    getter $isEmpty()\colon \mathrm{Boolean}$
```

Given an instance of $\mathrm{Heap}[\![K,V]\!]$, get the empty $\mathrm{Heap}[\![K,V]\!]$.

```
    getter $empty()\colon \mathrm{Heap}[\![K,V]\!]$
```

Get the (key,value) pair with minimal associated key.

```
    getter $minimum()\colon (K,V)$ throws $\mathrm{NotFound}$
```

Given an instance of $\mathrm{Heap}[\![K,V]\!]$, generate a singleton $\mathrm{Heap}[\![K,V]\!]$.

```
    $singleton(k\colon K, v\colon V)\colon \mathrm{Heap}[\![K,V]\!]$
```

Return a heap that contains the key-value pairings in both of the heaps passed in.

```
    $merge(h\colon \mathrm{Heap}[\![K,V]\!])\colon \mathrm{Heap}[\![K,V]\!]$
```

Return a heap that contains the additional key-value pairs.

```
    $insert(k\colon K, v\colon V)\colon \mathrm{Heap}[\![K,V]\!]$
```

Extract the (key,value) pair with minimal associated key, along with a heap with that key and value pair removed.

```
    $extractMin()\colon (K,V,\mathrm{Heap}[\![K,V]\!])$ throws $\mathrm{NotFound}$
end
object $\mathrm{HeapMerge}[\![K,V]\!](boiler\colon \mathrm{Heap}[\![K,V]\!])$ extends $\mathrm{Reduction}[\![\mathrm{Heap}[\![K,V]\!]]\!]$
end
trait $\mathrm{Pairing}[\![K,V]\!]$ extends $\mathrm{Heap}[\![K,V]\!]$
        comprises $\{\ldots\}$
    $dump()\colon \mathrm{String}$
end
$emptyPairing[\![K,V]\!]()\colon \mathrm{Pairing}[\![K,V]\!]$
$singletonPairing[\![K,V]\!](k\colon K, v\colon V)\colon \mathrm{Pairing}[\![K,V]\!]$
$pairing[\![K,V]\!](g\colon \mathrm{Generator}[\![(K,V)]\!])\colon \mathrm{Pairing}[\![K,V]\!]$
```

Not actually lazy pairing heaps; these are actuallly more eager in that they merge siblings incrementally on insertion.

```
trait LazyPairing⟦K, V⟧ extends Heap⟦K, V⟧
        comprises { ... }
    dump(): String
end
```

$emptyLazy⟦K, V⟧(): \text{LazyPairing}⟦K, V⟧$
$singletonLazy⟦K, V⟧(k : K, v : V): \text{LazyPairing}⟦K, V⟧$
$lazy⟦K, V⟧(g : \text{Generator}⟦(K, V)⟧): \text{Heap}⟦K, V⟧$

Use these default factories unless you are experimenting. Right now, they yield non-lazy pairing heaps.

$emptyHeap⟦K, V⟧(): \text{Pairing}⟦K, V⟧$
$singletonHeap⟦K, V⟧(k : K, v : V): \text{Pairing}⟦K, V⟧$
$heap⟦K, V⟧(g : \text{Generator}⟦(K, V)⟧): \text{Pairing}⟦K, V⟧$

`end`

## 25.9 SkipList

```
api SkipList
import PureList.{...}
```

A SkipList type consists of a root node and $pInverse = 1/p$, where the fraction $p$ is used in the negative binomial distribution to select random levels for insertion.

```
trait SkipList⟦Key, Val, nat pInverse⟧
    comprises { ... }
  toString() : String
```

The number of values stored in this tree.

$size() : \mathbb{Z}32$

Given a search key, try to return a value that matches that key.

$search(k : \text{Key}) : \text{Maybe}⟦\text{Val}⟧$

Add a (key, value) pair to this tree.

$add(k : \text{Key}, v : \text{Val}) : \text{SkipList}⟦\text{Key}, \text{Val}, pInverse⟧$

Remove one (key, value) pair from this tree.

$remove(k : \text{Key}) : \text{SkipList}⟦\text{Key}, \text{Val}, pInverse⟧$

Merge two skip trees.

$merge(other : \text{SkipList}⟦\text{Key}, \text{Val}, pInverse⟧) : \text{SkipList}⟦\text{Key}, \text{Val}, pInverse⟧$

`end`

(∗ Construct an empty skip list. ∗)
NewList⟦Key, Val, `nat` *pInverse*⟧() : SkipList⟦Key, Val, *pInverse*⟧

`end`

## 25.10 QuickSort

`api` QuickSort

`http://en.wikipedia.org/wiki/Quicksort`

$\mathit{quicksort}[\![T]\!](\mathit{lt}: (T, T) \rightarrow \text{Boolean}, \mathit{arr}: \text{Array}[\![T, \mathbb{Z}32]\!], \mathit{left}: \mathbb{Z}32, \mathit{right}: \mathbb{Z}32): ()$

`end`

# Part V

# Appendices

# Appendix A

# Fortress Calculi

## A.1 Basic Core Fortress

In this section, we define a basic core calculus for Fortress. We call this calculus *Basic Core Fortress*. Following the precedent set by prior core calculi such as Featherweight Generic Java [10], we have abided by the restriction that all valid Basic Core Fortress programs are valid Fortress programs.

### A.1.1 Syntax

A syntax for Basic Core Fortress is provided in Figure A.1. We use the following notational conventions:

- For brevity, we omit separators such as , and ; from Basic Core Fortress.
- $\overrightarrow{\tau}$ is a shorthand for a (possibly empty) sequence $\tau_1, \cdots, \tau_n$.
- Similarly, we abbreviate a sequence of relations $\alpha_1$ `extends` $N_1, \cdots, \alpha_n$ `extends` $N_n$ to $\overrightarrow{\alpha\ \texttt{extends}\ N}$
- We use $\tau_i$ to denote the $i$th element of $\overrightarrow{\tau}$.
- For simplicity, we assume that every name (type variables, field names, and parameters) is different and every trait/object declaration declares unique name.
- We prohibit cycles in type hierarchies.

The syntax of Basic Core Fortress allows only a small subset of the Fortress language to be formalized. Basic Core Fortress includes trait and object definitions, method and field invocations, and `self` expressions. The types of Basic Core Fortress include type variables, instantiated traits, instantiated objects, and the distinguished trait Object. Note that we syntactically prohibit extending objects. Among other features, Basic Core Fortress does *not* include top-level variable and function definitions, overloading, `excludes` clauses, `comprises` clauses, `where` clauses, object expressions, and function expressions. Basic Core Fortress will be extended to formalize a larger set of Fortress programs in the future.

### A.1.2 Dynamic Semantics

A dynamic semantics for Basic Core Fortress is provided in Figure A.2. This semantics has been mechanized via the PLT Redex tool [13]. It therefore follows the style of explicit evaluation contexts and redexes. The Basic Core

| | | | |
|---|---|---|---|
| $\alpha, \beta$ | | | type variables |
| $f$ | | | method name |
| $x$ | | | field name |
| $T$ | | | trait name |
| $O$ | | | object name |
| $\tau, \tau', \tau''$ | $::=$ | $\alpha$ | type |
| | $\mid$ | $\sigma$ | |
| $\sigma$ | $::=$ | $N$ | type that is not a type variable |
| | $\mid$ | $O[\![\overrightarrow{\tau}]\!]$ | |
| $N, M, L$ | $::=$ | $T[\![\overrightarrow{\tau}]\!]$ | type that can be a type bound |
| | $\mid$ | $\mathrm{Object}$ | |
| $e$ | $::=$ | $x$ | expression |
| | $\mid$ | $\texttt{self}$ | |
| | $\mid$ | $O[\![\overrightarrow{\tau}]\!](\overrightarrow{e})$ | |
| | $\mid$ | $e.x$ | |
| | $\mid$ | $e.f[\![\overrightarrow{\tau}]\!](\overrightarrow{e})$ | |
| $\mathit{fd}$ | $::=$ | $f[\![\overrightarrow{\alpha\ \texttt{extends}\ N}]\!](\overrightarrow{x{:}\tau}){:}\tau = e$ | method definition |
| $\mathit{td}$ | $::=$ | $\texttt{trait}\ T[\![\overrightarrow{\alpha\ \texttt{extends}\ N}]\!]\ \texttt{extends}\ \{\overrightarrow{N}\}\ \overrightarrow{\mathit{fd}}\ \texttt{end}$ | trait definition |
| $\mathit{od}$ | $::=$ | $\texttt{object}\ O[\![\overrightarrow{\alpha\ \texttt{extends}\ N}]\!](\overrightarrow{x{:}\tau})\ \texttt{extends}\ \{\overrightarrow{N}\}\ \overrightarrow{\mathit{fd}}\ \texttt{end}$ | object definition |
| $d$ | $::=$ | $\mathit{td}$ | definition |
| | $\mid$ | $\mathit{od}$ | |
| $p$ | $::=$ | $\overrightarrow{d}\ e$ | program |

Figure A.1: Syntax of Basic Core Fortress

Fortress dynamic semantics consists of two evaluations rules: one for field access and another for method invocation. For simplicity, we use ‘ _ ’ to denote some parts of the syntax that do not have key roles in a rule. We assume that _ does not expand across definition boundaries unless the entire definition is included in it.

### A.1.3 Static Semantics

A static semantics for Basic Core Fortress is provided in Figures A.3, A.4, and A.5. The Basic Core Fortress static semantics is based on the static semantics of Featherweight Generic Java (FGJ) [10]. The major difference is the division of classes into traits and objects. Both trait and object definitions include method definitions but only object definitions include field definitions. With traits, Basic Core Fortress supports multiple inheritance. However, due to the similarity of traits and objects, many of the rules in the Basic Core Fortress dynamic and static semantics combine the two cases. Note that Basic Core Fortress allows parametric polymorphism, subtype polymorphism, and overriding in much the same way that FGJ does.

We proved the type soundness of Basic Core Fortress using the standard technique of proving a progress theorem and a subject reduction theorem.

Values, evaluation contexts, redexes, and trait and object names

$$\begin{array}{lcll}
v & ::= & O[\![\overrightarrow{\tau}]\!](\overrightarrow{v}) & \text{value} \\
E & ::= & \Box & \text{evaluation context} \\
 & | & O[\![\overrightarrow{\tau}]\!](\overrightarrow{e}\,E\,\overrightarrow{e}) & \\
 & | & E.x & \\
 & | & E.f[\![\overrightarrow{\tau}]\!](\overrightarrow{e}) & \\
 & | & e.f[\![\overrightarrow{\tau}]\!](\overrightarrow{e}\,E\,\overrightarrow{e}) & \\
R & ::= & v.x & \text{redex} \\
 & | & v.f[\![\overrightarrow{\tau}]\!](\overrightarrow{v}) & \\
C & ::= & T & \text{trait name} \\
 & | & O & \text{object name}
\end{array}$$

Evaluation rules: $\boxed{p \vdash E[R] \longrightarrow E[e]}$

$$[\text{R-F}\small{\text{IELD}}] \qquad \frac{\texttt{object}\ O\ \_\ (\overrightarrow{x{:}\,\_})\ \_\ \texttt{end} \in p}{p \vdash E[O[\![\overrightarrow{\tau}]\!](\overrightarrow{v}).x_i] \longrightarrow E[v_i]}$$

$$[\text{R-M}\small{\text{ETHOD}}] \qquad \frac{\texttt{object}\ O\ \_\ (\overrightarrow{x{:}\,\_})\ \_\ \texttt{end} \in p \qquad \mathit{mbody}_p(f[\![\overrightarrow{\tau'}]\!], O[\![\overrightarrow{\tau}]\!]) = \{(\overrightarrow{x'}) \to e\}}{p \vdash E[O[\![\overrightarrow{\tau}]\!](\overrightarrow{v}).f[\![\overrightarrow{\tau'}]\!](\overrightarrow{v'})] \longrightarrow E[[\overrightarrow{v}/\overrightarrow{x}][O[\![\overrightarrow{\tau}]\!](\overrightarrow{v})/\texttt{self}][\overrightarrow{v'}/\overrightarrow{x'}]e]}$$

Method body lookup: $\boxed{\mathit{mbody}_p(f[\![\overrightarrow{\tau}]\!], \tau) = \{(\overrightarrow{x}) \to e\}}$

$$[\text{M}\small{\text{B}}\text{-S}\small{\text{ELF}}] \qquad \frac{\_\ C[\![\overrightarrow{\alpha\ \texttt{extends}\ \_}]\!]\ \_\ \overrightarrow{fd}\ \_ \in p \qquad f[\![\overrightarrow{\alpha'\ \texttt{extends}\ \_}]\!](\overrightarrow{x'{:}\,\_})\ \_\ = e \in \{\overrightarrow{fd}\}}{\mathit{mbody}_p(f[\![\overrightarrow{\tau'}]\!], C[\![\overrightarrow{\tau}]\!]) = \{[\overrightarrow{\tau'}/\overrightarrow{\alpha'}][\overrightarrow{\tau}/\overrightarrow{\alpha}](\overrightarrow{x'}) \to e\}}$$

$$[\text{M}\small{\text{B}}\text{-S}\small{\text{UPER}}] \qquad \frac{\_\ C[\![\overrightarrow{\alpha\ \texttt{extends}\ \_}]\!]\ \_\ \ \texttt{extends}\ \{\overrightarrow{N}\}\ \_\ \overrightarrow{fd}\ \_ \in p \qquad f \notin \{\overrightarrow{\mathit{Fname}(\mathit{fd})}\}}{\mathit{mbody}_p(f[\![\overrightarrow{\tau'}]\!], C[\![\overrightarrow{\tau}]\!]) = \displaystyle\bigcup_{N_i \in \{\overrightarrow{N}\}} \mathit{mbody}_p(f[\![\overrightarrow{\tau'}]\!], [\overrightarrow{\tau}/\overrightarrow{\alpha}]N_i)}$$

$$[\text{M}\small{\text{B}}\text{-O}\small{\text{BJ}}] \qquad \mathit{mbody}_p(f[\![\overrightarrow{\tau}]\!], \text{Object}) = \emptyset$$

Function/method name lookup: $\boxed{\mathit{Fname}(\mathit{fd}) = f}$

$\mathit{Fname}(f[\![\overrightarrow{\alpha\ \texttt{extends}\ N}]\!](\overrightarrow{x{:}\tau}){:}\tau = e) = f$

Figure A.2: Dynamic Semantics of Basic Core Fortress

Environments

| | | | |
|---|---|---|---|
| $\Delta$ | $::=$ | $\overrightarrow{\alpha <: N}$ | bound environment |
| $\Gamma$ | $::=$ | $\overrightarrow{x : \tau}$ | type environment |

Program typing: $\boxed{\vdash p : \tau}$

$$[\text{T-PROGRAM}] \quad \frac{p = \overrightarrow{d}\ e \qquad p \vdash \overrightarrow{d}\ \mathsf{ok} \qquad p; \emptyset; \emptyset \vdash e : \tau}{\vdash p : \tau}$$

Definition typing: $\boxed{p \vdash d\ \mathsf{ok}}$

$$[\text{T-TRAITDEF}] \quad \frac{\Delta = \overrightarrow{\alpha <: N} \qquad p; \Delta \vdash \overrightarrow{N}\ \mathsf{ok} \qquad p; \Delta \vdash \overrightarrow{M}\ \mathsf{ok} \qquad p; \Delta; \mathtt{self} : T[\![\overrightarrow{\alpha}]\!]; T \vdash \overrightarrow{fd}\ \mathsf{ok} \qquad p \vdash \mathit{oneOwner}(T)}{p \vdash \mathtt{trait}\ T[\![\overrightarrow{\alpha\ \mathtt{extends}\ N}]\!]\ \mathtt{extends}\ \{\overrightarrow{M}\}\ \overrightarrow{fd}\ \mathtt{end}\ \mathsf{ok}}$$

$$[\text{T-OBJECTDEF}] \quad \frac{\Delta = \overrightarrow{\alpha <: N} \qquad p; \Delta \vdash \overrightarrow{N}\ \mathsf{ok} \qquad p; \Delta \vdash \overrightarrow{\tau}\ \mathsf{ok} \qquad p; \Delta \vdash \overrightarrow{M}\ \mathsf{ok} \qquad p; \Delta; \mathtt{self} : O[\![\overrightarrow{\alpha}]\!]\ \overrightarrow{x : \tau}; O \vdash \overrightarrow{fd}\ \mathsf{ok} \qquad p \vdash \mathit{oneOwner}(O)}{p \vdash \mathtt{object}\ O[\![\overrightarrow{\alpha\ \mathtt{extends}\ N}]\!](\overrightarrow{x{:}\tau})\ \mathtt{extends}\ \{\overrightarrow{M}\}\ \overrightarrow{fd}\ \mathtt{end}\ \mathsf{ok}}$$

Method typing: $\boxed{p; \Delta; \Gamma; C \vdash fd\ \mathsf{ok}}$

$$[\text{T-METHODDEF}] \quad \frac{\_\ C\ \_\ \ \mathtt{extends}\ \{\overrightarrow{M}\}\ \_\ \in p \qquad p; \Delta \vdash \mathit{override}(f, \{\overrightarrow{M}\}, [\![\overrightarrow{\alpha\ \mathtt{extends}\ N}]\!]\ \overrightarrow{\tau} \to \tau_0) \qquad \Delta' = \Delta\ \overrightarrow{\alpha <: N} \qquad p; \Delta' \vdash \overrightarrow{N}\ \mathsf{ok} \qquad p; \Delta' \vdash \overrightarrow{\tau}\ \mathsf{ok} \qquad p; \Delta' \vdash \tau_0\ \mathsf{ok} \qquad p; \Delta'; \Gamma\ \overrightarrow{x : \tau} \vdash e : \tau' \qquad p; \Delta' \vdash \tau' <: \tau_0}{p; \Delta; \Gamma; C \vdash f[\![\overrightarrow{\alpha\ \mathtt{extends}\ N}]\!](\overrightarrow{x{:}\tau}){:}\tau_0 = e\ \mathsf{ok}}$$

Method overriding: $\boxed{p; \Delta \vdash \mathit{override}(f, \{\overrightarrow{N}\}, [\![\overrightarrow{\alpha\ \mathtt{extends}\ N}]\!]\ \overrightarrow{\tau} \to \tau)}$

$$[\text{OVERRIDE}] \quad \frac{\bigcup_{L_i \in \{\overrightarrow{L}\}} \mathit{mtype}_p(f, L_i) = \{[\![\overrightarrow{\beta\ \mathtt{extends}\ M}]\!]\ \overrightarrow{\tau'} \to \tau_0'\} \qquad \overrightarrow{N} = [\overrightarrow{\alpha}/\overrightarrow{\beta}]\overrightarrow{M} \qquad \overrightarrow{\tau} = [\overrightarrow{\alpha}/\overrightarrow{\beta}]\overrightarrow{\tau'} \qquad p; \overrightarrow{\alpha <: N} \vdash \tau_0 <: [\overrightarrow{\alpha}/\overrightarrow{\beta}]\tau_0'}{p; \Delta \vdash \mathit{override}(f, \{\overrightarrow{L}\}, [\![\overrightarrow{\alpha\ \mathtt{extends}\ N}]\!]\ \overrightarrow{\tau} \to \tau_0)}$$

Method type lookup: $\boxed{\mathit{mtype}_p(f, \tau) = \{[\![\overrightarrow{\alpha\ \mathtt{extends}\ N}]\!]\ \overrightarrow{\tau} \to \tau\}}$

$$[\text{MT-SELF}] \quad \frac{\_\ C[\![\overrightarrow{\alpha\ \mathtt{extends}\ \_}]\!]\ \_\ \overrightarrow{fd}\ \_\ \in p \qquad f[\![\overrightarrow{\beta\ \mathtt{extends}\ M}]\!](\overrightarrow{\_{:}\tau'}){:}\tau_0' = e \in \{\overrightarrow{fd}\}}{\mathit{mtype}_p(f, C[\![\overrightarrow{\tau}]\!]) = \{[\overrightarrow{\tau}/\overrightarrow{\alpha}][\![\overrightarrow{\beta\ \mathtt{extends}\ M}]\!]\ \overrightarrow{\tau'} \to \tau_0'\}}$$

$$[\text{MT-SUPER}] \quad \frac{\_\ C[\![\overrightarrow{\alpha\ \mathtt{extends}\ \_}]\!]\ \_\ \ \mathtt{extends}\ \{\overrightarrow{N}\}\ \_\ \overrightarrow{fd}\ \_\ \in p \qquad f \notin \{\overrightarrow{\mathit{Fname}(fd)}\}}{\mathit{mtype}_p(f, C[\![\overrightarrow{\tau}]\!]) = \bigcup_{N_i \in \{\overrightarrow{N}\}} \mathit{mtype}_p(f, [\overrightarrow{\tau}/\overrightarrow{\alpha}]N_i)}$$

$$[\text{MT-OBJ}] \qquad \mathit{mtype}_p(f, \text{Object}) = \emptyset$$

Figure A.3: Static Semantics of Basic Core Fortress (I)

Expression typing: $\boxed{p;\Delta;\Gamma \vdash e : \tau}$

[T-VAR] $$p;\Delta;\Gamma \vdash x : \Gamma(x)$$

[T-SELF] $$p;\Delta;\Gamma \vdash \texttt{self} : \Gamma(\texttt{self})$$

[T-OBJECT] $$\frac{\begin{array}{c}\texttt{object}\ O[\![\overrightarrow{\alpha\ \texttt{extends}\ \_}]\!](\overrightarrow{\_:\tau'})\ \_\ \texttt{end} \in p \qquad p;\Delta \vdash O[\![\overrightarrow{\tau}]\!]\,\mathsf{ok}\\ p;\Delta;\Gamma \vdash \overrightarrow{e} : \overrightarrow{\tau''} \qquad p;\Delta \vdash \overrightarrow{\tau''}\ <:\ [\overrightarrow{\tau}/\overrightarrow{\alpha}]\overrightarrow{\tau'}\end{array}}{p;\Delta;\Gamma \vdash O[\![\overrightarrow{\tau}]\!](\overrightarrow{e}) : O[\![\overrightarrow{\tau}]\!]}$$

[T-FIELD] $$\frac{p;\Delta;\Gamma \vdash e_0 : \tau_0 \qquad \mathit{bound}_\Delta(\tau_0) = O[\![\overrightarrow{\tau'}]\!] \qquad \texttt{object}\ O[\![\overrightarrow{\alpha\ \texttt{extends}\ \_}]\!](\overrightarrow{x{:}\tau})\ \_\ \texttt{end} \in p}{p;\Delta;\Gamma \vdash e_0\,.\,x_i : [\overrightarrow{\tau'}/\overrightarrow{\alpha}]\tau_i}$$

[T-METHOD] $$\frac{\begin{array}{c}p;\Delta;\Gamma \vdash e_0 : \tau_0 \qquad \mathit{mtype}_p(f, \mathit{bound}_\Delta(\tau_0)) = \{[\![\overrightarrow{\alpha\ \texttt{extends}\ N}]\!]\ \overrightarrow{\tau'} \to \tau_0'\}\\ p;\Delta \vdash \overrightarrow{\tau}\ \mathsf{ok} \qquad p;\Delta \vdash \overrightarrow{\tau}\ <:\ [\overrightarrow{\tau}/\overrightarrow{\alpha}]\overrightarrow{N}\\ p;\Delta;\Gamma \vdash \overrightarrow{e} : \overrightarrow{\tau''} \qquad p;\Delta \vdash \overrightarrow{\tau''}\ <:\ [\overrightarrow{\tau}/\overrightarrow{\alpha}]\overrightarrow{\tau'}\end{array}}{p;\Delta;\Gamma \vdash e_0.f[\![\overrightarrow{\tau}]\!](\overrightarrow{e}) : [\overrightarrow{\tau}/\overrightarrow{\alpha}]\tau_0'}$$

Subtyping: $\boxed{p;\Delta \vdash \tau\ <:\ \tau}$

[S-OBJ] $$p;\Delta \vdash \tau\ <:\ \text{Object}$$

[S-REFL] $$p;\Delta \vdash \tau\ <:\ \tau$$

[S-TRANS] $$\frac{p;\Delta \vdash \tau_1\ <:\ \tau_2 \qquad p;\Delta \vdash \tau_2\ <:\ \tau_3}{p;\Delta \vdash \tau_1\ <:\ \tau_3}$$

[S-VAR] $$p;\Delta \vdash \alpha\ <:\ \Delta(\alpha)$$

[S-TAPP] $$\frac{\_\ C[\![\overrightarrow{\alpha\ \texttt{extends}\ \_}]\!]\ \_\ \ \texttt{extends}\ \{\overrightarrow{N}\}\ \_\ \in p}{p;\Delta \vdash C[\![\overrightarrow{\tau}]\!]\ <:\ [\overrightarrow{\tau}/\overrightarrow{\alpha}]N_i}$$

Well-formed types: $\boxed{p;\Delta \vdash \tau\ \mathsf{ok}}$

[W-OBJ] $$p;\Delta \vdash \text{Object}\ \mathsf{ok}$$

[W-VAR] $$\frac{\alpha \in \mathit{dom}(\Delta)}{p;\Delta \vdash \alpha\ \mathsf{ok}}$$

[W-TAPP] $$\frac{\_\ C[\![\overrightarrow{\alpha\ \texttt{extends}\ N}]\!]\ \_\ \in p \qquad p;\Delta \vdash \overrightarrow{\tau}\ \mathsf{ok} \qquad p;\Delta \vdash \overrightarrow{\tau}\ <:\ [\overrightarrow{\tau}/\overrightarrow{\alpha}]\overrightarrow{N}}{p;\Delta \vdash C[\![\overrightarrow{\tau}]\!]\ \mathsf{ok}}$$

Figure A.4: Static Semantics of Basic Core Fortress (II)

Bound of type: $\boxed{bound_\Delta(\tau) = \sigma}$

$$\begin{array}{lcl} bound_\Delta(\alpha) & = & \Delta(\alpha) \\ bound_\Delta(\sigma) & = & \sigma \end{array}$$

One owner for all the visible methods: $\boxed{p \vdash oneOwner(C)}$

$$[\textsc{OneOwner}] \quad \frac{\forall f \in visible_p(C)\ .\ f \textit{ only occurs once in } visible_p(C)}{p \vdash oneOwner(C)}$$

Auxiliary functions for methods: $\boxed{defined_p \text{ / } inherited_p \text{ / } visible_p(C) = \{\overrightarrow{f}\}}$

$$\begin{array}{rll} defined_p(C) = & \{\overrightarrow{Fname(fd)}\} & \text{where } \_\ C\ \_\ \overrightarrow{fd}\ \_\ \in p \\ inherited_p(C) = & \biguplus_{N_i \in \{\overrightarrow{N}\}} \{f_i \mid f_i \in visible_p(N_i), f_i \notin defined_p(C)\} & \text{where } \_\ C\ \_\ \texttt{extends}\ \{\overrightarrow{N}\}\ \_\ \in p \\ visible_p(C) = & defined_p(C) \uplus inherited_p(C) & \end{array}$$

Figure A.5: Static Semantics of Basic Core Fortress (III)

## A.2 Core Fortress with Overloading

In this section, we define a Fortress core calculus with overloading for dotted methods and first-order functions. We call this calculus *Core Fortress with Overloading*. Core Fortress with Overloading is an extension of Basic Core Fortress with overloading.

### A.2.1 Syntax

The syntax for Core Fortress with Overloading is provided in Figure A.6.

### A.2.2 Dynamic Semantics

A dynamic semantics for Core Fortress with Overloading is provided in Figure A.7.

### A.2.3 Static Semantics

A static semantics for Core Fortress with Overloading is provided in Figures A.8, A.9, A.10, and A.11.

We proved the type soundness of Core Fortress with Overloading using the standard technique of proving a progress theorem and a subject reduction theorem.

| | | | |
|---|---|---|---|
| $\alpha, \beta$ | | | type variables |
| $f$ | | | function or method name |
| $x$ | | | field name |
| $T$ | | | trait name |
| $O$ | | | object name |
| $\tau, \tau', \tau''$ | $::=$ | $\alpha$ | type |
| | $\mid$ | $\sigma$ | |
| $\sigma$ | $::=$ | $N$ | type that is not a type variable |
| | $\mid$ | $O[\![\overrightarrow{\tau}]\!]$ | |
| $N, M, L$ | $::=$ | $T[\![\overrightarrow{\tau}]\!]$ | type that can be a type bound |
| | $\mid$ | $\mathrm{Object}$ | |
| $e$ | $::=$ | $x$ | expression |
| | $\mid$ | $\texttt{self}$ | |
| | $\mid$ | $O[\![\overrightarrow{\tau}]\!](\overrightarrow{e})$ | |
| | $\mid$ | $e.x$ | |
| | $\mid$ | $e.f[\![\overrightarrow{\tau}]\!](\overrightarrow{e})$ | |
| | $\mid$ | $f[\![\overrightarrow{\tau}]\!](\overrightarrow{e})$ | |
| $\mathit{fd}$ | $::=$ | $f[\![\overrightarrow{\alpha\ \texttt{extends}\ N}]\!](\overrightarrow{x{:}\tau}){:}\tau = e$ | function or method definition |
| $\mathit{td}$ | $::=$ | $\texttt{trait}\ T[\![\overrightarrow{\alpha\ \texttt{extends}\ N}]\!]\ \texttt{extends}\ \{\overrightarrow{N}\}\ \overrightarrow{\mathit{fd}}\ \texttt{end}$ | trait definition |
| $\mathit{od}$ | $::=$ | $\texttt{object}\ O[\![\overrightarrow{\alpha\ \texttt{extends}\ N}]\!](\overrightarrow{x{:}\tau})\ \texttt{extends}\ \{\overrightarrow{N}\}\ \overrightarrow{\mathit{fd}}\ \texttt{end}$ | object definition |
| $d$ | $::=$ | $\mathit{fd}$ | definition |
| | $\mid$ | $\mathit{td}$ | |
| | $\mid$ | $\mathit{od}$ | |
| $p$ | $::=$ | $\overrightarrow{d}\ e$ | program |

Figure A.6: Syntax of Core Fortress with Overloading

Values, evaluation contexts, and redexes

$$
\begin{array}{lcll}
v & ::= & O[\![\overrightarrow{\tau}]\!](\overrightarrow{v}) & \text{value} \\
E & ::= & \square & \text{evaluation context} \\
 & \mid & O[\![\overrightarrow{\tau}]\!](\overrightarrow{e}\,E\,\overrightarrow{e}) & \\
 & \mid & E.x & \\
 & \mid & E.f[\![\overrightarrow{\tau}]\!](\overrightarrow{e}) & \\
 & \mid & e.f[\![\overrightarrow{\tau}]\!](\overrightarrow{e}\,E\,\overrightarrow{e}) & \\
 & \mid & f[\![\overrightarrow{\tau}]\!](\overrightarrow{e}\,E\,\overrightarrow{e}) & \\
R & ::= & v.x & \text{redex} \\
 & \mid & v.f[\![\overrightarrow{\tau}]\!](\overrightarrow{v}) & \\
 & \mid & f[\![\overrightarrow{\tau}]\!](\overrightarrow{v}) &
\end{array}
$$

Evaluation rules: $\boxed{p \vdash E[R] \longrightarrow E[e]}$

[R-FIELD]
$$\frac{\texttt{object}\; O \,\_\, (\overrightarrow{x\colon \_}\,)\,\_\; \texttt{end} \in p}{p \vdash E[O[\![\overrightarrow{\tau}]\!](\overrightarrow{v}).x_i] \longrightarrow E[v_i]}$$

[R-METHOD]
$$\frac{\begin{array}{c}\texttt{object}\; O[\![\overrightarrow{\alpha\;\texttt{extends}\;N}]\!](\overrightarrow{x\colon \_}\,)\,\_\; \texttt{end} \in p \qquad \overrightarrow{\mathit{type}(v')} = \overrightarrow{\tau''} \\ \mathit{mostspecific}_{p;\emptyset}(\mathit{applicable}_{p;\emptyset}(f[\![\overrightarrow{\tau'}]\!](\overrightarrow{\tau''}), \mathit{visible}_p(O[\![\overrightarrow{\tau}]\!]))) = f[\![\overrightarrow{\alpha'\;\texttt{extends}\;N'}]\!](\overrightarrow{x'\colon \_}\,)\colon \_ = e\end{array}}{p \vdash E[O[\![\overrightarrow{\tau}]\!](\overrightarrow{v}).f[\![\overrightarrow{\tau'}]\!](\overrightarrow{v'})] \longrightarrow E[\overrightarrow{v}/\overrightarrow{x}][O[\![\overrightarrow{\tau}]\!](\overrightarrow{v})/\texttt{self}][\overrightarrow{v'}/\overrightarrow{x'}]e]}$$

[R-FUNCTION]
$$\frac{\begin{array}{c}\overrightarrow{\mathit{type}(v)} = \overrightarrow{\tau'} \\ \mathit{mostspecific}_{p;\emptyset}(\mathit{applicable}_{p;\emptyset}(f[\![\overrightarrow{\tau}]\!](\overrightarrow{\tau'}), \{(\mathit{fd}, \mathrm{Object}) \mid \mathit{fd} \in p\})) = f[\![\overrightarrow{\alpha\;\texttt{extends}\;N}]\!](\overrightarrow{x\colon \_}\,)\colon \_ = e\end{array}}{p \vdash E[f[\![\overrightarrow{\tau}]\!](\overrightarrow{v})] \longrightarrow E[\overrightarrow{v}/\overrightarrow{x}]e]}$$

Types of values: $\boxed{\mathit{type}(v) = \tau}$

$\mathit{type}(O[\![\overrightarrow{\tau}]\!](\overrightarrow{v})) = O[\![\overrightarrow{\tau}]\!]$

Figure A.7: Dynamic Semantics of Core Fortress with Overloading

Environments and trait or object names

| | | | |
|---|---|---|---|
| $\Delta$ | $::=$ | $\overrightarrow{\alpha <: N}$ | bound environment |
| $\Gamma$ | $::=$ | $\overrightarrow{x : \tau}$ | type environment |
| $C$ | $::=$ | $T$ | trait name |
| | $\mid$ | $O$ | object name |

Program typing: $\boxed{\vdash p : \tau}$

$$[\text{T-P}\textsc{rogram}]\quad \frac{p = \overrightarrow{d}\ e \qquad p;\emptyset;\emptyset \vdash \overrightarrow{d}\ \mathsf{ok} \qquad p \vdash \mathit{validFun}(\overrightarrow{d}) \qquad p;\emptyset;\emptyset \vdash e : \tau}{\vdash p : \tau}$$

Trait typing: $\boxed{p;\emptyset;\emptyset \vdash \mathit{td}\ \mathsf{ok}}$

$$[\text{T-T}\textsc{rait}\text{D}\textsc{ef}]\quad \frac{\begin{array}{c}\Delta = \overrightarrow{\alpha <: N} \qquad p;\Delta \vdash \overrightarrow{N}\ \mathsf{ok} \qquad p;\Delta \vdash \overrightarrow{M}\ \mathsf{ok}\\ p;\Delta;\mathtt{self} : T[\![\overrightarrow{\alpha}]\!] \vdash \overrightarrow{\mathit{fd}}\ \mathsf{ok} \qquad p \vdash \mathit{validMeth}(T)\end{array}}{p;\emptyset;\emptyset \vdash \mathtt{trait}\ T[\![\overrightarrow{\alpha\ \mathtt{extends}\ N}]\!]\ \mathtt{extends}\ \{\overrightarrow{M}\}\ \overrightarrow{\mathit{fd}}\ \mathtt{end}\ \mathsf{ok}}$$

Object typing: $\boxed{p;\emptyset;\emptyset \vdash \mathit{od}\ \mathsf{ok}}$

$$[\text{T-O}\textsc{bject}\text{D}\textsc{ef}]\quad \frac{\begin{array}{c}\Delta = \overrightarrow{\alpha <: N} \qquad p;\Delta \vdash \overrightarrow{N}\ \mathsf{ok} \qquad p;\Delta \vdash \overrightarrow{\tau}\ \mathsf{ok} \qquad p;\Delta \vdash \overrightarrow{M}\ \mathsf{ok}\\ p;\Delta;\mathtt{self} : O[\![\overrightarrow{\alpha}]\!]\ \overrightarrow{x : \tau} \vdash \overrightarrow{\mathit{fd}}\ \mathsf{ok} \qquad p \vdash \mathit{validMeth}(O)\end{array}}{p;\emptyset;\emptyset \vdash \mathtt{object}\ O[\![\overrightarrow{\alpha\ \mathtt{extends}\ N}]\!](\overrightarrow{x{:}\tau})\ \mathtt{extends}\ \{\overrightarrow{M}\}\ \overrightarrow{\mathit{fd}}\ \mathtt{end}\ \mathsf{ok}}$$

Function and method typing: $\boxed{p;\Delta;\Gamma \vdash \mathit{fd}\ \mathsf{ok}}$

$$[\text{T-F}\textsc{un}\text{M}\textsc{eth}\text{D}\textsc{ef}]\quad \frac{\begin{array}{c}\Delta' = \Delta\ \overrightarrow{\alpha <: N} \qquad p;\Delta' \vdash \overrightarrow{N}\ \mathsf{ok} \qquad p;\Delta' \vdash \overrightarrow{\tau}\ \mathsf{ok} \qquad p;\Delta' \vdash \tau_0\ \mathsf{ok}\\ p;\Delta';\Gamma\ \overrightarrow{x : \tau} \vdash e : \tau' \qquad p;\Delta' \vdash \tau' <: \tau_0\end{array}}{p;\Delta;\Gamma \vdash f[\![\overrightarrow{\alpha\ \mathtt{extends}\ N}]\!](\overrightarrow{x{:}\tau}){:}\tau_0 = e\ \mathsf{ok}}$$

Valid method declarations: $\boxed{p \vdash \mathit{validMeth}(C)}$

$$[\text{V}\textsc{alid}\text{M}\textsc{eth}]\quad \frac{\begin{array}{l}\forall\ (\mathit{fd}, C[\![\overrightarrow{\tau^c}]\!]), (\mathit{fd}', C'[\![\overrightarrow{\tau^{c'}}]\!]) \in \mathit{visible}_p(C^o[\![\overrightarrow{\alpha^o}]\!]).\\ \text{where} \quad \_\ C^o[\![\overrightarrow{\alpha^o\ \mathtt{extends}\ \_}]\!]\ \_ \in p,\\ \qquad\qquad \mathit{fd} \neq \mathit{fd}' \quad \text{(not same declaration)},\\ \qquad\qquad \mathit{fd} = f[\![\overrightarrow{\alpha\ \mathtt{extends}\ N}]\!](\overrightarrow{\_{:}\tau}){:}\tau^r = \_\,, \quad \mathit{fd}' = f[\![\overrightarrow{\alpha'\ \mathtt{extends}\ N'}]\!](\overrightarrow{\_{:}\tau'}){:}\tau^{r'} = \_\,,\\ p \vdash \mathit{valid}([\![\overrightarrow{\alpha\ \mathtt{extends}\ N}]\!]C[\![\overrightarrow{\tau^c}]\!]\overrightarrow{\tau} \to \tau^r, [\![\overrightarrow{\alpha'\ \mathtt{extends}\ N'}]\!]C'[\![\overrightarrow{\tau^{c'}}]\!]\overrightarrow{\tau'} \to \tau^{r'}, \mathit{visible}_p(C^o[\![\overrightarrow{\alpha^o}]\!]))\end{array}}{p \vdash \mathit{validMeth}(C^o)}$$

Figure A.8: Static Semantics of Core Fortress with Overloading (I)

Valid function declarations: $\boxed{p \vdash \mathit{validFun}(\overrightarrow{d}\,)}$

$$[\textsc{ValidFun}]\quad \frac{\begin{array}{l} \forall\, \mathit{fd}, \mathit{fd}' \in \overrightarrow{d}\,. \\ \quad \text{where} \quad \mathit{fd} \neq \mathit{fd}' \quad \text{(not same declaration)}, \\ \qquad\qquad \mathit{fd} = f[\![\overrightarrow{\alpha \texttt{ extends } N}]\!](\overrightarrow{\_:\tau}):\tau^r = \_\,, \quad \mathit{fd}' = f[\![\overrightarrow{\alpha' \texttt{ extends } N'}]\!](\overrightarrow{\_:\tau'}):\tau^{r'} = \_\,, \\ p \vdash \mathit{valid}([\![\overrightarrow{\alpha \texttt{ extends } N}]\!]\text{Object}\,\overrightarrow{\tau} \to \tau^r, [\![\overrightarrow{\alpha' \texttt{ extends } N'}]\!]\text{Object}\,\overrightarrow{\tau'} \to \tau^{r'}, \{(\mathit{fd}, \text{Object}) \mid \mathit{fd} \in \overrightarrow{d}\,\}) \end{array}}{p \vdash \mathit{validFun}(\overrightarrow{d}\,)}$$

Valid declarations: $\boxed{p \vdash \mathit{valid}([\![\overrightarrow{\alpha \texttt{ extends } N}]\!]\,\overrightarrow{\tau} \to \tau, [\![\overrightarrow{\alpha \texttt{ extends } N}]\!]\,\overrightarrow{\tau} \to \tau, \{\overrightarrow{(\mathit{fd}, \tau)}\})}$

$$[\textsc{Valid}]\quad \frac{\begin{array}{l} \Delta = \overrightarrow{\alpha \;<:\; N}, \quad |\overrightarrow{\tau}| = n \\ \quad 1. \qquad |\overrightarrow{\tau}| \neq |\overrightarrow{\tau'}| \\ \vee \quad 2. \qquad 1)\ \overrightarrow{N} = [\overrightarrow{\alpha}/\overrightarrow{\alpha'}]\overrightarrow{N'} \\ \qquad\quad \wedge \quad 2)\ \forall\, 1 \leq i \leq n.\ p;\Delta \vdash \tau_i \;<:\; [\overrightarrow{\alpha}/\overrightarrow{\alpha'}]\tau'_i \quad \vee \quad p;\Delta \vdash [\overrightarrow{\alpha}/\overrightarrow{\alpha'}]\tau'_i \;<:\; \tau_i \\ \qquad\quad \wedge \quad 3)\ \exists\, 1 \leq i \leq n.\ \tau_i \neq [\overrightarrow{\alpha}/\overrightarrow{\alpha'}]\tau'_i \\ \qquad\quad \wedge \quad 4)\ \exists\, (f[\![\overrightarrow{\alpha'' \texttt{ extends } N''}]\!](\overrightarrow{\_:\tau''}):\tau^{r''} = \_\,, \tau''_0) \in S. \\ \qquad\qquad\quad \text{where} \\ \qquad\qquad\qquad \forall\, 0 \leq i \leq n. \quad p;\Delta \vdash \mathit{meet}(\{\tau_i, [\overrightarrow{\alpha}/\overrightarrow{\alpha'}]\tau'_i\}, [\overrightarrow{\alpha}/\overrightarrow{\alpha''}]\tau''_i) \\ \qquad\qquad\quad \wedge \quad \overrightarrow{N} = [\overrightarrow{\alpha}/\overrightarrow{\alpha''}]\overrightarrow{N''} \\ \qquad\qquad\quad \wedge \quad p;\Delta \vdash [\overrightarrow{\alpha}/\overrightarrow{\alpha''}]\tau^{r''} \;<:\; \tau^r \\ \qquad\qquad\quad \wedge \quad p;\Delta \vdash [\overrightarrow{\alpha'}/\overrightarrow{\alpha''}]\tau^{r''} \;<:\; \tau^{r'} \end{array}}{p \vdash \mathit{valid}([\![\overrightarrow{\alpha \texttt{ extends } N}]\!]\,\overrightarrow{\tau} \to \tau^r, [\![\overrightarrow{\alpha' \texttt{ extends } N'}]\!]\,\overrightarrow{\tau'} \to \tau^{r'}, S)}$$

Expression typing: $\boxed{p;\Delta;\Gamma \vdash e : \tau}$

$$[\text{T-}\textsc{Var}]\qquad p;\Delta;\Gamma \vdash x : \Gamma(x)$$

$$[\text{T-}\textsc{Self}]\qquad p;\Delta;\Gamma \vdash \texttt{self} : \Gamma(\texttt{self})$$

$$[\text{T-}\textsc{Object}]\quad \frac{\begin{array}{c} \texttt{object}\ O[\![\overrightarrow{\alpha \texttt{ extends } \_}]\!](\overrightarrow{\_:\tau'})\ \_\ \texttt{end} \in p \qquad p;\Delta \vdash O[\![\overrightarrow{\tau}]\!]\,\mathsf{ok} \\ p;\Delta;\Gamma \vdash \overrightarrow{e} : \overrightarrow{\tau''} \qquad p;\Delta \vdash \overrightarrow{\tau''} \;<:\; [\overrightarrow{\tau}/\overrightarrow{\alpha}]\overrightarrow{\tau'} \end{array}}{p;\Delta;\Gamma \vdash O[\![\overrightarrow{\tau}]\!](\overrightarrow{e}) : O[\![\overrightarrow{\tau}]\!]}$$

$$[\text{T-}\textsc{Field}]\quad \frac{p;\Delta;\Gamma \vdash e_0 : \tau_0 \qquad \mathit{bound}_\Delta(\tau_0) = O[\![\overrightarrow{\tau'}]\!] \qquad \texttt{object}\ O[\![\overrightarrow{\alpha \texttt{ extends } \_}]\!](\overrightarrow{x:\tau})\ \_\ \texttt{end} \in p}{p;\Delta;\Gamma \vdash e_0\,\texttt{.}\,x_i : [\overrightarrow{\tau'}/\overrightarrow{\alpha}]\tau_i}$$

$$[\text{T-}\textsc{Method}]\quad \frac{\begin{array}{c} p;\Delta;\Gamma \vdash e_0 : \tau_0 \qquad p;\Delta \vdash \overrightarrow{\tau}\ \mathsf{ok} \qquad p;\Delta;\Gamma \vdash \overrightarrow{e} : \overrightarrow{\tau'} \\ \mathit{mostspecific}_{p;\Delta}(\mathit{applicable}_{p;\Delta}(f[\![\overrightarrow{\tau}]\!](\overrightarrow{\tau'}), \mathit{visible}_p(\mathit{bound}_\Delta(\tau_0)))) = f[\![\overrightarrow{\alpha \texttt{ extends } N}]\!](\_):\tau^r\ \_ \end{array}}{p;\Delta;\Gamma \vdash e_0.f[\![\overrightarrow{\tau}]\!](\overrightarrow{e}) : \tau^r}$$

$$[\text{T-}\textsc{Function}]\quad \frac{\begin{array}{c} p;\Delta \vdash \overrightarrow{\tau}\ \mathsf{ok} \qquad p;\Delta;\Gamma \vdash \overrightarrow{e} : \overrightarrow{\tau'} \\ \mathit{mostspecific}_{p;\Delta}(\mathit{applicable}_{p;\Delta}(f[\![\overrightarrow{\tau}]\!](\overrightarrow{\tau'}), \{(\mathit{fd}, \text{Object}) \mid \mathit{fd} \in p\})) = f[\![\overrightarrow{\alpha \texttt{ extends } N}]\!](\_):\tau^r\ \_ \end{array}}{p;\Delta;\Gamma \vdash f[\![\overrightarrow{\tau}]\!](\overrightarrow{e}) : \tau^r}$$

Figure A.9: Static Semantics of Core Fortress with Overloading (II)

Subtyping: $\boxed{p; \Delta \vdash \tau \ <: \ \tau}$

[S-OBJ] $$p; \Delta \vdash \tau \ <: \ \text{Object}$$

[S-REFL] $$p; \Delta \vdash \tau \ <: \ \tau$$

[S-TRANS] $$\frac{p; \Delta \vdash \tau_1 \ <: \ \tau_2 \qquad p; \Delta \vdash \tau_2 \ <: \ \tau_3}{p; \Delta \vdash \tau_1 \ <: \ \tau_3}$$

[S-VAR] $$p; \Delta \vdash \alpha \ <: \ \Delta(\alpha)$$

[S-TAPP] $$\frac{\_ \ C[\![\overrightarrow{\alpha \ \texttt{extends} \ \_}]\!]\_ \ \ \texttt{extends} \ \{\overrightarrow{N}\} \ \_ \in p}{p; \Delta \vdash C[\![\overrightarrow{\tau}]\!] \ <: \ [\overrightarrow{\tau}/\overrightarrow{\alpha}]N_i}$$

Well-formed types: $\boxed{p; \Delta \vdash \tau \ \mathsf{ok}}$

[W-OBJ] $$p; \Delta \vdash \text{Object} \ \mathsf{ok}$$

[W-VAR] $$\frac{\alpha \in \mathit{dom}(\Delta)}{p; \Delta \vdash \alpha \ \mathsf{ok}}$$

[W-TAPP] $$\frac{\_ \ C[\![\overrightarrow{\alpha \ \texttt{extends} \ N}]\!]\_ \in p \qquad p; \Delta \vdash \overrightarrow{\tau} \ \mathsf{ok} \qquad p; \Delta \vdash \overrightarrow{\tau} \ <: \ [\overrightarrow{\tau}/\overrightarrow{\alpha}]\overrightarrow{N}}{p; \Delta \vdash C[\![\overrightarrow{\tau}]\!] \ \mathsf{ok}}$$

Most specific definitions: $\boxed{\mathit{mostspecific}_{p;\Delta}(\{\overrightarrow{(\mathit{fd}, \tau)}\}) = \mathit{fd}}$

$$\frac{\begin{array}{c}\overrightarrow{\mathit{fd}} = f[\![(\overrightarrow{\alpha \ \texttt{extends} \ N})_1]\!]((\overrightarrow{\_ : \tau^a})_1) : \tau^r_1 \ \_ \ \cdots f[\![(\overrightarrow{\alpha \ \texttt{extends} \ N})_n]\!]((\overrightarrow{\_ : \tau^a})_n) : \tau^r_n \ \_ \\ 1 \le i \le n \qquad (\overrightarrow{\tau^a})_i = \tau^a_{i1} \cdots \tau^a_{im} \qquad \tau^a_{i0} = \tau_i \qquad \forall \, 0 \le j \le m \, . \, p; \Delta \vdash \mathit{meet}(\{\tau^a_{1j}, \cdots, \tau^a_{nj}\}, \tau^a_{ij})\end{array}}{\mathit{mostspecific}_{p;\Delta}(\{\overrightarrow{(\mathit{fd}, \tau)}\}) = \mathit{fd}_i}$$

Applicable definitions: $\boxed{\mathit{applicable}_{p;\Delta}(f[\![\overrightarrow{\tau}]\!](\overrightarrow{\tau}), \{\overrightarrow{(\mathit{fd}, \tau)}\}) = \{\overrightarrow{(\mathit{fd}, \tau)}\}}$

$$\mathit{applicable}_{p;\Delta}(f[\![\overrightarrow{\tau'}]\!](\overrightarrow{\tau''}), S) = \left\{ \begin{array}{ll} \{([\overrightarrow{\tau'}/\overrightarrow{\alpha}]\mathit{fd}, \tau) \mid & \text{where } (\mathit{fd}, \tau) \in S, \\ & \mathit{fd} = f[\![\overrightarrow{\alpha \ \texttt{extends} \ N}]\!](\overrightarrow{x{:}\tau'''}) : \_ \, , \\ & p; \Delta \vdash \overrightarrow{\tau''} \ <: \ [\overrightarrow{\tau'}/\overrightarrow{\alpha}]\overrightarrow{\tau'''} \\ & p; \Delta \vdash \overrightarrow{\tau'} \ <: \ [\overrightarrow{\tau'}/\overrightarrow{\alpha}]\overrightarrow{N}\} \end{array} \right.$$

Figure A.10: Static Semantics of Core Fortress with Overloading (III)

Method definition lookup: $\boxed{\mathit{visible}_p \text{ / } \mathit{defined}_p(C[\![\overrightarrow{\tau}]\!]) = \{\overrightarrow{(\mathit{fd}, C[\![\overrightarrow{\tau}]\!])}\}}$

$$\mathit{visible}_p(C[\![\overrightarrow{\tau}]\!]) = \mathit{defined}_p(C[\![\overrightarrow{\tau}]\!]) \cup \bigcup_{C'[\![\overrightarrow{\tau'}]\!] \in \{\overrightarrow{N}\}} \mathit{visible}_p([\overrightarrow{\tau}/\overrightarrow{\alpha}]C'[\![\overrightarrow{\tau'}]\!]) \quad \text{where } \_\ C[\![\overrightarrow{\alpha\ \_}]\!]\ \_\ \texttt{extends}\ \{\overrightarrow{N}\}\ \_\ \in p$$

$$\mathit{defined}_p(C[\![\overrightarrow{\tau}]\!]) = \{\overrightarrow{([\overrightarrow{\tau}/\overrightarrow{\alpha}]\mathit{fd}, C[\![\overrightarrow{\tau}]\!])}\} \quad \text{where } \_\ C[\![\overrightarrow{\alpha\ \_}]\!]\ \_\ \overrightarrow{\mathit{fd}}\ \_\ \in p$$

Most specific type: $\boxed{p; \Delta \vdash \mathit{meet}(\{\overrightarrow{\tau}\}, \tau)}$

$$[\text{MEET}] \quad \frac{\tau' \in \{\overrightarrow{\tau}\} \qquad \forall\, 1 \leq i \leq |\overrightarrow{\tau}|\ .\ p; \Delta \vdash \tau' <: \tau_i}{p; \Delta \vdash \mathit{meet}(\{\overrightarrow{\tau}\}, \tau')}$$

Bound of type: $\boxed{\mathit{bound}_\Delta(\tau) = \sigma}$

$$\begin{array}{lcl} \mathit{bound}_\Delta(\alpha) & = & \Delta(\alpha) \\ \mathit{bound}_\Delta(\sigma) & = & \sigma \end{array}$$

Figure A.11: Static Semantics of Core Fortress with Overloading (IV)

# Appendix B

# Rendering of Fortress Code

In order to more closely approximate mathematical notation, Fortress mandates standard rendering for various input elements, particularly for numerals and identifiers, as specified in Section 4.16.

In this appendix, we describe the detailed rules for rendering an identifier, as well as rules used for rendering other constructions.

## B.1 Rendering of Fortress Identifiers

If an identifier consists of letters and possibly digits, but no underscores, prime marks, or apostrophes, then the rules are fairly simple:

(a) If the identifier consists of two ASCII capital letters that are the same, possibly followed by digits, then a single capital letter is rendered double-struck, followed by full-sized (not subscripted) digits in roman font.

| | | | | | |
|---|---|---|---|---|---|
| `QQ` | *is rendered as* | $\mathbb{Q}$ | `RR64` | *is rendered as* | $\mathbb{R}64$ |
| `ZZ` | *is rendered as* | $\mathbb{Z}$ | `ZZ512` | *is rendered as* | $\mathbb{Z}512$ |

(b) Otherwise, if the identifier has more than two characters and begins with a capital letter, then it is rendered in roman font. (Such names are typically used as names of types in Fortress. Note that an identifier cannot consist entirely of capital letters, because such a token is considered to be an operator.)

| | | | | | |
|---|---|---|---|---|---|
| `Integer` | *is rendered as* | Integer | `Matrix` | *is rendered as* | $\mathrm{Matrix}$ |
| `TotalOrder` | *is rendered as* | TotalOrder | `BooleanAlgebra` | *is rendered as* | $\mathrm{BooleanAlgebra}$ |
| `Fred17` | *is rendered as* | Fred17 | `R2D2` | *is rendered as* | $\mathrm{R2D2}$ |

(c) Otherwise, if the identifier consists of one or more letters followed by one or more digits, then the letters are rendered in italic and the digits are rendered as roman subscripts.

| | | | | | |
|---|---|---|---|---|---|
| `a3` | *is rendered as* | $a_3$ | `foo7` | *is rendered as* | $foo_7$ |
| `M1` | *is rendered as* | $M_1$ | `z128` | *is rendered as* | $z_{128}$ |
| `OMEGA13` | *is rendered as* | $\Omega_{13}$ | `myFavoriteThings1625` | *is rendered as* | $myFavoriteThings_{1625}$ |

(d) The following names are always rendered in roman type out of respect for tradition:

| | | | | | |
|---|---|---|---|---|---|
| sin | cos | tan | cot | sec | csc |
| sinh | cosh | tanh | coth | sech | csch |
| arcsin | arccos | arctan | arccot | arcsec | arccsc |
| arsinh | arcosh | artanh | arcoth | arsech | arcsch |
| arg | deg | det | exp | inf | sup |
| lg | ln | log | gcd | max | min |

(e) Otherwise the identifier is simply rendered entirely in italic type.

| | | | | | |
|---|---|---|---|---|---|
| `a` | *is rendered as* | $a$ | `foobar` | *is rendered as* | $foobar$ |
| `length` | *is rendered as* | $length$ | `isInstanceOf` | *is rendered as* | $isInstanceOf$ |
| `foo7a` | *is rendered as* | $foo7a$ | `l33tsp33k` | *is rendered as* | $l33tsp33k$ |

If the identifier begins or ends with an underscore, or both, but has no other underscores, or prime marks, or apostrophes:

(f) If the identifier, ignoring its underscores, consists of two ASCII capital letters that are the same, possibly followed by one or more digits, then a single capital letter is rendered in sans-serif (for a leading underscore), script (for a trailing underscore), or italic san-serif (for both a leading and a trailing underscore), and any digits are rendered as roman subscripts.

(g) Otherwise, the identifier without its underscores is rendered in boldface (for a leading underscore), roman (for a trailing underscore), or bold italic (for both a leading and a trailing underscore); except that if the identifier, ignoring its underscores, consists of one or more letters followed by one or more digits, then the digits are rendered as roman subscripts regardless of the underscores.

| | | | | | |
|---|---|---|---|---|---|
| `m_` | *is rendered as* | m | `s_` | *is rendered as* | s |
| `km_` | *is rendered as* | km | `kg_` | *is rendered as* | kg |
| `V_` | *is rendered as* | V | `kW_` | *is rendered as* | kW |
| `_v` | *is rendered as* | $\mathbf{v}$ | `_foo13` | *is rendered as* | $\mathbf{foo}_{13}$ |

(Roman identifiers are typically used for names of SI dimensional units. See sections 6.1.1 and 6.2.1 of [21] for style questions with respect to dimensions and units.)

These last two rules are actually special cases of the following general rules that apply whenever an identifier contains at least one underscore, prime mark, or apostrophe:

An identifier containing underscores is divided into portions by its underscores; in addition, any apostrophe, prime, or double prime character separates portions and is also itself a portion.

(h) If any portion is empty other than the first or last, then the entire identifier is rendered in italics, underscores and all.

Otherwise, the portions are rendered as follows. The idea is that there is a *principal portion* that may be preceded and/or followed by modifiers, and there may also be a *face portion*:

- If the first portion is not empty, `script`, `fraktur`, `sansserif`, or `monospace`, then the principal portion is the first portion and there is no face portion.
- If the first portion is `script`, `fraktur`, `sansserif`, or `monospace`, then the principal portion is the second portion and the face portion is the first portion.
- If the first portion is empty and the second portion is not `script`, `fraktur`, `sansserif`, or `monospace`, then the principal portion is the second portion and there is no face portion.
- Otherwise the principal portion is the third portion and the face portion is the second portion.

If there is no face portion, the principal portion will be rendered in ordinary italics. However, if the first portion is empty (that is, the identifier begins with a leading underscore), then the principal portion is to be rendered in roman boldface. If the last portion is empty (that is, the identifier ends with a trailing underscore), then the principal portion will be roman rather than italic, or bold italic rather than bold.

If there is a face portion, then that describes an alternate typeface to be used in rendering the principal portion. If there is no face portion, but the principal portion consists of two copies of the same letter, then it is rendered as a single letter in a double-struck face (also known as "blackboard bold"), sans-serif, script, or italic sans-serif font according to whether the first and last portions are (not empty, not empty), (empty, not empty), (not empty, empty), or (empty, empty), respectively. Otherwise, if the first portion is empty (that is, the identifier begins with a leading underscore), then the principal portion is to be rendered in a bold version of the selected face, and if the last portion is empty (that is, the identifier ends with a trailing underscore), then the principal portion to be rendered in an italic (or bold italic) version of the selected face. The bold and italic modifiers may be used only in combination with certain faces; the following are the allowed combinations:

```
script
bold script
fraktur
bold fraktur
double-struck
sans-serif
bold sans-serif
italic sans-serif
bold italic sans-serif
monospace
```

If a combination can be properly rendered, then the principal portion is rendered but not any preceding portions or underscores. If a combination cannot be properly rendered, then the principal portion and all portions and underscores preceding it are rendered all in italics if possible, and otherwise all in some other default face.

If the principal portion consists of a sequence of letters followed by a sequence of digits, then the letters are rendered in the chosen face and the digits are rendered as roman subscripts. Otherwise the entire principal portion is rendered in the chosen face. The remaining portions (excepting the last, if it is empty) are then processed according to the following rules:

- If a portion is `bar`, then a bar is rendered above what has already been rendered, excluding superscripts and subscripts. For example, `x_bar` is rendered as $\bar{x}$, `x17_bar` is rendered as $\bar{x}_{17}$, `x_bar_bar` is rendered as $\bar{\bar{x}}$, and `foo_bar` is rendered as $\overline{foo}$. (Contrast this last with `foo_baz`, which is rendered as *foo_baz*.)
- If a portion is `vec`, then a right-pointing arrow is rendered above what has already been rendered, excluding superscripts and subscripts. For example, `v_vec` is rendered as $\vec{v}$, `v17_vec` is rendered as $\vec{v}_{17}$, and `zoom_vec` is rendered as $\overrightarrow{zoom}$.
- If a portion is `hat`, then a hat is rendered above what has already been rendered, excluding superscripts and subscripts. For example, `x17_hat` is rendered as $\hat{x}_{17}$.
- If a portion is `dot`, then a dot is rendered above what has already been rendered, excluding superscripts and subscripts; but if the preceding portion was also `dot`, then the new dot is rendered appropriately relative to the previous dot(s). Up to four dots will be rendered side-by-side rather than vertically. For example, `a_dot` is rendered as $\dot{a}$, `a_dot_dot` is rendered as $\ddot{a}$, `a_dot_dot_dot` is rendered as $\dddot{a}$, `a_dot_dot_dot_dot` is rendered as $\ddddot{a}$. Also, `a_vec_dot` is rendered as $\dot{\vec{v}}$.
- If a portion is `star`, then an asterisk `*` is rendered as a superscript. For example, `a_star` is rendered as $a^*$, `a_star_star` is rendered as $a^{**}$, `ZZ_star` is rendered as $\mathbb{Z}^*$.
- If a portion is `splat`, then a number sign `#` is rendered as a superscript. For example, `QQ_splat` is rendered as $\mathbb{Q}^\#$.
- If a portion is `prime`, then a prime mark is rendered as a superscript.

- A prime character is treated the same as `prime`, and a double prime character is treated the same as two consecutive `prime` portions. An apostrophe is treated the same as a prime character, but only if all characters following it in the identifier, if any, are also apostrophes. For example, `a'` is rendered as $a'$, `a13'` is rendered as $a'_{13}$, and `a''` is rendered as $a''$, but `don't` is rendered as *don't*.
- If a portion is `super` and another portion follows, then that other portion is rendered as a superscript in roman type, and enclosed in parentheses if it is all digits.
- If a portion is `sub` and another portion follows, then that other portion is rendered as a subscript in roman type, and enclosed in parentheses if it is all digits, and preceded by a subscript-separating comma if this portion was immediately preceded by another portion that was rendered as a subscript.
- If a portion consists entirely of capital letters and would, if considered by itself as an identifier, be the name of a non-letter Unicode character that would be subject to replacement by preprocessing, then that Unicode character is rendered as a subscript. For example, `id_OPLUS` is rendered as $id_{\oplus}$, `ZZ_GT` is rendered as $\mathbb{Z}_{>}$, and `QQ_star_LE` is rendered as $\mathbb{Q}^{*}_{\leq}$.
- If the portion is the last portion, and the principal portion was a single letter (or two letters indicating a double-struck letter), and none of the preceding rules in this list applies, it is rendered as a subscript in roman type. For example, `T_min` is rendered as $T_{\mathrm{min}}$. Note that `T_MAX` is rendered simply as `T_MAX`—because all its letters are capital letters, it is considered to be an operator—but `T_sub_MAX` is rendered as $T_{\mathrm{MAX}}$.
- Otherwise, this portion and all succeeding portions are rendered in italics, along with any underscores that appear adjacent to any of them.

Examples:

| | | | | | |
|---|---|---|---|---|---|
| `M` | *is rendered as* | $M$ | `_M` | *is rendered as* | $\mathbf{M}$ |
| `v_vec` | *is rendered as* | $\vec{v}$ | `_v_vec` | *is rendered as* | $\vec{\mathbf{v}}$ |
| `v1` | *is rendered as* | $v_1$ | `v_x` | *is rendered as* | $v_{\mathrm{x}}$ |
| `_v1` | *is rendered as* | $\mathbf{v}_1$ | `_v_x` | *is rendered as* | $\mathbf{v}_{\mathrm{x}}$ |
| `a_dot` | *is rendered as* | $\dot{a}$ | `a_dot_dot` | *is rendered as* | $\ddot{a}$ |
| `a_dot_dot_dot` | *is rendered as* | $\dddot{a}$ | `a_dot_dot_dot_dot` | *is rendered as* | $\ddddot{a}$ |
| `a_dot_dot_dot_dot_dot` | *is rendered as* | $\overset{\cdot\cdot\cdot\cdot\cdot}{a}$ | `p13'` | *is rendered as* | $p'_{13}$ |
| `p'` | *is rendered as* | $p'$ | `p_prime` | *is rendered as* | $p'$ |
| `T_min` | *is rendered as* | $T_{\mathrm{min}}$ | `T_max` | *is rendered as* | $T_{\mathrm{max}}$ |
| `foo_bar` | *is rendered as* | $\overline{\mathit{foo}}$ | `foo_baz` | *is rendered as* | *foo_baz* |

In this way, through the use of underscore characters and annotation portions delimited by underscores, the programmer can exercise considerable typographical control over the rendering of variable names; but if no underscores are used, the rendering rules are quite simple.

## B.2 Rendering of Other Fortress Constructs

An expression of the form `a^b` or `a^(b)` is rendered by making the expression $b$ a superscript, and possibly removing an outer set of parentheses, if it is reasonably simple; otherwise it is rendered as if `^` were an ordinary binary operator.

| | | |
|---|---|---|
| `x^y` | *is rendered as* | $x^y$ |
| `x^43` | *is rendered as* | $x^{43}$ |
| `x^(n+1)` | *is rendered as* | $x^{n+1}$ |
| `x^(\|s.substring(a,b)\|)` | *is rendered as* | $x\hat{}(\lvert s.\mathit{substring}(a,b)\rvert)$ |

An expression of the form `a[b]` is rendered by making the expression $b$ a subscript, if it is reasonably simple.

| | | |
|---|---|---|
| `a[43]` | *is rendered as* | $a_{43}$ |
| `a[k]` | *is rendered as* | $a_k$ |
| `a[n_max]` | *is rendered as* | $a_{n_{\max}}$ |
| `a[k+k']` | *is rendered as* | $a_{k+k'}$ |
| `a[b-c]` | *is rendered as* | $a_{b-c}$ |
| `a[3 k + 1]` | *is rendered as* | $a_{3k+1}$ |
| `a[b[c[d]]]` | *is rendered as* | $a[b[c_d]]$ |
| `a[s.substring(p,q)]` | *is rendered as* | $a[s.\mathit{substring}(p,q)]$ |

If there is more than one subscript expression, then they are rendered as subscripts only if each subscript is a single letter or digit, or a single letter followed by digits and/or prime marks:

| | | |
|---|---|---|
| `a[1,1]` | *is rendered as* | $a_{11}$ |
| `a[1,n]` | *is rendered as* | $a_{1n}$ |
| `a[j,m,n]` | *is rendered as* | $a_{jmn}$ |
| `a[n1,n2,n3]` | *is rendered as* | $a_{n_1n_2n_3}$ |
| `a[k,k',k'']` | *is rendered as* | $a_{kk'k''}$ |
| `a[1,n-1]` | *is rendered as* | $a[1,n-1]$ |

(There is, of course, some opportunity here to make the rules for subscripted rendering more elaborate.)

If there is whitespace adjacent to the subscripting brackets or any separating comma, then actual subscripts are not used and the brackets remain (so putting spaces after the commas of a multidimensional subscript is a simple way to guarantee that bracket syntax will be used in the formatted version):

| | | |
|---|---|---|
| `a[i,j]` | *is rendered as* | $a_{ij}$ |
| `a[i, j]` | *is rendered as* | $a[i,j]$ |
| `a[b[c[ d ]]]` | *is rendered as* | $a[b[c[\,d\,]]]$ |

The presence or absence of whitespace inside and adjacent to enclosing operators is reflected in the rendered form by the presence or absence of a thin space:

| | | |
|---|---|---|
| `a[s.substring(p,q)]` | *is rendered as* | $a[s.\mathit{substring}(p,q)]$ |
| `a[ s.substring(p,q) ]` | *is rendered as* | $a[\,s.\mathit{substring}(p,q)\,]$ |
| `{1,2}` | *is rendered as* | $\{1,2\}$ |
| `{ 1, 2 }` | *is rendered as* | $\{\,1,2\,\}$ |
| `<\|1,2,3\|>` | *is rendered as* | $\langle 1,2,3\rangle$ |
| `<\| x+y \| x<-a, y<-b \|>` | *is rendered as* | $\langle\, x+y \mid x\leftarrow a, y\leftarrow b\,\rangle$ |

Rendered comprehensions look best if there is whitespace inside the enclosers and on both sides of the separating vertical bar.

The presence or absence of whitespace on either side of a colon is also handled especially carefully. In general, one should put a space on both sides, or neither side, of a colon that is used as an operator to construct a range; on the other hand, one should put a space after, but *not* before, a colon that separates a variable or function header from a type. This will produce the best rendered spacing. (The assignment operator $:=$ is recognized separately and whitespace doesn't matter.)

| | | |
|---|---|---|
| `for i <- 1:10 do print i end` | *is rendered as* | `for` $i\leftarrow 1\!:\!10$ `do` $\mathit{print}\ i$ `end` |
| `for i <- 0 : n+1 do print i end` | *is rendered as* | `for` $i\leftarrow 0:n+1$ `do` $\mathit{print}\ i$ `end` |
| `k: ZZ32 := 5` | *is rendered as* | $k\!:\mathbb{Z}32 := 5$ |
| `factorial(n: NN): NN` | *is rendered as* | $\mathit{factorial}(n\!:\mathbb{N})\!:\mathbb{N}$ |
| `a:=b` | *is rendered as* | $a := b$ |

Generators and filters in brackets after a reduction operator are rendered by stacking them beneath the operator:

`PROD[k<-1#n] n` *is rendered as* $\prod_{k \leftarrow 1 \# n} n$

`SUM[i<-1:n, j<-1:p, prime j] a[i] x^j` *is rendered as* $\sum_{\substack{i \leftarrow 1:n \\ j \leftarrow 1:p \\ prime\ j}} a_i\, x^j$

`MAX[j<-S, k<-j:j+m] a[k]` *is rendered as* $\underset{\substack{j \leftarrow S \\ k \leftarrow j:j+m}}{\mathtt{MAX}} \quad a_k$

# Appendix C

# Operator Precedence, Associativity, Chaining, and Enclosure

This appendix contains the detailed rules for which Unicode 5.0 characters may be used as operators, which operators form enclosing pairs, which operators may be chained, which operators are associative, and what precedence relationships exist among the various operators. An infix operator is left associative, nonassociative, or chainable. If no precedence relationship is stated explicitly for any given pair of operators, then there is no precedence relationship between those two operators. Remember that precedence is not transitive in Fortress.

In each of the character lists below, each line gives the Unicode codepoint, the full Unicode 5.0 name, a rendering of the character in TEX (if possible), and any ASCII shorthand or short names for the character.

## C.1 Bracket Pairs for Enclosing Operators

Here are the bracket pairs that may be used as enclosing operators. Note that there is one group of four brackets; within that group, either left bracket may be paired with either right bracket to form an enclosing operator.

| | | | |
|---|---|---|---|
| U+005B | LEFT SQUARE BRACKET | [ | [ |
| U+005D | RIGHT SQUARE BRACKET | ] | ] |
| U+007B | LEFT CURLY BRACKET | { | { |
| U+007D | RIGHT CURLY BRACKET | } | } |
| U+2045 | LEFT SQUARE BRACKET WITH QUILL | | [./ |
| U+2046 | RIGHT SQUARE BRACKET WITH QUILL | | /.] |
| U+2308 | LEFT CEILING | ⌈ | \|/ |
| U+2309 | RIGHT CEILING | ⌉ | \\| |
| U+230A | LEFT FLOOR | ⌊ | \|\ |
| U+230B | RIGHT FLOOR | ⌋ | /\| |
| U+27C5 | LEFT S-SHAPED BAG DELIMITER | | \|.\ |
| U+27C6 | RIGHT S-SHAPED BAG DELIMITER | | /.\| |
| U+27E8 | MATHEMATICAL LEFT ANGLE BRACKET | ⟨ | <\| |
| U+27E9 | MATHEMATICAL RIGHT ANGLE BRACKET | ⟩ | \|> |
| U+27EA | MATHEMATICAL LEFT DOUBLE ANGLE BRACKET | ⟪ | <<\| |
| U+27EB | MATHEMATICAL RIGHT DOUBLE ANGLE BRACKET | ⟫ | \|>> |

| | | | |
|---|---|---|---|
| U+2983 | LEFT WHITE CURLY BRACKET | ⦃ | {\ |
| U+2984 | RIGHT WHITE CURLY BRACKET | ⦄ | \} |
| U+2985 | LEFT WHITE PARENTHESIS | | (\ |
| U+2986 | RIGHT WHITE PARENTHESIS | | \) |
| U+2987 | Z NOTATION LEFT IMAGE BRACKET | | (/ |
| U+2988 | Z NOTATION RIGHT IMAGE BRACKET | | /) |
| U+2989 | Z NOTATION LEFT BINDING BRACKET | | <\|\| |
| U+298A | Z NOTATION RIGHT BINDING BRACKET | | \|\|> |
| U+298B | LEFT SQUARE BRACKET WITH UNDERBAR | | [.\ |
| U+298C | RIGHT SQUARE BRACKET WITH UNDERBAR | | \.] |
| U+298D | LEFT SQUARE BRACKET WITH TICK IN TOP CORNER | | [.// |
| U+298E | RIGHT SQUARE BRACKET WITH TICK IN BOTTOM CORNER | | //.] |
| U+298F | LEFT SQUARE BRACKET WITH TICK IN BOTTOM CORNER | | [.\\ |
| U+2990 | RIGHT SQUARE BRACKET WITH TICK IN TOP CORNER | | \\.] |
| U+2991 | LEFT ANGLE BRACKET WITH DOT | | <.\| |
| U+2992 | RIGHT ANGLE BRACKET WITH DOT | | \|.> |
| U+2993 | LEFT ARC LESS-THAN BRACKET | | (< |
| U+2994 | RIGHT ARC GREATER-THAN BRACKET | | >) |
| U+2995 | DOUBLE LEFT ARC GREATER-THAN BRACKET | | ((> |
| U+2996 | DOUBLE RIGHT ARC LESS-THAN BRACKET | | <)) |
| U+2997 | LEFT BLACK TORTOISE SHELL BRACKET | | [*/ |
| U+2998 | RIGHT BLACK TORTOISE SHELL BRACKET | | /*] |
| U+29D8 | LEFT WIGGLY FENCE | | [/\/ |
| U+29D9 | RIGHT WIGGLY FENCE | | /\/] |
| U+29DA | LEFT DOUBLE WIGGLY FENCE | | [/\/\/ |
| U+29DB | RIGHT DOUBLE WIGGLY FENCE | | /\/\/] |
| U+29FC | LEFT-POINTING CURVED ANGLE BRACKET | | <\|\|\| |
| U+29FD | RIGHT-POINTING CURVED ANGLE BRACKET | | \|\|\|> |
| U+300C | LEFT CORNER BRACKET | ⌜ | </ |
| U+300D | RIGHT CORNER BRACKET | ⌝ | \> |
| U+300E | LEFT WHITE CORNER BRACKET | | <</ |
| U+300F | RIGHT WHITE CORNER BRACKET | | \>> |
| U+3010 | LEFT BLACK LENTICULAR BRACKET | | {*/ |
| U+3011 | RIGHT BLACK LENTICULAR BRACKET | | /*} |
| U+3018 | LEFT WHITE TORTOISE SHELL BRACKET | | [// |
| U+3014 | LEFT TORTOISE SHELL BRACKET | | [/ |
| U+3015 | RIGHT TORTOISE SHELL BRACKET | | /] |
| U+3019 | RIGHT WHITE TORTOISE SHELL BRACKET | | //] |
| U+3016 | LEFT WHITE LENTICULAR BRACKET | | {/ |
| U+3017 | RIGHT WHITE LENTICULAR BRACKET | | /} |

## C.2 Vertical-Line Operators

The following are vertical-line operators. They are left associative when they are used as infix operators.

| | | | |
|---|---|---|---|
| U+007C | VERTICAL LINE | \| | \| |
| U+2016 | DOUBLE VERTICAL LINE | ‖ | \|\| |
| U+2AF4 | TRIPLE VERTICAL BAR BINARY RELATION | | \|\|\| |

## C.3 Arithmetic Operators

### C.3.1 Multiplication and Division

The following are multiplication operators. They are left associative.

| | | | |
|---|---|---|---|
| U+002A | ASTERISK | * | * |
| U+00B7 | MIDDLE DOT | · | DOT |
| U+00D7 | MULTIPLICATION SIGN | × | TIMES BY |
| U+2217 | ASTERISK OPERATOR | ∗ | * |
| U+228D | MULTISET MULTIPLICATION | | |
| U+2297 | CIRCLED TIMES | ⊗ | OTIMES |
| U+2299 | CIRCLED DOT OPERATOR | ⊙ | ODOT |
| U+229B | CIRCLED ASTERISK OPERATOR | ⊛ | CIRCLEDAST |
| U+22A0 | SQUARED TIMES | ⊠ | BOXTIMES |
| U+22A1 | SQUARED DOT OPERATOR | ⊡ | BOXDOT |
| U+22C5 | DOT OPERATOR | ⋅ | |
| U+29C6 | SQUARED ASTERISK | | BOXAST |
| U+29D4 | TIMES WITH LEFT HALF BLACK | | |
| U+29D5 | TIMES WITH RIGHT HALF BLACK | | |
| U+2A2F | VECTOR OR CROSS PRODUCT | ⨯ | CROSS |
| U+2A30 | MULTIPLICATION SIGN WITH DOT ABOVE | | DOTTIMES |
| U+2A31 | MULTIPLICATION SIGN WITH UNDERBAR | | |
| U+2A34 | MULTIPLICATION SIGN IN LEFT HALF CIRCLE | | |
| U+2A35 | MULTIPLICATION SIGN IN RIGHT HALF CIRCLE | | |
| U+2A36 | CIRCLED MULTIPLICATION SIGN WITH CIRCUMFLEX ACCENT | | |
| U+2A37 | MULTIPLICATION SIGN IN DOUBLE CIRCLE | | |
| U+2A3B | MULTIPLICATION SIGN IN TRIANGLE | | TRITIMES |

The following are division operators. They are nonassociative.

| | | | |
|---|---|---|---|
| U+002F | SOLIDUS | / | / |
| U+00F7 | DIVISION SIGN | ÷ | DIV |
| U+2215 | DIVISION SLASH | ∕ | / |
| U+2298 | CIRCLED DIVISION SLASH | ⊘ | OSLASH |
| U+29B8 | CIRCLED REVERSE SOLIDUS | | |
| U+29BC | CIRCLED ANTICLOCKWISE-ROTATED DIVISION SIGN | | |
| U+29C4 | SQUARED RISING DIAGONAL SLASH | | BOXSLASH |
| U+29F5 | REVERSE SOLIDUS OPERATOR | ⧵ | \ |
| U+29F8 | BIG SOLIDUS | ⧸ | |
| U+29F9 | BIG REVERSE SOLIDUS | ⧹ | |
| U+2A38 | CIRCLED DIVISION SIGN | | ODIV |
| U+2AFD | DOUBLE SOLIDUS OPERATOR | ⫽ | // |
| U+2AFB | TRIPLE SOLIDUS BINARY RELATION | | /// |

Note also that `per` is treated as a division operator.

### C.3.2 Addition and Subtraction

Addition and subtraction operators are left associative.

The following three operators have the same precedence and may be mixed.

| | | | |
|---|---|---|---|
| U+002B | PLUS SIGN | + | + |
| U+002D | HYPHEN-MINUS | − | - |
| U+2212 | MINUS SIGN | − | - |

They each have lower precedence than any of the following multiplication and division operators:

| | | | |
|---|---|---|---|
| U+002A | ASTERISK | * | * |
| U+002F | SOLIDUS | / | / |
| U+00B7 | MIDDLE DOT | · | DOT |
| U+00D7 | MULTIPLICATION SIGN | × | TIMES BY |
| U+00F7 | DIVISION SIGN | ÷ | DIV |
| U+2215 | DIVISION SLASH | / | / |
| U+2217 | ASTERISK OPERATOR | ∗ | * |
| U+22C5 | DOT OPERATOR | ⋅ | |
| U+2A2F | VECTOR OR CROSS PRODUCT | × | CROSS |

The following two operators have the same precedence and may be mixed.

| | | | |
|---|---|---|---|
| U+2214 | DOT PLUS | ∔ | DOTPLUS |
| U+2238 | DOT MINUS | ∸ | DOTMINUS |

They each have lower precedence than this multiplication operator:

| | | | |
|---|---|---|---|
| U+2A30 | MULTIPLICATION SIGN WITH DOT ABOVE | | DOTTIMES |

The following two operators have the same precedence and may be mixed.

| | | | |
|---|---|---|---|
| U+2A25 | PLUS SIGN WITH DOT BELOW | | |
| U+2A2A | MINUS SIGN WITH DOT BELOW | | |

The following two operators have the same precedence and may be mixed.

| | | | |
|---|---|---|---|
| U+2A39 | PLUS SIGN IN TRIANGLE | | TRIPLUS |
| U+2A3A | MINUS SIGN IN TRIANGLE | | TRIMINUS |

They each have lower precedence than this multiplication operator:

| | | | |
|---|---|---|---|
| U+2A3B | MULTIPLICATION SIGN IN TRIANGLE | | TRITIMES |

The following two operators have the same precedence and may be mixed.

| | | | |
|---|---|---|---|
| U+2295 | CIRCLED PLUS | ⊕ | OPLUS |
| U+2296 | CIRCLED MINUS | ⊖ | OMINUS |

They each have lower precedence than any of the following multiplication and division operators:

| | | | |
|---|---|---|---|
| U+2297 | CIRCLED TIMES | ⊗ | OTIMES |
| U+2298 | CIRCLED DIVISION SLASH | ⊘ | OSLASH |
| U+2299 | CIRCLED DOT OPERATOR | ⊙ | ODOT |
| U+229B | CIRCLED ASTERISK OPERATOR | ⊛ | CIRCLEDAST |
| U+2A38 | CIRCLED DIVISION SIGN | | ODIV |

The following two operators have the same precedence and may be mixed.

| | | | |
|---|---|---|---|
| U+229E | SQUARED PLUS | ⊞ | BOXPLUS |
| U+229F | SQUARED MINUS | ⊟ | BOXMINUS |

They each have lower precedence than any of these multiplication or division operators:

| | | | |
|---|---|---|---|
| U+22A0 | SQUARED TIMES | ⊠ | BOXTIMES |
| U+22A1 | SQUARED DOT OPERATOR | ⊡ | BOXDOT |
| U+29C4 | SQUARED RISING DIAGONAL SLASH | | BOXSLASH |
| U+29C6 | SQUARED ASTERISK | | BOXAST |

These are other miscellaneous addition and subtraction operators:

| | | |
|---|---|---|
| U+00B1 | PLUS-MINUS SIGN | ± |
| U+2213 | MINUS-OR-PLUS SIGN | ∓ |
| U+2242 | MINUS TILDE | |
| U+2A22 | PLUS SIGN WITH SMALL CIRCLE ABOVE | ⨢ |
| U+2A23 | PLUS SIGN WITH CIRCUMFLEX ACCENT ABOVE | ⨣ |
| U+2A24 | PLUS SIGN WITH TILDE ABOVE | ⨤ |
| U+2A26 | PLUS SIGN WITH TILDE BELOW | ⨦ |
| U+2A27 | PLUS SIGN WITH SUBSCRIPT TWO | ⨧ |
| U+2A28 | PLUS SIGN WITH BLACK TRIANGLE | |
| U+2A29 | MINUS SIGN WITH COMMA ABOVE | ⨩ |
| U+2A2B | MINUS SIGN WITH FALLING DOTS | |
| U+2A2C | MINUS SIGN WITH RISING DOTS | |
| U+2A2D | PLUS SIGN IN LEFT HALF CIRCLE | |
| U+2A2E | PLUS SIGN IN RIGHT HALF CIRCLE | |

## C.3.3 Miscellaneous Arithmetic Operators

The operators MAX, MIN, REM, MOD, GCD, LCM, CHOOSE, and per, none of which corresponds to a single Unicode character, are considered to be arithmetic operators, having higher precedence than certain relational operators, as described in a later section. Among these operators, MAX, MIN, GCD, and LCM are left associative and REM, MOD, CHOOSE, and per are nonassociative.

## C.3.4 Set Intersection, Union, and Difference

The following are the set intersection operators. They are left associative.

| | | | |
|---|---|---|---|
| U+2229 | INTERSECTION | ∩ | CAP INTERSECT |
| U+22D2 | DOUBLE INTERSECTION | ⋒ | CAPCAP |
| U+2A40 | INTERSECTION WITH DOT | | |
| U+2A43 | INTERSECTION WITH OVERBAR | ⩃ | |
| U+2A44 | INTERSECTION WITH LOGICAL AND | | |
| U+2A4B | INTERSECTION BESIDE AND JOINED WITH INTERSECTION | | |
| U+2A4D | CLOSED INTERSECTION WITH SERIFS | | |
| U+2ADB | TRANSVERSAL INTERSECTION | | |

The following are the set union operators. They are left associative.

| | | | |
|---|---|---|---|
| U+222A | UNION | ∪ | CUP UNION |
| U+228E | MULTISET UNION | ⊎ | UPLUS |

| | | | |
|---|---|---|---|
| U+22D3 | DOUBLE UNION | ⋓ | CUPCUP |
| U+2A41 | UNION WITH MINUS SIGN | | |
| U+2A42 | UNION WITH OVERBAR | ⩂ | |
| U+2A45 | UNION WITH LOGICAL OR | | |
| U+2A4A | UNION BESIDE AND JOINED WITH UNION | | |
| U+2A4C | CLOSED UNION WITH SERIFS | | |
| U+2A50 | CLOSED UNION WITH SERIFS AND SMASH PRODUCT | | |

They each have lower precedence than any of the set intersection operators.

This is a miscellaneous set operator. It is nonassociative.

| | | | |
|---|---|---|---|
| U+2216 | SET MINUS | ∖ | SETMINUS |

### C.3.5 Square Arithmetic Operators

Square arithmetic operators are left associative.

The following are the square intersection operators.

| | | | |
|---|---|---|---|
| U+2293 | SQUARE CAP | ⊓ | SQCAP |
| U+2A4E | DOUBLE SQUARE INTERSECTION | | SQCAPCAP |

The following are the square union operators:

| | | | |
|---|---|---|---|
| U+2294 | SQUARE CUP | ⊔ | SQCUP |
| U+2A4F | DOUBLE SQUARE UNION | | SQCUPCUP |

They each have lower precedence than either of the square intersection operators.

### C.3.6 Curly Arithmetic Operators

Curly arithmetic operators are left associative.

The following is the curly intersection operator:

| | | | |
|---|---|---|---|
| U+22CF | CURLY LOGICAL AND | ⋏ | CURLYAND |

The following is the curly union operator:

| | | | |
|---|---|---|---|
| U+22CE | CURLY LOGICAL OR | ⋎ | CURLYOR |

It has lower precedence than the curly intersection operator.

## C.4 Relational Operators

### C.4.1 Equivalence and Inequivalence Operators

Every operator listed in this section has lower precedence than any operator listed in Section C.3.

The following are equivalence operators. They may be chained. Moreover, they may be chained with any other single group of chainable relational operators, as described in later sections.

| | | | |
|---|---|---|---|
| U+003D | EQUALS SIGN | = | EQ |
| U+2243 | ASYMPTOTICALLY EQUAL TO | ≃ | SIMEQ |
| U+2245 | APPROXIMATELY EQUAL TO | ≅ | |
| U+2246 | APPROXIMATELY BUT NOT ACTUALLY EQUAL TO | | |
| U+2248 | ALMOST EQUAL TO | ≈ | APPROX |
| U+224A | ALMOST EQUAL OR EQUAL TO | ≊ | APPROXEQ |
| U+224C | ALL EQUAL TO | | |
| U+224D | EQUIVALENT TO | ≍ | |
| U+224E | GEOMETRICALLY EQUIVALENT TO | ≎ | BUMPEQV |
| U+2251 | GEOMETRICALLY EQUAL TO | ≑ | DOTEQDOT |
| U+2252 | APPROXIMATELY EQUAL TO OR THE IMAGE OF | ≒ | |
| U+2253 | IMAGE OF OR APPROXIMATELY EQUAL TO | ≓ | |
| U+2256 | RING IN EQUAL TO | ≖ | EQRING |
| U+2257 | RING EQUAL TO | ≗ | RINGEQ |
| U+225B | STAR EQUALS | | |
| U+225C | DELTA EQUAL TO | ≜ | EQDEL |
| U+225D | EQUAL TO BY DEFINITION | | EQDEF |
| U+225F | QUESTIONED EQUAL TO | | |
| U+2261 | IDENTICAL TO | ≡ | EQV EQUIV |
| U+2263 | STRICTLY EQUIVALENT TO | | === SEQV |
| U+229C | CIRCLED EQUALS | | |
| U+22CD | REVERSED TILDE EQUALS | ⋍ | |
| U+22D5 | EQUAL AND PARALLEL TO | | |
| U+29E3 | EQUALS SIGN AND SLANTED PARALLEL | | |
| U+29E4 | EQUALS SIGN AND SLANTED PARALLEL WITH TILDE ABOVE | | |
| U+29E5 | IDENTICAL TO AND SLANTED PARALLEL | | |
| U+2A66 | EQUALS SIGN WITH DOT BELOW | | |
| U+2A67 | IDENTICAL WITH DOT ABOVE | | |
| U+2A6C | SIMILAR MINUS SIMILAR | | |
| U+2A6E | EQUALS WITH ASTERISK | | |
| U+2A6F | ALMOST EQUAL TO WITH CIRCUMFLEX ACCENT | | |
| U+2A70 | APPROXIMATELY EQUAL OR EQUAL TO | | |
| U+2A71 | EQUALS SIGN ABOVE PLUS SIGN | | |
| U+2A72 | PLUS SIGN ABOVE EQUALS SIGN | | |
| U+2A73 | EQUALS SIGN ABOVE TILDE OPERATOR | | |
| U+2A75 | TWO CONSECUTIVE EQUALS SIGNS | | |
| U+2A76 | THREE CONSECUTIVE EQUALS SIGNS | | |
| U+2A77 | EQUALS SIGN WITH TWO DOTS ABOVE AND TWO DOTS BELOW | | |
| U+2A78 | EQUIVALENT WITH FOUR DOTS ABOVE | | |
| U+2AAE | EQUALS SIGN WITH BUMPY ABOVE | | |
| U+FE66 | SMALL EQUALS SIGN | | |
| U+FF1D | FULLWIDTH EQUALS SIGN | | |

The following are inequivalence operators. They might not be chained and they are nonassociative.

| | | | |
|---|---|---|---|
| U+2244 | NOT ASYMPTOTICALLY EQUAL TO | ≄ | NSIMEQ |
| U+2247 | NEITHER APPROXIMATELY NOR ACTUALLY EQUAL TO | ≇ | |
| U+2249 | NOT ALMOST EQUAL TO | ≉ | NAPPROX |
| U+2260 | NOT EQUAL TO | ≠ | =/= NE |
| U+2262 | NOT IDENTICAL TO | ≢ | NEQV |
| U+226D | NOT EQUIVALENT TO | ≭ | |

## C.4.2 Plain Comparison Operators

Every operator listed in this section has lower precedence than any operator listed in Sections C.3.1, C.3.2, and C.3.3.

The following are less-than operators. They may be mixed and chained with each other and with equivalence operators (see Section C.4.1).

| | | | |
|---|---|---|---|
| U+003C | LESS-THAN SIGN | < | < LT |
| U+2264 | LESS-THAN OR EQUAL TO | ≤ | <= LE |
| U+2266 | LESS-THAN OVER EQUAL TO | ≦ | |
| U+2268 | LESS-THAN BUT NOT EQUAL TO | ≨ | |
| U+226A | MUCH LESS-THAN | ≪ | << |
| U+2272 | LESS-THAN OR EQUIVALENT TO | ≲ | |
| U+22D6 | LESS-THAN WITH DOT | ⋖ | DOTLT |
| U+22D8 | VERY MUCH LESS-THAN | ⋘ | <<< |
| U+22DC | EQUAL TO OR LESS-THAN | | |
| U+22E6 | LESS-THAN BUT NOT EQUIVALENT TO | ⋦ | |
| U+29C0 | CIRCLED LESS-THAN | | |
| U+2A79 | LESS-THAN WITH CIRCLE INSIDE | | |
| U+2A7B | LESS-THAN WITH QUESTION MARK ABOVE | | |
| U+2A7D | LESS-THAN OR SLANTED EQUAL TO | | |
| U+2A7F | LESS-THAN OR SLANTED EQUAL TO WITH DOT INSIDE | | |
| U+2A81 | LESS-THAN OR SLANTED EQUAL TO WITH DOT ABOVE | | |
| U+2A83 | LESS-THAN OR SLANTED EQUAL TO WITH DOT ABOVE RIGHT | | |
| U+2A85 | LESS-THAN OR APPROXIMATE | | |
| U+2A87 | LESS-THAN AND SINGLE-LINE NOT EQUAL TO | | |
| U+2A89 | LESS-THAN AND NOT APPROXIMATE | | |
| U+2A8D | LESS-THAN ABOVE SIMILAR OR EQUAL | | |
| U+2A95 | SLANTED EQUAL TO OR LESS-THAN | | |
| U+2A97 | SLANTED EQUAL TO OR LESS-THAN WITH DOT INSIDE | | |
| U+2A99 | DOUBLE-LINE EQUAL TO OR LESS-THAN | | |
| U+2A9B | DOUBLE-LINE SLANTED EQUAL TO OR LESS-THAN | | |
| U+2A9D | SIMILAR OR LESS-THAN | | |
| U+2A9F | SIMILAR ABOVE LESS-THAN ABOVE EQUALS SIGN | | |
| U+2AA1 | DOUBLE NESTED LESS-THAN | | |
| U+2AA3 | DOUBLE NESTED LESS-THAN WITH UNDERBAR | | |
| U+2AA6 | LESS-THAN CLOSED BY CURVE | | |
| U+2AA8 | LESS-THAN CLOSED BY CURVE ABOVE SLANTED EQUAL | | |
| U+2AF7 | TRIPLE NESTED LESS-THAN | | |
| U+2AF9 | DOUBLE-LINE SLANTED LESS-THAN OR EQUAL TO | | |
| U+FE64 | SMALL LESS-THAN SIGN | | |
| U+FF1C | FULLWIDTH LESS-THAN SIGN | | |

The following are greater-than operators. They may be mixed and chained with each other and with equivalence operators (see Section C.4.1).

| | | | |
|---|---|---|---|
| U+003E | GREATER-THAN SIGN | > | > GT |
| U+2265 | GREATER-THAN OR EQUAL TO | ≥ | >= GE |
| U+2267 | GREATER-THAN OVER EQUAL TO | ≧ | |
| U+2269 | GREATER-THAN BUT NOT EQUAL TO | ≩ | |
| U+226B | MUCH GREATER-THAN | ≫ | >> |
| U+2273 | GREATER-THAN OR EQUIVALENT TO | ≳ | |
| U+22D7 | GREATER-THAN WITH DOT | ⋗ | DOTGT |
| U+22D9 | VERY MUCH GREATER-THAN | ⋙ | >>> |

| | | | |
|---|---|---|---|
| U+22DD | EQUAL TO OR GREATER-THAN | | |
| U+22E7 | GREATER-THAN BUT NOT EQUIVALENT TO | ⋧ | |
| U+29C1 | CIRCLED GREATER-THAN | | |
| U+2A7A | GREATER-THAN WITH CIRCLE INSIDE | | |
| U+2A7C | GREATER-THAN WITH QUESTION MARK ABOVE | | |
| U+2A7E | GREATER-THAN OR SLANTED EQUAL TO | | |
| U+2A80 | GREATER-THAN OR SLANTED EQUAL TO WITH DOT INSIDE | | |
| U+2A82 | GREATER-THAN OR SLANTED EQUAL TO WITH DOT ABOVE | | |
| U+2A84 | GREATER-THAN OR SLANTED EQUAL TO WITH DOT ABOVE LEFT | | |
| U+2A86 | GREATER-THAN OR APPROXIMATE | | |
| U+2A88 | GREATER-THAN AND SINGLE-LINE NOT EQUAL TO | | |
| U+2A8A | GREATER-THAN AND NOT APPROXIMATE | | |
| U+2A8E | GREATER-THAN ABOVE SIMILAR OR EQUAL | | |
| U+2A96 | SLANTED EQUAL TO OR GREATER-THAN | | |
| U+2A98 | SLANTED EQUAL TO OR GREATER-THAN WITH DOT INSIDE | | |
| U+2A9A | DOUBLE-LINE EQUAL TO OR GREATER-THAN | | |
| U+2A9C | DOUBLE-LINE SLANTED EQUAL TO OR GREATER-THAN | | |
| U+2A9E | SIMILAR OR GREATER-THAN | | |
| U+2AA0 | SIMILAR ABOVE GREATER-THAN ABOVE EQUALS SIGN | | |
| U+2AA2 | DOUBLE NESTED GREATER-THAN | | |
| U+2AA7 | GREATER-THAN CLOSED BY CURVE | | |
| U+2AA9 | GREATER-THAN CLOSED BY CURVE ABOVE SLANTED EQUAL | | |
| U+2AF8 | TRIPLE NESTED GREATER-THAN | | |
| U+2AFA | DOUBLE-LINE SLANTED GREATER-THAN OR EQUAL TO | | |
| U+FE65 | SMALL GREATER-THAN SIGN | | |
| U+FF1E | FULLWIDTH GREATER-THAN SIGN | | |

The following are miscellaneous plain comparison operators. They might not be mixed or chained and they are nonassociative.

| | | | |
|---|---|---|---|
| U+226E | NOT LESS-THAN | ≮ | NLT |
| U+226F | NOT GREATER-THAN | ≯ | NGT |
| U+2270 | NEITHER LESS-THAN NOR EQUAL TO | ≰ | NLE |
| U+2271 | NEITHER GREATER-THAN NOR EQUAL TO | ≱ | NGE |
| U+2274 | NEITHER LESS-THAN NOR EQUIVALENT TO | ≴ | |
| U+2275 | NEITHER GREATER-THAN NOR EQUIVALENT TO | ≵ | |
| U+2276 | LESS-THAN OR GREATER-THAN | ≶ | |
| U+2277 | GREATER-THAN OR LESS-THAN | ≷ | |
| U+2278 | NEITHER LESS-THAN NOR GREATER-THAN | | |
| U+2279 | NEITHER GREATER-THAN NOR LESS-THAN | | |
| U+22DA | LESS-THAN EQUAL TO OR GREATER-THAN | ⋚ | |
| U+22DB | GREATER-THAN EQUAL TO OR LESS-THAN | ⋛ | |
| U+2A8B | LESS-THAN ABOVE DOUBLE-LINE EQUAL ABOVE GREATER-THAN | | |
| U+2A8C | GREATER-THAN ABOVE DOUBLE-LINE EQUAL ABOVE LESS-THAN | | |
| U+2A8F | LESS-THAN ABOVE SIMILAR ABOVE GREATER-THAN | | |
| U+2A90 | GREATER-THAN ABOVE SIMILAR ABOVE LESS-THAN | | |
| U+2A91 | LESS-THAN ABOVE GREATER-THAN ABOVE DOUBLE-LINE EQUAL | | |
| U+2A92 | GREATER-THAN ABOVE LESS-THAN ABOVE DOUBLE-LINE EQUAL | | |
| U+2A93 | LESS-THAN ABOVE SLANTED EQUAL ABOVE GREATER-THAN ABOVE SLANTED EQUAL | | |
| U+2A94 | GREATER-THAN ABOVE SLANTED EQUAL ABOVE LESS-THAN ABOVE SLANTED EQUAL | | |
| U+2AA4 | GREATER-THAN OVERLAPPING LESS-THAN | | |
| U+2AA5 | GREATER-THAN BESIDE LESS-THAN | | |

The following are not really comparison operators, but it is convenient to list them here because they also have lower precedence than any operator listed in Sections C.3.1, C.3.2, and C.3.3. They are nonassociative.

| | | | |
|---|---|---|---|
| U+0023 | NUMBER SIGN | # | # |
| U+003A | COLON | : | : |

## C.4.3 Set Comparison Operators

Every operator listed in this section has lower precedence than any operator listed in Section C.3.4.

The following are subset comparison operators. They may be mixed and chained with each other and with equivalence operators (see Section C.4.1).

| | | | |
|---|---|---|---|
| U+2282 | SUBSET OF | ⊂ | SUBSET |
| U+2286 | SUBSET OF OR EQUAL TO | ⊆ | SUBSETEQ |
| U+228A | SUBSET OF WITH NOT EQUAL TO | ⊊ | SUBSETNEQ |
| U+22D0 | DOUBLE SUBSET | ⋐ | SUBSUB |
| U+27C3 | OPEN SUBSET | | |
| U+2ABD | SUBSET WITH DOT | | |
| U+2ABF | SUBSET WITH PLUS SIGN BELOW | | |
| U+2AC1 | SUBSET WITH MULTIPLICATION SIGN BELOW | | |
| U+2AC3 | SUBSET OF OR EQUAL TO WITH DOT ABOVE | | |
| U+2AC5 | SUBSET OF ABOVE EQUALS SIGN | | |
| U+2AC7 | SUBSET OF ABOVE TILDE OPERATOR | | |
| U+2AC9 | SUBSET OF ABOVE ALMOST EQUAL TO | | |
| U+2ACB | SUBSET OF ABOVE NOT EQUAL TO | | |
| U+2ACF | CLOSED SUBSET | | |
| U+2AD1 | CLOSED SUBSET OR EQUAL TO | | |
| U+2AD5 | SUBSET ABOVE SUBSET | | |

The following are superset comparison operators. They may be mixed and chained with each other and with equivalence operators (see Section C.4.1).

| | | | |
|---|---|---|---|
| U+2283 | SUPERSET OF | ⊃ | SUPSET |
| U+2287 | SUPERSET OF OR EQUAL TO | ⊇ | SUPSETEQ |
| U+228B | SUPERSET OF WITH NOT EQUAL TO | ⊋ | SUPSETNEQ |
| U+22D1 | DOUBLE SUPERSET | ⋑ | SUPSUP |
| U+27C4 | OPEN SUPERSET | | |
| U+2ABE | SUPERSET WITH DOT | | |
| U+2AC0 | SUPERSET WITH PLUS SIGN BELOW | | |
| U+2AC2 | SUPERSET WITH MULTIPLICATION SIGN BELOW | | |
| U+2AC4 | SUPERSET OF OR EQUAL TO WITH DOT ABOVE | | |
| U+2AC6 | SUPERSET OF ABOVE EQUALS SIGN | | |
| U+2AC8 | SUPERSET OF ABOVE TILDE OPERATOR | | |
| U+2ACA | SUPERSET OF ABOVE ALMOST EQUAL TO | | |
| U+2ACC | SUPERSET OF ABOVE NOT EQUAL TO | | |
| U+2AD0 | CLOSED SUPERSET | | |
| U+2AD2 | CLOSED SUPERSET OR EQUAL TO | | |
| U+2AD6 | SUPERSET ABOVE SUPERSET | | |

The following are miscellaneous set comparison operators. They might not be mixed or chained and they are nonassociative.

| | | | |
|---|---|---|---|
| U+2284 | NOT A SUBSET OF | ⊄ | NSUBSET |

| | | | |
|---|---|---|---|
| U+2285 | NOT A SUPERSET OF | ⊅ | NSUPSET |
| U+2288 | NEITHER A SUBSET OF NOR EQUAL TO | ⊈ | NSUBSETEQ |
| U+2289 | NEITHER A SUPERSET OF NOR EQUAL TO | ⊉ | NSUPSETEQ |
| U+2AD3 | SUBSET ABOVE SUPERSET | | |
| U+2AD4 | SUPERSET ABOVE SUBSET | | |
| U+2AD7 | SUPERSET BESIDE SUBSET | | |
| U+2AD8 | SUPERSET BESIDE AND JOINED BY DASH WITH SUBSET | | |

### C.4.4 Square Comparison Operators

Every operator listed in this section has lower precedence than any operator listed in Section C.3.5.

The following are square “image of” comparison operators. They may be mixed and chained with each other and with equivalence operators (see Section C.4.1).

| | | | |
|---|---|---|---|
| U+228F | SQUARE IMAGE OF | ⊏ | SQSUBSET |
| U+2291 | SQUARE IMAGE OF OR EQUAL TO | ⊑ | SQSUBSETEQ |
| U+22E4 | SQUARE IMAGE OF OR NOT EQUAL TO | | |

The following are square “original of” comparison operators. They may be mixed and chained with each other and with equivalence operators (see Section C.4.1).

| | | | |
|---|---|---|---|
| U+2290 | SQUARE ORIGINAL OF | ⊐ | SQSUPSET |
| U+2292 | SQUARE ORIGINAL OF OR EQUAL TO | ⊒ | SQSUPSETEQ |
| U+22E5 | SQUARE ORIGINAL OF OR NOT EQUAL TO | | |

The following are miscellaneous square comparison operators. They might not be mixed or chained and they are nonassociative.

| | | |
|---|---|---|
| U+22E2 | NOT SQUARE IMAGE OF OR EQUAL TO | ⋢ |
| U+22E3 | NOT SQUARE ORIGINAL OF OR EQUAL TO | ⋣ |

### C.4.5 Curly Comparison Operators

Every operator listed in this section has lower precedence than any operator listed in Section C.3.6.

The following are curly “precedes” comparison operators. They may be mixed and chained with each other and with equivalence operators (see Section C.4.1).

| | | | |
|---|---|---|---|
| U+227A | PRECEDES | ≺ | PREC |
| U+227C | PRECEDES OR EQUAL TO | ≼ | PRECEQ |
| U+227E | PRECEDES OR EQUIVALENT TO | ≾ | PRECSIM |
| U+22B0 | PRECEDES UNDER RELATION | | |
| U+22DE | EQUAL TO OR PRECEDES | ⋞ | EQPREC |
| U+22E8 | PRECEDES BUT NOT EQUIVALENT TO | ⋨ | PRECNSIM |
| U+2AAF | PRECEDES ABOVE SINGLE-LINE EQUALS SIGN | | |
| U+2AB1 | PRECEDES ABOVE SINGLE-LINE NOT EQUAL TO | | |
| U+2AB3 | PRECEDES ABOVE EQUALS SIGN | | |
| U+2AB5 | PRECEDES ABOVE NOT EQUAL TO | | |
| U+2AB7 | PRECEDES ABOVE ALMOST EQUAL TO | | |
| U+2AB9 | PRECEDES ABOVE NOT ALMOST EQUAL TO | | |
| U+2ABB | DOUBLE PRECEDES | | |

The following are curly “succeeds” comparison operators. They may be mixed and chained with each other and with equivalence operators (see Section C.4.1).

| | | | |
|---|---|---|---|
| U+227B | SUCCEEDS | ≻ | SUCC |
| U+227D | SUCCEEDS OR EQUAL TO | ≽ | SUCCEQ |
| U+227F | SUCCEEDS OR EQUIVALENT TO | ≿ | SUCCSIM |
| U+22B1 | SUCCEEDS UNDER RELATION | | |
| U+22DF | EQUAL TO OR SUCCEEDS | ⋟ | EQSUCC |
| U+22E9 | SUCCEEDS BUT NOT EQUIVALENT TO | ⋩ | SUCCNSIM |
| U+2AB0 | SUCCEEDS ABOVE SINGLE-LINE EQUALS SIGN | | |
| U+2AB2 | SUCCEEDS ABOVE SINGLE-LINE NOT EQUAL TO | | |
| U+2AB4 | SUCCEEDS ABOVE EQUALS SIGN | | |
| U+2AB6 | SUCCEEDS ABOVE NOT EQUAL TO | | |
| U+2AB8 | SUCCEEDS ABOVE ALMOST EQUAL TO | | |
| U+2ABA | SUCCEEDS ABOVE NOT ALMOST EQUAL TO | | |
| U+2ABC | DOUBLE SUCCEEDS | | |

The following are miscellaneous curly comparison operators. They might not be mixed or chained and they are nonassociative.

| | | | |
|---|---|---|---|
| U+2280 | DOES NOT PRECEDE | ⊀ | NPREC |
| U+2281 | DOES NOT SUCCEED | ⊁ | NSUCC |
| U+22E0 | DOES NOT PRECEDE OR EQUAL | ⋠ | |
| U+22E1 | DOES NOT SUCCEED OR EQUAL | ⋡ | |

## C.4.6 Triangular Comparison Operators

The following are triangular “subgroup” comparison operators. They may be mixed and chained with each other and with equivalence operators (see Section C.4.1).

| | | |
|---|---|---|
| U+22B2 | NORMAL SUBGROUP OF | ⊲ |
| U+22B4 | NORMAL SUBGROUP OF OR EQUAL TO | ⊴ |

The following are triangular “contains as subgroup” comparison operators. They may be mixed and chained with each other and with equivalence operators (see Section C.4.1).

| | | |
|---|---|---|
| U+22B3 | CONTAINS AS NORMAL SUBGROUP | ⊳ |
| U+22B5 | CONTAINS AS NORMAL SUBGROUP OR EQUAL TO | ⊵ |

The following are miscellaneous triangular comparison operators. They might not be mixed or chained and they are nonassociative.

| | | |
|---|---|---|
| U+22EA | NOT NORMAL SUBGROUP OF | ⋪ |
| U+22EB | DOES NOT CONTAIN AS NORMAL SUBGROUP | ⋫ |
| U+22EC | NOT NORMAL SUBGROUP OF OR EQUAL TO | ⋬ |
| U+22ED | DOES NOT CONTAIN AS NORMAL SUBGROUP OR EQUAL | ⋭ |

## C.4.7 Chickenfoot Comparison Operators

The following are chickenfoot “smaller than” comparison operators. They may be mixed and chained with each other and with equivalence operators (see Section C.4.1).

| | | | |
|---|---|---|---|
| U+2AAA | SMALLER THAN | ⪪ | SMALLER |
| U+2AAC | SMALLER THAN OR EQUAL TO | ⪬ | SMALLEREQ |

The following are chickenfoot “larger than” comparison operators. They may be mixed and chained with each other and with equivalence operators (see Section C.4.1).

| | | | |
|---|---|---|---|
| U+2AAB | LARGER THAN | ⪫ | LARGER |
| U+2AAD | LARGER THAN OR EQUAL TO | ⪭ | LARGEREQ |

### C.4.8 Miscellaneous Relational Operators

The following operators are considered to be relational operators, having higher precedence than certain boolean operators, as described in a later section. They are nonassociative.

| | | | |
|---|---|---|---|
| U+2208 | ELEMENT OF | ∈ | IN |
| U+2209 | NOT AN ELEMENT OF | ∉ | NOTIN |
| U+220A | SMALL ELEMENT OF | ∊ | |
| U+220B | CONTAINS AS MEMBER | ∋ | CONTAINS |
| U+220C | DOES NOT CONTAIN AS MEMBER | ∌ | |
| U+220D | SMALL CONTAINS AS MEMBER | ∍ | |
| U+22F2 | ELEMENT OF WITH LONG HORIZONTAL STROKE | | |
| U+22F3 | ELEMENT OF WITH VERTICAL BAR AT END OF HORIZONTAL STROKE | | |
| U+22F4 | SMALL ELEMENT OF WITH VERTICAL BAR AT END OF HORIZONTAL STROKE | | |
| U+22F5 | ELEMENT OF WITH DOT ABOVE | ⋵ | |
| U+22F6 | ELEMENT OF WITH OVERBAR | ⋶ | |
| U+22F7 | SMALL ELEMENT OF WITH OVERBAR | ⋷ | |
| U+22F8 | ELEMENT OF WITH UNDERBAR | ⋸ | |
| U+22F9 | ELEMENT OF WITH TWO HORIZONTAL STROKES | | |
| U+22FA | CONTAINS WITH LONG HORIZONTAL STROKE | | |
| U+22FB | CONTAINS WITH VERTICAL BAR AT END OF HORIZONTAL STROKE | | |
| U+22FC | SMALL CONTAINS WITH VERTICAL BAR AT END OF HORIZONTAL STROKE | | |
| U+22FD | CONTAINS WITH OVERBAR | ⋽ | |
| U+22FE | SMALL CONTAINS WITH OVERBAR | ⋾ | |
| U+22FF | Z NOTATION BAG MEMBERSHIP | | |

## C.5 Boolean Operators

Every operator listed in this section has lower precedence than any operator listed in Section C.4.

The following are the boolean conjunction operators. They are left associative.

| | | | |
|---|---|---|---|
| U+2227 | LOGICAL AND | ∧ | AND |
| U+27D1 | AND WITH DOT | | |
| U+2A51 | LOGICAL AND WITH DOT ABOVE | ⩑ | |
| U+2A53 | DOUBLE LOGICAL AND | | |
| U+2A55 | TWO INTERSECTING LOGICAL AND | ⩕ | |
| U+2A5A | LOGICAL AND WITH MIDDLE STEM | | |
| U+2A5C | LOGICAL AND WITH HORIZONTAL DASH | | |
| U+2A5E | LOGICAL AND WITH DOUBLE OVERBAR | | |
| U+2A60 | LOGICAL AND WITH DOUBLE UNDERBAR | | |

The following are the boolean disjunction operators. They are left associative.

| | | | |
|---|---|---|---|
| U+2228 | LOGICAL OR | ∨ | OR |
| U+2A52 | LOGICAL OR WITH DOT ABOVE | ⩒ | |

|  |  |  |  |
|---|---|---|---|
| U+2A54 | DOUBLE LOGICAL OR |  |  |
| U+2A56 | TWO INTERSECTING LOGICAL OR | ⩖ |  |
| U+2A5B | LOGICAL OR WITH MIDDLE STEM |  |  |
| U+2A5D | LOGICAL OR WITH HORIZONTAL DASH |  |  |
| U+2A62 | LOGICAL OR WITH DOUBLE OVERBAR |  |  |
| U+2A63 | LOGICAL OR WITH DOUBLE UNDERBAR |  |  |

They each have lower precedence than any of the boolean conjunction operators.

The following are miscellaneous boolean operators that are nonassociative.

|  |  |  |  |
|---|---|---|---|
| U+2192 | RIGHTWARDS ARROW | → | -> IMPLIES |
| U+2194 | LEFT RIGHT ARROW | ↔ | <-> IFF |
| U+22BC | NAND | ⊼ |  |
| U+22BD | NOR | ⊽ |  |

The following is a miscellaneous boolean operator that is left associative.

|  |  |  |  |
|---|---|---|---|
| U+22BB | XOR | ⊻ |  |

## C.6 Other Operators

All the operators listed in this section are nonassociative.

As specified in Section 16.2, the following operator has higher precedence than any other operator.

|  |  |  |  |
|---|---|---|---|
| U+005E | CIRCUMFLEX ACCENT | ˆ | ^ |

Each of the following operators has no defined precedence relationships to any of the other operators listed in this appendix.

|  |  |  |  |
|---|---|---|---|
| U+0021 | EXCLAMATION MARK | ! | ! |
| U+0024 | DOLLAR SIGN | $ | $ |
| U+0025 | PERCENT SIGN | % | % |
| U+003F | QUESTION MARK | ? | ? |
| U+0040 | COMMERCIAL AT | @ | @ |
| U+007E | TILDE | ˜ | ~ |
| U+00A1 | INVERTED EXCLAMATION MARK | ¡ |  |
| U+00A2 | CENT SIGN |  | CENTS |
| U+00A3 | POUND SIGN |  |  |
| U+00A4 | CURRENCY SIGN |  |  |
| U+00A5 | YEN SIGN |  |  |
| U+00A6 | BROKEN BAR |  |  |
| U+00AC | NOT SIGN | ¬ | NOT |
| U+00B0 | DEGREE SIGN | ° | DEGREES |
| U+00BF | INVERTED QUESTION MARK | ¿ |  |
| U+203C | DOUBLE EXCLAMATION MARK | ‼ | !! |
| U+2190 | LEFTWARDS ARROW | ← | <- |
| U+2191 | UPWARDS ARROW | ↑ | UPARROW |
| U+2193 | DOWNWARDS ARROW | ↓ | DOWNARROW |
| U+2195 | UP DOWN ARROW | ↕ | UPDOWNARROW |

| | | | |
|---|---|---|---|
| U+2196 | NORTH WEST ARROW | ↖ | NWARROW |
| U+2197 | NORTH EAST ARROW | ↗ | NEARROW |
| U+2198 | SOUTH EAST ARROW | ↘ | SEARROW |
| U+2199 | SOUTH WEST ARROW | ↙ | SWARROW |
| U+219A | LEFTWARDS ARROW WITH STROKE | ↚ | <-/- |
| U+219B | RIGHTWARDS ARROW WITH STROKE | ↛ | -/-> |
| U+219C | LEFTWARDS WAVE ARROW | | |
| U+219D | RIGHTWARDS WAVE ARROW | ↝ | LEADSTO |
| U+219E | LEFTWARDS TWO HEADED ARROW | | |
| U+219F | UPWARDS TWO HEADED ARROW | | |
| U+21A0 | RIGHTWARDS TWO HEADED ARROW | | |
| U+21A1 | DOWNWARDS TWO HEADED ARROW | | |
| U+21A2 | LEFTWARDS ARROW WITH TAIL | | |
| U+21A3 | RIGHTWARDS ARROW WITH TAIL | | |
| U+21A4 | LEFTWARDS ARROW FROM BAR | | |
| U+21A5 | UPWARDS ARROW FROM BAR | | |
| U+21A6 | RIGHTWARDS ARROW FROM BAR | ↦ | MAPSTO \|-> |
| U+21A7 | DOWNWARDS ARROW FROM BAR | | |
| U+21A8 | UP DOWN ARROW WITH BASE | | |
| U+21A9 | LEFTWARDS ARROW WITH HOOK | | |
| U+21AA | RIGHTWARDS ARROW WITH HOOK | | |
| U+21AB | LEFTWARDS ARROW WITH LOOP | | |
| U+21AC | RIGHTWARDS ARROW WITH LOOP | | |
| U+21AD | LEFT RIGHT WAVE ARROW | | |
| U+21AE | LEFT RIGHT ARROW WITH STROKE | | |
| U+21AF | DOWNWARDS ZIGZAG ARROW | | |
| U+21B0 | UPWARDS ARROW WITH TIP LEFTWARDS | | |
| U+21B1 | UPWARDS ARROW WITH TIP RIGHTWARDS | | |
| U+21B2 | DOWNWARDS ARROW WITH TIP LEFTWARDS | | |
| U+21B3 | DOWNWARDS ARROW WITH TIP RIGHTWARDS | | |
| U+21B4 | RIGHTWARDS ARROW WITH CORNER DOWNWARDS | | |
| U+21B5 | DOWNWARDS ARROW WITH CORNER LEFTWARDS | | |
| U+21B6 | ANTICLOCKWISE TOP SEMICIRCLE ARROW | | |
| U+21B7 | CLOCKWISE TOP SEMICIRCLE ARROW | | |
| U+21B8 | NORTH WEST ARROW TO LONG BAR | | |
| U+21B9 | LEFTWARDS ARROW TO BAR OVER RIGHTWARDS ARROW TO BAR | | |
| U+21BA | ANTICLOCKWISE OPEN CIRCLE ARROW | | |
| U+21BB | CLOCKWISE OPEN CIRCLE ARROW | | |
| U+21BC | LEFTWARDS HARPOON WITH BARB UPWARDS | ↼ | LEFTHARPOONUP |
| U+21BD | LEFTWARDS HARPOON WITH BARB DOWNWARDS | ↽ | LEFTHARPOONDOWN |
| U+21BE | UPWARDS HARPOON WITH BARB RIGHTWARDS | ↾ | UPHARPOONRIGHT |
| U+21BF | UPWARDS HARPOON WITH BARB LEFTWARDS | ↿ | UPHARPOONLEFT |
| U+21C0 | RIGHTWARDS HARPOON WITH BARB UPWARDS | ⇀ | RIGHTHARPOONUP |
| U+21C1 | RIGHTWARDS HARPOON WITH BARB DOWNWARDS | ⇁ | RIGHTHARPOONDOWN |
| U+21C2 | DOWNWARDS HARPOON WITH BARB RIGHTWARDS | ⇂ | DOWNHARPOONRIGHT |
| U+21C3 | DOWNWARDS HARPOON WITH BARB LEFTWARDS | ⇃ | DOWNHARPOONLEFT |
| U+21C4 | RIGHTWARDS ARROW OVER LEFTWARDS ARROW | ⇄ | RIGHTLEFTARROWS |
| U+21C5 | UPWARDS ARROW LEFTWARDS OF DOWNWARDS ARROW | | |
| U+21C6 | LEFTWARDS ARROW OVER RIGHTWARDS ARROW | ⇆ | LEFTRIGHTARROWS |
| U+21C7 | LEFTWARDS PAIRED ARROWS | ⇇ | LEFTLEFTARROWS |
| U+21C8 | UPWARDS PAIRED ARROWS | ⇈ | UPUPARROWS |
| U+21C9 | RIGHTWARDS PAIRED ARROWS | ⇉ | RIGHTRIGHTARROWS |

| | | | |
|---|---|---|---|
| U+21CA | DOWNWARDS PAIRED ARROWS | ⇊ | DOWNDOWNARROWS |
| U+21CB | LEFTWARDS HARPOON OVER RIGHTWARDS HARPOON | | |
| U+21CC | RIGHTWARDS HARPOON OVER LEFTWARDS HARPOON | ⇌ | RIGHTLEFTHARPOONS |
| U+21CD | LEFTWARDS DOUBLE ARROW WITH STROKE | ⇍ | |
| U+21CE | LEFT RIGHT DOUBLE ARROW WITH STROKE | ⇎ | |
| U+21CF | RIGHTWARDS DOUBLE ARROW WITH STROKE | ⇏ | |
| U+21D0 | LEFTWARDS DOUBLE ARROW | ⇐ | |
| U+21D1 | UPWARDS DOUBLE ARROW | ⇑ | |
| U+21D2 | RIGHTWARDS DOUBLE ARROW | ⇒ | => |
| U+21D3 | DOWNWARDS DOUBLE ARROW | ⇓ | |
| U+21D4 | LEFT RIGHT DOUBLE ARROW | ⇔ | <=> |
| U+21D5 | UP DOWN DOUBLE ARROW | ⇕ | |
| U+21D6 | NORTH WEST DOUBLE ARROW | | |
| U+21D7 | NORTH EAST DOUBLE ARROW | | |
| U+21D8 | SOUTH EAST DOUBLE ARROW | | |
| U+21D9 | SOUTH WEST DOUBLE ARROW | | |
| U+21DA | LEFTWARDS TRIPLE ARROW | ⇚ | |
| U+21DB | RIGHTWARDS TRIPLE ARROW | ⇛ | |
| U+21DC | LEFTWARDS SQUIGGLE ARROW | | |
| U+21DD | RIGHTWARDS SQUIGGLE ARROW | ⇝ | |
| U+21DE | UPWARDS ARROW WITH DOUBLE STROKE | | |
| U+21DF | DOWNWARDS ARROW WITH DOUBLE STROKE | | |
| U+21E0 | LEFTWARDS DASHED ARROW | ⇠ | |
| U+21E1 | UPWARDS DASHED ARROW | | |
| U+21E2 | RIGHTWARDS DASHED ARROW | ⇢ | |
| U+21E3 | DOWNWARDS DASHED ARROW | | |
| U+21E4 | LEFTWARDS ARROW TO BAR | | |
| U+21E5 | RIGHTWARDS ARROW TO BAR | | |
| U+21E6 | LEFTWARDS WHITE ARROW | | |
| U+21E7 | UPWARDS WHITE ARROW | | |
| U+21E8 | RIGHTWARDS WHITE ARROW | | |
| U+21E9 | DOWNWARDS WHITE ARROW | | |
| U+21EA | UPWARDS WHITE ARROW FROM BAR | | |
| U+21EB | UPWARDS WHITE ARROW ON PEDESTAL | | |
| U+21EC | UPWARDS WHITE ARROW ON PEDESTAL WITH HORIZONTAL BAR | | |
| U+21ED | UPWARDS WHITE ARROW ON PEDESTAL WITH VERTICAL BAR | | |
| U+21EE | UPWARDS WHITE DOUBLE ARROW | | |
| U+21EF | UPWARDS WHITE DOUBLE ARROW ON PEDESTAL | | |
| U+21F0 | RIGHTWARDS WHITE ARROW FROM WALL | | |
| U+21F1 | NORTH WEST ARROW TO CORNER | | |
| U+21F2 | SOUTH EAST ARROW TO CORNER | | |
| U+21F3 | UP DOWN WHITE ARROW | | |
| U+21F4 | RIGHT ARROW WITH SMALL CIRCLE | | |
| U+21F5 | DOWNWARDS ARROW LEFTWARDS OF UPWARDS ARROW | | |
| U+21F6 | THREE RIGHTWARDS ARROWS | | |
| U+21F7 | LEFTWARDS ARROW WITH VERTICAL STROKE | | |
| U+21F8 | RIGHTWARDS ARROW WITH VERTICAL STROKE | | |
| U+21F9 | LEFT RIGHT ARROW WITH VERTICAL STROKE | | |
| U+21FA | LEFTWARDS ARROW WITH DOUBLE VERTICAL STROKE | | |
| U+21FB | RIGHTWARDS ARROW WITH DOUBLE VERTICAL STROKE | | |
| U+21FC | LEFT RIGHT ARROW WITH DOUBLE VERTICAL STROKE | | |
| U+21FD | LEFTWARDS OPEN-HEADED ARROW | | |

|  |  |  |  |
|---|---|---|---|
| U+21FE | RIGHTWARDS OPEN-HEADED ARROW |  |  |
| U+21FF | LEFT RIGHT OPEN-HEADED ARROW |  |  |
| U+2201 | COMPLEMENT | ∁ |  |
| U+2202 | PARTIAL DIFFERENTIAL | ∂ | DEL |
| U+2204 | THERE DOES NOT EXIST | ∄ |  |
| U+2206 | INCREMENT | ∆ |  |
| U+220F | N-ARY PRODUCT | ∏ | PROD |
| U+2210 | N-ARY COPRODUCT | ∐ | COPROD |
| U+2211 | N-ARY SUMMATION | ∑ | SUM |
| U+2218 | RING OPERATOR | ∘ | CIRC RING COMPOSE |
| U+2219 | BULLET OPERATOR | ∙ |  |
| U+221A | SQUARE ROOT | √ | SQRT |
| U+221B | CUBE ROOT |  | CBRT |
| U+221C | FOURTH ROOT |  | FOURTHROOT |
| U+221D | PROPORTIONAL TO | ∝ | PROPTO |
| U+2223 | DIVIDES | ∣ | DIVIDES |
| U+2224 | DOES NOT DIVIDE | ∤ |  |
| U+2225 | PARALLEL TO | ∥ | PARALLEL |
| U+2226 | NOT PARALLEL TO | ∦ | NPARALLEL |
| U+222B | INTEGRAL | ∫ |  |
| U+222C | DOUBLE INTEGRAL |  |  |
| U+222D | TRIPLE INTEGRAL |  |  |
| U+222E | CONTOUR INTEGRAL | ∮ |  |
| U+222F | SURFACE INTEGRAL |  |  |
| U+2230 | VOLUME INTEGRAL |  |  |
| U+2231 | CLOCKWISE INTEGRAL |  |  |
| U+2232 | CLOCKWISE CONTOUR INTEGRAL |  |  |
| U+2233 | ANTICLOCKWISE CONTOUR INTEGRAL |  |  |
| U+2234 | THEREFORE | ∴ |  |
| U+2235 | BECAUSE | ∵ |  |
| U+2236 | RATIO |  |  |
| U+2237 | PROPORTION |  |  |
| U+2239 | EXCESS |  |  |
| U+223A | GEOMETRIC PROPORTION |  |  |
| U+223B | HOMOTHETIC |  |  |
| U+223C | TILDE OPERATOR | ∼ |  |
| U+223D | REVERSED TILDE | ∽ |  |
| U+223E | INVERTED LAZY S |  |  |
| U+223F | SINE WAVE |  |  |
| U+2240 | WREATH PRODUCT | ≀ | WREATH |
| U+2241 | NOT TILDE | ≁ |  |
| U+224B | TRIPLE TILDE |  |  |
| U+224F | DIFFERENCE BETWEEN | ≏ | BUMPEQ |
| U+2250 | APPROACHES THE LIMIT | ≐ | DOTEQ |
| U+2258 | CORRESPONDS TO |  |  |
| U+2259 | ESTIMATES |  |  |
| U+225A | EQUIANGULAR TO |  |  |
| U+225E | MEASURED BY |  |  |
| U+226C | BETWEEN | ≬ |  |
| U+228C | MULTISET |  |  |
| U+229A | CIRCLED RING OPERATOR | ⊚ | CIRCLEDRING |
| U+229D | CIRCLED DASH | ⊝ |  |

| | | | |
|---|---|---|---|
| U+22A2 | RIGHT TACK | ⊢ | VDASH TURNSTILE |
| U+22A3 | LEFT TACK | ⊣ | DASHV |
| U+22A6 | ASSERTION | ⊦ | |
| U+22A7 | MODELS | ⊧ | |
| U+22A8 | TRUE | ⊨ | |
| U+22A9 | FORCES | ⊩ | |
| U+22AA | TRIPLE VERTICAL BAR RIGHT TURNSTILE | ⊪ | |
| U+22AB | DOUBLE VERTICAL BAR DOUBLE RIGHT TURNSTILE | | |
| U+22AC | DOES NOT PROVE | ⊬ | |
| U+22AD | NOT TRUE | | |
| U+22AE | DOES NOT FORCE | ⊮ | |
| U+22AF | NEGATED DOUBLE VERTICAL BAR DOUBLE RIGHT TURNSTILE | ⊯ | |
| U+22B6 | ORIGINAL OF | | |
| U+22B7 | IMAGE OF | | |
| U+22B8 | MULTIMAP | ⊸ | |
| U+22B9 | HERMITIAN CONJUGATE MATRIX | | |
| U+22BA | INTERCALATE | ⊺ | |
| U+22BE | RIGHT ANGLE WITH ARC | | |
| U+22BF | RIGHT TRIANGLE | | |
| U+22C0 | N-ARY LOGICAL AND | ⋀ | BIGAND ALL |
| U+22C1 | N-ARY LOGICAL OR | ⋁ | BIGOR ANY |
| U+22C2 | N-ARY INTERSECTION | ⋂ | BIGCAP BIGINTERSECT |
| U+22C3 | N-ARY UNION | ⋃ | BIGCUP BIGUNION |
| U+22C4 | DIAMOND OPERATOR | ⋄ | DIAMOND |
| U+22C6 | STAR OPERATOR | ⋆ | STAR |
| U+22C7 | DIVISION TIMES | ⋇ | |
| U+22C8 | BOWTIE | ⋈ | |
| U+22C9 | LEFT NORMAL FACTOR SEMIDIRECT PRODUCT | ⋉ | |
| U+22CA | RIGHT NORMAL FACTOR SEMIDIRECT PRODUCT | ⋊ | |
| U+22CB | LEFT SEMIDIRECT PRODUCT | ⋋ | |
| U+22CC | RIGHT SEMIDIRECT PRODUCT | ⋌ | |
| U+22D4 | PITCHFORK | ⋔ | |
| U+22EE | VERTICAL ELLIPSIS | | |
| U+22EF | MIDLINE HORIZONTAL ELLIPSIS | | |
| U+22F0 | UP RIGHT DIAGONAL ELLIPSIS | | |
| U+22F1 | DOWN RIGHT DIAGONAL ELLIPSIS | | |
| U+27C0 | THREE DIMENSIONAL ANGLE | | |
| U+27C1 | WHITE TRIANGLE CONTAINING SMALL WHITE TRIANGLE | | |
| U+27C2 | PERPENDICULAR | | PERP |
| U+27D0 | WHITE DIAMOND WITH CENTRED DOT | | |
| U+27D2 | ELEMENT OF OPENING UPWARDS | | |
| U+27D3 | LOWER RIGHT CORNER WITH DOT | | |
| U+27D4 | UPPER LEFT CORNER WITH DOT | | |
| U+27D5 | LEFT OUTER JOIN | | |
| U+27D6 | RIGHT OUTER JOIN | | |
| U+27D7 | FULL OUTER JOIN | | |
| U+27D8 | LARGE UP TACK | | |
| U+27D9 | LARGE DOWN TACK | | |
| U+27DA | LEFT AND RIGHT DOUBLE TURNSTILE | | |
| U+27DB | LEFT AND RIGHT TACK | | |
| U+27DC | LEFT MULTIMAP | | |
| U+27DD | LONG RIGHT TACK | | |

|  |  |
|---|---|
| U+27DE | LONG LEFT TACK |
| U+27DF | UP TACK WITH CIRCLE ABOVE |
| U+27E0 | LOZENGE DIVIDED BY HORIZONTAL RULE |
| U+27E1 | WHITE CONCAVE-SIDED DIAMOND |
| U+27E2 | WHITE CONCAVE-SIDED DIAMOND WITH LEFTWARDS TICK |
| U+27E3 | WHITE CONCAVE-SIDED DIAMOND WITH RIGHTWARDS TICK |
| U+27E4 | WHITE SQUARE WITH LEFTWARDS TICK |
| U+27E5 | WHITE SQUARE WITH RIGHTWARDS TICK |
| U+27F0 | UPWARDS QUADRUPLE ARROW |
| U+27F1 | DOWNWARDS QUADRUPLE ARROW |
| U+27F2 | ANTICLOCKWISE GAPPED CIRCLE ARROW |
| U+27F3 | CLOCKWISE GAPPED CIRCLE ARROW |
| U+27F4 | RIGHT ARROW WITH CIRCLED PLUS |
| U+27F5 | LONG LEFTWARDS ARROW |
| U+27F6 | LONG RIGHTWARDS ARROW |
| U+27F7 | LONG LEFT RIGHT ARROW |
| U+27F8 | LONG LEFTWARDS DOUBLE ARROW |
| U+27F9 | LONG RIGHTWARDS DOUBLE ARROW |
| U+27FA | LONG LEFT RIGHT DOUBLE ARROW |
| U+27FB | LONG LEFTWARDS ARROW FROM BAR |
| U+27FC | LONG RIGHTWARDS ARROW FROM BAR |
| U+27FD | LONG LEFTWARDS DOUBLE ARROW FROM BAR |
| U+27FE | LONG RIGHTWARDS DOUBLE ARROW FROM BAR |
| U+27FF | LONG RIGHTWARDS SQUIGGLE ARROW |
| U+2900 | RIGHTWARDS TWO-HEADED ARROW WITH VERTICAL STROKE |
| U+2901 | RIGHTWARDS TWO-HEADED ARROW WITH DOUBLE VERTICAL STROKE |
| U+2902 | LEFTWARDS DOUBLE ARROW WITH VERTICAL STROKE |
| U+2903 | RIGHTWARDS DOUBLE ARROW WITH VERTICAL STROKE |
| U+2904 | LEFT RIGHT DOUBLE ARROW WITH VERTICAL STROKE |
| U+2905 | RIGHTWARDS TWO-HEADED ARROW FROM BAR |
| U+2906 | LEFTWARDS DOUBLE ARROW FROM BAR |
| U+2907 | RIGHTWARDS DOUBLE ARROW FROM BAR |
| U+2908 | DOWNWARDS ARROW WITH HORIZONTAL STROKE |
| U+2909 | UPWARDS ARROW WITH HORIZONTAL STROKE |
| U+290A | UPWARDS TRIPLE ARROW |
| U+290B | DOWNWARDS TRIPLE ARROW |
| U+290C | LEFTWARDS DOUBLE DASH ARROW |
| U+290D | RIGHTWARDS DOUBLE DASH ARROW |
| U+290E | LEFTWARDS TRIPLE DASH ARROW |
| U+290F | RIGHTWARDS TRIPLE DASH ARROW |
| U+2910 | RIGHTWARDS TWO-HEADED TRIPLE DASH ARROW |
| U+2911 | RIGHTWARDS ARROW WITH DOTTED STEM |
| U+2912 | UPWARDS ARROW TO BAR |
| U+2913 | DOWNWARDS ARROW TO BAR |
| U+2914 | RIGHTWARDS ARROW WITH TAIL WITH VERTICAL STROKE |
| U+2915 | RIGHTWARDS ARROW WITH TAIL WITH DOUBLE VERTICAL STROKE |
| U+2916 | RIGHTWARDS TWO-HEADED ARROW WITH TAIL |
| U+2917 | RIGHTWARDS TWO-HEADED ARROW WITH TAIL WITH VERTICAL STROKE |
| U+2918 | RIGHTWARDS TWO-HEADED ARROW WITH TAIL WITH DOUBLE VERTICAL STROKE |
| U+2919 | LEFTWARDS ARROW-TAIL |
| U+291A | RIGHTWARDS ARROW-TAIL |
| U+291B | LEFTWARDS DOUBLE ARROW-TAIL |

| | |
|---|---|
| U+291C | RIGHTWARDS DOUBLE ARROW-TAIL |
| U+291D | LEFTWARDS ARROW TO BLACK DIAMOND |
| U+291E | RIGHTWARDS ARROW TO BLACK DIAMOND |
| U+291F | LEFTWARDS ARROW FROM BAR TO BLACK DIAMOND |
| U+2920 | RIGHTWARDS ARROW FROM BAR TO BLACK DIAMOND |
| U+2921 | NORTH WEST AND SOUTH EAST ARROW |
| U+2922 | NORTH EAST AND SOUTH WEST ARROW |
| U+2923 | NORTH WEST ARROW WITH HOOK |
| U+2924 | NORTH EAST ARROW WITH HOOK |
| U+2925 | SOUTH EAST ARROW WITH HOOK |
| U+2926 | SOUTH WEST ARROW WITH HOOK |
| U+2927 | NORTH WEST ARROW AND NORTH EAST ARROW |
| U+2928 | NORTH EAST ARROW AND SOUTH EAST ARROW |
| U+2929 | SOUTH EAST ARROW AND SOUTH WEST ARROW |
| U+292A | SOUTH WEST ARROW AND NORTH WEST ARROW |
| U+292B | RISING DIAGONAL CROSSING FALLING DIAGONAL |
| U+292C | FALLING DIAGONAL CROSSING RISING DIAGONAL |
| U+292D | SOUTH EAST ARROW CROSSING NORTH EAST ARROW |
| U+292E | NORTH EAST ARROW CROSSING SOUTH EAST ARROW |
| U+292F | FALLING DIAGONAL CROSSING NORTH EAST ARROW |
| U+2930 | RISING DIAGONAL CROSSING SOUTH EAST ARROW |
| U+2931 | NORTH EAST ARROW CROSSING NORTH WEST ARROW |
| U+2932 | NORTH WEST ARROW CROSSING NORTH EAST ARROW |
| U+2933 | WAVE ARROW POINTING DIRECTLY RIGHT |
| U+2934 | ARROW POINTING RIGHTWARDS THEN CURVING UPWARDS |
| U+2935 | ARROW POINTING RIGHTWARDS THEN CURVING DOWNWARDS |
| U+2936 | ARROW POINTING DOWNWARDS THEN CURVING LEFTWARDS |
| U+2937 | ARROW POINTING DOWNWARDS THEN CURVING RIGHTWARDS |
| U+2938 | RIGHT-SIDE ARC CLOCKWISE ARROW |
| U+2939 | LEFT-SIDE ARC ANTICLOCKWISE ARROW |
| U+293A | TOP ARC ANTICLOCKWISE ARROW |
| U+293B | BOTTOM ARC ANTICLOCKWISE ARROW |
| U+293C | TOP ARC CLOCKWISE ARROW WITH MINUS |
| U+293D | TOP ARC ANTICLOCKWISE ARROW WITH PLUS |
| U+293E | LOWER RIGHT SEMICIRCULAR CLOCKWISE ARROW |
| U+293F | LOWER LEFT SEMICIRCULAR ANTICLOCKWISE ARROW |
| U+2940 | ANTICLOCKWISE CLOSED CIRCLE ARROW |
| U+2941 | CLOCKWISE CLOSED CIRCLE ARROW |
| U+2942 | RIGHTWARDS ARROW ABOVE SHORT LEFTWARDS ARROW |
| U+2943 | LEFTWARDS ARROW ABOVE SHORT RIGHTWARDS ARROW |
| U+2944 | SHORT RIGHTWARDS ARROW ABOVE LEFTWARDS ARROW |
| U+2945 | RIGHTWARDS ARROW WITH PLUS BELOW |
| U+2946 | LEFTWARDS ARROW WITH PLUS BELOW |
| U+2947 | RIGHTWARDS ARROW THROUGH X |
| U+2948 | LEFT RIGHT ARROW THROUGH SMALL CIRCLE |
| U+2949 | UPWARDS TWO-HEADED ARROW FROM SMALL CIRCLE |
| U+294A | LEFT BARB UP RIGHT BARB DOWN HARPOON |
| U+294B | LEFT BARB DOWN RIGHT BARB UP HARPOON |
| U+294C | UP BARB RIGHT DOWN BARB LEFT HARPOON |
| U+294D | UP BARB LEFT DOWN BARB RIGHT HARPOON |
| U+294E | LEFT BARB UP RIGHT BARB UP HARPOON |
| U+294F | UP BARB RIGHT DOWN BARB RIGHT HARPOON |

| | |
|---|---|
| U+2950 | LEFT BARB DOWN RIGHT BARB DOWN HARPOON |
| U+2951 | UP BARB LEFT DOWN BARB LEFT HARPOON |
| U+2952 | LEFTWARDS HARPOON WITH BARB UP TO BAR |
| U+2953 | RIGHTWARDS HARPOON WITH BARB UP TO BAR |
| U+2954 | UPWARDS HARPOON WITH BARB RIGHT TO BAR |
| U+2955 | DOWNWARDS HARPOON WITH BARB RIGHT TO BAR |
| U+2956 | LEFTWARDS HARPOON WITH BARB DOWN TO BAR |
| U+2957 | RIGHTWARDS HARPOON WITH BARB DOWN TO BAR |
| U+2958 | UPWARDS HARPOON WITH BARB LEFT TO BAR |
| U+2959 | DOWNWARDS HARPOON WITH BARB LEFT TO BAR |
| U+295A | LEFTWARDS HARPOON WITH BARB UP FROM BAR |
| U+295B | RIGHTWARDS HARPOON WITH BARB UP FROM BAR |
| U+295C | UPWARDS HARPOON WITH BARB RIGHT FROM BAR |
| U+295D | DOWNWARDS HARPOON WITH BARB RIGHT FROM BAR |
| U+295E | LEFTWARDS HARPOON WITH BARB DOWN FROM BAR |
| U+295F | RIGHTWARDS HARPOON WITH BARB DOWN FROM BAR |
| U+2960 | UPWARDS HARPOON WITH BARB LEFT FROM BAR |
| U+2961 | DOWNWARDS HARPOON WITH BARB LEFT FROM BAR |
| U+2962 | LEFTWARDS HARPOON WITH BARB UP ABOVE LEFTWARDS HARPOON WITH BARB DOWN |
| U+2963 | UPWARDS HARPOON WITH BARB LEFT BESIDE UPWARDS HARPOON WITH BARB RIGHT |
| U+2964 | RIGHTWARDS HARPOON WITH BARB UP ABOVE RIGHTWARDS HARPOON WITH BARB DOWN |
| U+2965 | DOWNWARDS HARPOON WITH BARB LEFT BESIDE DOWNWARDS HARPOON WITH BARB RIGHT |
| U+2966 | LEFTWARDS HARPOON WITH BARB UP ABOVE RIGHTWARDS HARPOON WITH BARB UP |
| U+2967 | LEFTWARDS HARPOON WITH BARB DOWN ABOVE RIGHTWARDS HARPOON WITH BARB DOWN |
| U+2968 | RIGHTWARDS HARPOON WITH BARB UP ABOVE LEFTWARDS HARPOON WITH BARB UP |
| U+2969 | RIGHTWARDS HARPOON WITH BARB DOWN ABOVE LEFTWARDS HARPOON WITH BARB DOWN |
| U+296A | LEFTWARDS HARPOON WITH BARB UP ABOVE LONG DASH |
| U+296B | LEFTWARDS HARPOON WITH BARB DOWN BELOW LONG DASH |
| U+296C | RIGHTWARDS HARPOON WITH BARB UP ABOVE LONG DASH |
| U+296D | RIGHTWARDS HARPOON WITH BARB DOWN BELOW LONG DASH |
| U+296E | UPWARDS HARPOON WITH BARB LEFT BESIDE DOWNWARDS HARPOON WITH BARB RIGHT |
| U+296F | DOWNWARDS HARPOON WITH BARB LEFT BESIDE UPWARDS HARPOON WITH BARB RIGHT |
| U+2970 | RIGHT DOUBLE ARROW WITH ROUNDED HEAD |
| U+2971 | EQUALS SIGN ABOVE RIGHTWARDS ARROW |
| U+2972 | TILDE OPERATOR ABOVE RIGHTWARDS ARROW |
| U+2973 | LEFTWARDS ARROW ABOVE TILDE OPERATOR |
| U+2974 | RIGHTWARDS ARROW ABOVE TILDE OPERATOR |
| U+2975 | RIGHTWARDS ARROW ABOVE ALMOST EQUAL TO |
| U+2976 | LESS-THAN ABOVE LEFTWARDS ARROW |
| U+2977 | LEFTWARDS ARROW THROUGH LESS-THAN |
| U+2978 | GREATER-THAN ABOVE RIGHTWARDS ARROW |
| U+2979 | SUBSET ABOVE RIGHTWARDS ARROW |
| U+297A | LEFTWARDS ARROW THROUGH SUBSET |
| U+297B | SUPERSET ABOVE LEFTWARDS ARROW |
| U+297C | LEFT FISH TAIL |
| U+297D | RIGHT FISH TAIL |
| U+297E | UP FISH TAIL |
| U+297F | DOWN FISH TAIL |
| U+2980 | TRIPLE VERTICAL BAR DELIMITER |
| U+2981 | Z NOTATION SPOT |
| U+2982 | Z NOTATION TYPE COLON |
| U+2999 | DOTTED FENCE |

| | |
|---|---|
| U+299A | VERTICAL ZIGZAG LINE |
| U+299B | MEASURED ANGLE OPENING LEFT |
| U+299C | RIGHT ANGLE VARIANT WITH SQUARE |
| U+299D | MEASURED RIGHT ANGLE WITH DOT |
| U+299E | ANGLE WITH S INSIDE |
| U+299F | ACUTE ANGLE |
| U+29A0 | SPHERICAL ANGLE OPENING LEFT |
| U+29A1 | SPHERICAL ANGLE OPENING UP |
| U+29A2 | TURNED ANGLE |
| U+29A3 | REVERSED ANGLE |
| U+29A4 | ANGLE WITH UNDERBAR |
| U+29A5 | REVERSED ANGLE WITH UNDERBAR |
| U+29A6 | OBLIQUE ANGLE OPENING UP |
| U+29A7 | OBLIQUE ANGLE OPENING DOWN |
| U+29A8 | MEASURED ANGLE WITH OPEN ARM ENDING IN ARROW POINTING UP AND RIGHT |
| U+29A9 | MEASURED ANGLE WITH OPEN ARM ENDING IN ARROW POINTING UP AND LEFT |
| U+29AA | MEASURED ANGLE WITH OPEN ARM ENDING IN ARROW POINTING DOWN AND RIGHT |
| U+29AB | MEASURED ANGLE WITH OPEN ARM ENDING IN ARROW POINTING DOWN AND LEFT |
| U+29AC | MEASURED ANGLE WITH OPEN ARM ENDING IN ARROW POINTING RIGHT AND UP |
| U+29AD | MEASURED ANGLE WITH OPEN ARM ENDING IN ARROW POINTING LEFT AND UP |
| U+29AE | MEASURED ANGLE WITH OPEN ARM ENDING IN ARROW POINTING RIGHT AND DOWN |
| U+29AF | MEASURED ANGLE WITH OPEN ARM ENDING IN ARROW POINTING LEFT AND DOWN |
| U+29B0 | REVERSED EMPTY SET |
| U+29B1 | EMPTY SET WITH OVERBAR |
| U+29B2 | EMPTY SET WITH SMALL CIRCLE ABOVE |
| U+29B3 | EMPTY SET WITH RIGHT ARROW ABOVE |
| U+29B4 | EMPTY SET WITH LEFT ARROW ABOVE |
| U+29B5 | CIRCLE WITH HORIZONTAL BAR |
| U+29B6 | CIRCLED VERTICAL BAR |
| U+29B7 | CIRCLED PARALLEL |
| U+29B9 | CIRCLED PERPENDICULAR |
| U+29BA | CIRCLE DIVIDED BY HORIZONTAL BAR AND TOP HALF DIVIDED BY VERTICAL BAR |
| U+29BB | CIRCLE WITH SUPERIMPOSED X |
| U+29BD | UP ARROW THROUGH CIRCLE |
| U+29BE | CIRCLED WHITE BULLET |
| U+29BF | CIRCLED BULLET |
| U+29C2 | CIRCLE WITH SMALL CIRCLE TO THE RIGHT |
| U+29C3 | CIRCLE WITH TWO HORIZONTAL STROKES TO THE RIGHT |
| U+29C5 | SQUARED FALLING DIAGONAL SLASH |
| U+29C7 | SQUARED SMALL CIRCLE |
| U+29C8 | SQUARED SQUARE |
| U+29C9 | TWO JOINED SQUARES |
| U+29CA | TRIANGLE WITH DOT ABOVE |
| U+29CB | TRIANGLE WITH UNDERBAR |
| U+29CC | S IN TRIANGLE |
| U+29CD | TRIANGLE WITH SERIFS AT BOTTOM |
| U+29CE | RIGHT TRIANGLE ABOVE LEFT TRIANGLE |
| U+29CF | LEFT TRIANGLE BESIDE VERTICAL BAR |
| U+29D0 | VERTICAL BAR BESIDE RIGHT TRIANGLE |
| U+29D1 | BOWTIE WITH LEFT HALF BLACK |
| U+29D2 | BOWTIE WITH RIGHT HALF BLACK |
| U+29D3 | BLACK BOWTIE |

| | | | |
|---|---|---|---|
| U+29D6 | WHITE HOURGLASS | | |
| U+29D7 | BLACK HOURGLASS | | |
| U+29DC | INCOMPLETE INFINITY | | |
| U+29DD | TIE OVER INFINITY | | |
| U+29DE | INFINITY NEGATED WITH VERTICAL BAR | | |
| U+29DF | DOUBLE-ENDED MULTIMAP | | |
| U+29E0 | SQUARE WITH CONTOURED OUTLINE | | |
| U+29E1 | INCREASES AS | | |
| U+29E2 | SHUFFLE PRODUCT | | |
| U+29E6 | GLEICH STARK | | |
| U+29E7 | THERMODYNAMIC | | |
| U+29E8 | DOWN-POINTING TRIANGLE WITH LEFT HALF BLACK | | |
| U+29E9 | DOWN-POINTING TRIANGLE WITH RIGHT HALF BLACK | | |
| U+29EA | BLACK DIAMOND WITH DOWN ARROW | | |
| U+29EB | BLACK LOZENGE | | |
| U+29EC | WHITE CIRCLE WITH DOWN ARROW | | |
| U+29ED | BLACK CIRCLE WITH DOWN ARROW | | |
| U+29EE | ERROR-BARRED WHITE SQUARE | | |
| U+29EF | ERROR-BARRED BLACK SQUARE | | |
| U+29F0 | ERROR-BARRED WHITE DIAMOND | | |
| U+29F1 | ERROR-BARRED BLACK DIAMOND | | |
| U+29F2 | ERROR-BARRED WHITE CIRCLE | | |
| U+29F3 | ERROR-BARRED BLACK CIRCLE | | |
| U+29F4 | RULE-DELAYED | | |
| U+29F6 | SOLIDUS WITH OVERBAR | | |
| U+29F7 | REVERSE SOLIDUS WITH HORIZONTAL STROKE | | |
| U+29FA | DOUBLE PLUS | | ++ |
| U+29FB | TRIPLE PLUS | | +++ |
| U+29FE | TINY | | |
| U+29FF | MINY | | |
| U+2A00 | N-ARY CIRCLED DOT OPERATOR | ⨀ | BIGODOT |
| U+2A01 | N-ARY CIRCLED PLUS OPERATOR | ⨁ | BIGOPLUS |
| U+2A02 | N-ARY CIRCLED TIMES OPERATOR | ⨂ | BIGOTIMES |
| U+2A03 | N-ARY UNION OPERATOR WITH DOT | | BIGUDOT |
| U+2A04 | N-ARY UNION OPERATOR WITH PLUS | | BIGUPLUS |
| U+2A05 | N-ARY SQUARE INTERSECTION OPERATOR | | BIGSQCAP |
| U+2A06 | N-ARY SQUARE UNION OPERATOR | | BIGSQCUP |
| U+2A07 | TWO LOGICAL AND OPERATOR | | |
| U+2A08 | TWO LOGICAL OR OPERATOR | | |
| U+2A09 | N-ARY TIMES OPERATOR | | BIGTIMES |
| U+2A0A | MODULO TWO SUM | | |
| U+2A10 | CIRCULATION FUNCTION | | |
| U+2A11 | ANTICLOCKWISE INTEGRATION | | |
| U+2A12 | LINE INTEGRATION WITH RECTANGULAR PATH AROUND POLE | | |
| U+2A13 | LINE INTEGRATION WITH SEMICIRCULAR PATH AROUND POLE | | |
| U+2A14 | LINE INTEGRATION NOT INCLUDING THE POLE | | |
| U+2A1D | JOIN | ⨝ | JOIN |
| U+2A1E | LARGE LEFT TRIANGLE OPERATOR | | |
| U+2A1F | Z NOTATION SCHEMA COMPOSITION | | |
| U+2A20 | Z NOTATION SCHEMA PIPING | | |
| U+2A21 | Z NOTATION SCHEMA PROJECTION | | |
| U+2A32 | SEMIDIRECT PRODUCT WITH BOTTOM CLOSED | | |

| | |
|---|---|
| U+2A33 | SMASH PRODUCT |
| U+2A3C | INTERIOR PRODUCT |
| U+2A3D | RIGHTHAND INTERIOR PRODUCT |
| U+2A3E | Z NOTATION RELATIONAL COMPOSITION |
| U+2A3F | AMALGAMATION OR COPRODUCT |
| U+2A57 | SLOPING LARGE OR |
| U+2A58 | SLOPING LARGE AND |
| U+2A61 | SMALL VEE WITH UNDERBAR |
| U+2A64 | Z NOTATION DOMAIN ANTIRESTRICTION |
| U+2A65 | Z NOTATION RANGE ANTIRESTRICTION |
| U+2A68 | TRIPLE HORIZONTAL BAR WITH DOUBLE VERTICAL STROKE |
| U+2A69 | TRIPLE HORIZONTAL BAR WITH TRIPLE VERTICAL STROKE |
| U+2A6A | TILDE OPERATOR WITH DOT ABOVE |
| U+2A6B | TILDE OPERATOR WITH RISING DOTS |
| U+2A6D | CONGRUENT WITH DOT ABOVE |
| U+2ACD | SQUARE LEFT OPEN BOX OPERATOR |
| U+2ACE | SQUARE RIGHT OPEN BOX OPERATOR |
| U+2AD9 | ELEMENT OF OPENING DOWNWARDS |
| U+2ADA | PITCHFORK WITH TEE TOP |
| U+2ADC | FORKING |
| U+2ADD | NONFORKING |
| U+2ADE | SHORT LEFT TACK |
| U+2ADF | SHORT DOWN TACK |
| U+2AE0 | SHORT UP TACK |
| U+2AE1 | PERPENDICULAR WITH S |
| U+2AE2 | VERTICAL BAR TRIPLE RIGHT TURNSTILE |
| U+2AE3 | DOUBLE VERTICAL BAR LEFT TURNSTILE |
| U+2AE4 | VERTICAL BAR DOUBLE LEFT TURNSTILE |
| U+2AE5 | DOUBLE VERTICAL BAR DOUBLE LEFT TURNSTILE |
| U+2AE6 | LONG DASH FROM LEFT MEMBER OF DOUBLE VERTICAL |
| U+2AE7 | SHORT DOWN TACK WITH OVERBAR |
| U+2AE8 | SHORT UP TACK WITH UNDERBAR |
| U+2AE9 | SHORT UP TACK ABOVE SHORT DOWN TACK |
| U+2AEA | DOUBLE DOWN TACK |
| U+2AEB | DOUBLE UP TACK |
| U+2AEC | DOUBLE STROKE NOT SIGN |
| U+2AED | REVERSED DOUBLE STROKE NOT SIGN |
| U+2AEE | DOES NOT DIVIDE WITH REVERSED NEGATION SLASH |
| U+2AEF | VERTICAL LINE WITH CIRCLE ABOVE |
| U+2AF0 | VERTICAL LINE WITH CIRCLE BELOW |
| U+2AF1 | DOWN TACK WITH CIRCLE BELOW |
| U+2AF2 | PARALLEL WITH HORIZONTAL STROKE |
| U+2AF3 | PARALLEL WITH TILDE OPERATOR |
| U+2AF5 | TRIPLE VERTICAL BAR WITH HORIZONTAL STROKE |
| U+2AF6 | TRIPLE COLON OPERATOR |
| U+2AFC | LARGE TRIPLE VERTICAL BAR OPERATOR |
| U+2AFE | WHITE VERTICAL BAR |
| U+2AFF | N-ARY WHITE VERTICAL BAR |

# Appendix D

# Simplified Grammar for Application Programmers and Library Writers

In this chapter, we provide a simplified grammar of the Fortress concrete syntax as documentation, to enable Fortress programmers to understand it more easily. It includes unimplemented Fortress language features such as dimensions and units, tests and properties, coercions, and where clauses. The full Fortress grammar is described in Appendix E.

The simplified grammar does not describe the details of the uses of operators, whitespaces, and semicolons. Instead, they are described as follows according to three different contexts influencing the whitespace-sensitivity of expressions:

**statement** Expressions immediately enclosed by an expression block are in a statement-like context. Multiple expressions can appear on a line if they are separated (or terminated) by semicolons. If an expression can legally end at the end of a line, it does; if it cannot, it does not. A prefix or infix operator that lacks its last operand prevents an expression from ending. For example,

```
$an = expression+$
      $spanning+$
      $four+$
      $lines$
$a = oneLiner$
$four(); on(); one(); line();$
```

**nested** An expression or list of expressions immediately enclosed by parentheses or braces is nested. Multiple expressions are separated by commas, and the end of a line does not end an expression. Because of this effect, the meaning of a several lines of code can change if they are wrapped in parentheses. Parentheses can also be used to ensure that a multiline expression is not terminated prematurely without paying special attention to line endings.

```
$lhs = rhs$
$-aSeparateExpression$

$postProfit(revenue$
            $-expenses)$
```

**pasted** Fortress has special syntax for matrix pasting. Within square brackets, whitespace-separated expressions are treated (depending on their type) as either matrix elements or submatrices within a row. Because whitespace is the separator, it also ends expressions where possible. In addition, newline-or-semicolon-separated rows are pasted vertically along their columns. Higher-dimensional pasting is expressed with repeated semicolons, but repeated newlines do not have the same effect.

$$\begin{aligned}
&id2a = [1\ 0; 0\ 1\,]\\
&id2b = [1\ 0;\\
&\qquad\quad 0\ 1\,]\\
&id2c = [1\ 0\\
&\qquad\quad 0\ 1\,]\\
&cube_2 = [1\ 0; 0\ 1; ; 1\ -1; 1\ 1\,]
\end{aligned}$$

The simplified grammar is written in the extended BNF form with the following three postfix operators:

- ?: which means that its argument (the symbol or group of symbols in parentheses to the left of the operator) can appear zero or one time
- $*$: which means that its argument can appear zero or more times
- $+$: which means that its argument can appear one or more times

## D.1 Components and APIs

| | | |
|---|---|---|
| *File* | ::= | *CompilationUnit* |
| | \| | *Imports*? *Exports Decls*? |
| | \| | *Imports*? *AbsDecls* |
| | \| | *Imports AbsDecls*? |
| *CompilationUnit* | ::= | *Component* |
| | \| | *Api* |
| *Component* | ::= | `native` ? `component` *APIName Imports*? *Exports Decls*? `end` |
| *Api* | ::= | `native` ? `api` *APIName Imports*? *AbsDecls*? `end` |
| *Imports* | ::= | *Import*$^+$ |
| *Import* | ::= | `import` *ImportedNames* |
| | \| | `import api` *AliasedAPINames* |
| *ImportedNames* | ::= | *APIName* . { ... } ( `except` *SimpleNames*)? |
| | \| | *APIName* . { *AliasedSimpleNameList* ( , ... )? } |
| | \| | *QualifiedName* ( `as` *Id*)? |
| *SimpleNames* | ::= | *SimpleName* |
| | \| | { *SimpleNameList* } |
| *SimpleNameList* | ::= | *SimpleName*( , *SimpleName*)$^*$ |
| *AliasedSimpleName* | ::= | *Id* ( `as` *Id*)? |
| | \| | `opr` *Op* ( `as` *Op*)? |
| | \| | `opr` *EncloserPair* ( `as` *EncloserPair*)? |
| *AliasedSimpleNameList* | ::= | *AliasedSimpleName*( , *AliasedSimpleName*)$^*$ |
| *AliasedAPINames* | ::= | *AliasedAPIName* |
| | \| | { *AliasedAPINameList* } |
| *AliasedAPIName* | ::= | *APIName* ( `as` *Id*)? |
| *AliasedAPINameList* | ::= | *AliasedAPIName*( , *AliasedAPIName*)$^*$ |
| *Exports* | ::= | *Export*$^+$ |
| *Export* | ::= | `export` *APINames* |
| *APINames* | ::= | *APIName* |
| | \| | { *APINameList* } |
| *APINameList* | ::= | *APIName*( , *APIName*)$^*$ |

## D.2 Top-level Declarations

| | | |
|---|---|---|
| *Decls* | ::= | *Decl*$^+$ |
| *Decl* | ::= | *TraitDecl* |
| | \| | *ObjectDecl* |
| | \| | *VarDecl* |
| | \| | *FnDecl* |
| | \| | *DimUnitDecl* |
| | \| | *TypeAlias* |
| | \| | *TestDecl* |
| | \| | *PropertyDecl* |
| | \| | *ExternalSyntax* |
| *AbsDecls* | ::= | *AbsDecl*$^+$ |
| *AbsDecl* | ::= | *AbsTraitDecl* |
| | \| | *AbsObjectDecl* |
| | \| | *AbsVarDecl* |
| | \| | *AbsFnDecl* |
| | \| | *DimUnitDecl* |
| | \| | *TypeAlias* |
| | \| | *TestDecl* |
| | \| | *PropertyDecl* |
| | \| | *AbsExternalSyntax* |

## D.3 Trait and Object Declarations

A *TraitHeaderFront* may have 0 or 1 of each *TraitClause*.

| | | |
|---|---|---|
| *TraitDecl* | ::= | *TraitMods*? *TraitHeaderFront TraitClauses GoInATrait*? `end` |
| *TraitHeaderFront* | ::= | `trait` *Id StaticParams*? *ExtendsWhere*? |
| *TraitClauses* | ::= | *TraitClause*$^*$ |
| *TraitClause* | ::= | *Excludes* |
| | \| | *Comprises* |
| | \| | *Where* |
| *GoInATrait* | ::= | *Coercions*? *GoFrontInATrait GoBackInATrait*? |
| | \| | *Coercions*? *GoBackInATrait* |
| | \| | *Coercions* |
| *Coercions* | ::= | *Coercion*$^+$ |
| *GoFrontInATrait* | ::= | *GoesFrontInATrait*$^+$ |
| *GoesFrontInATrait* | ::= | *AbsFldDecl* |
| | \| | *GetterSetterDecl* |
| | \| | *PropertyDecl* |
| *GoBackInATrait* | ::= | *GoesBackInATrait*$^+$ |
| *GoesBackInATrait* | ::= | *MdDecl* |
| | \| | *PropertyDecl* |
| *ObjectDecl* | ::= | *ObjectMods*? *ObjectHeader GoInAnObject*? `end` |
| *ObjectHeader* | ::= | `object` *Id StaticParams*? *ObjectValParam*? *ExtendsWhere*? *FnClauses* |

| | | |
|---|---|---|
| *ObjectValParam* | ::= | ( *ObjectParams*? ) |
| *ObjectParams* | ::= | (*ObjectParam* ,)$^*$ (*ObjectVarargs* ,)? *ObjectKeyword*( , *ObjectKeyword*)$^*$ |
| | $\mid$ | (*ObjectParam* ,)$^*$ *ObjectVarargs* |
| | $\mid$ | *ObjectParam* ( , *ObjectParam*)$^*$ |
| *ObjectVarargs* | ::= | `transient` *Varargs* |
| *ObjectKeyword* | ::= | *ObjectParam* = *Expr* |
| *ObjectParam* | ::= | *ParamFldMods*? *Param* |
| | $\mid$ | `transient` *Param* |
| *GoInAnObject* | ::= | *Coercions*? *GoFrontInAnObject* *GoBackInAnObject*? |
| | $\mid$ | *Coercions*? *GoBackInAnObject* |
| | $\mid$ | *Coercions* |
| *GoFrontInAnObject* | ::= | *GoesFrontInAnObject*$^+$ |
| *GoesFrontInAnObject* | ::= | *FldDecl* |
| | $\mid$ | *GetterSetterDef* |
| | $\mid$ | *PropertyDecl* |
| *GoBackInAnObject* | ::= | *GoesBackInAnObject*$^+$ |
| *GoesBackInAnObject* | ::= | *MdDef* |
| | $\mid$ | *PropertyDecl* |
| *AbsTraitDecl* | ::= | *AbsTraitMods*? *TraitHeaderFront* *AbsTraitClauses* *AbsGoInATrait*? `end` |
| *AbsTraitClauses* | ::= | *AbsTraitClause*$^*$ |
| *AbsTraitClause* | ::= | *Excludes* |
| | $\mid$ | *AbsComprises* |
| | $\mid$ | *Where* |
| *AbsGoInATrait* | ::= | *AbsCoercions*? *AbsGoFrontInATrait* *AbsGoBackInATrait*? |
| | $\mid$ | *AbsCoercions*? *AbsGoBackInATrait* |
| | $\mid$ | *AbsCoercions* |
| *AbsCoercions* | ::= | *AbsCoercion*$^+$ |
| *AbsGoFrontInATrait* | ::= | *AbsGoesFrontInATrait*$^+$ |
| *AbsGoesFrontInATrait* | ::= | *ApiFldDecl* |
| | $\mid$ | *AbsGetterSetterDecl* |
| | $\mid$ | *PropertyDecl* |
| *AbsGoBackInATrait* | ::= | *AbsGoesBackInATrait*$^+$ |
| *AbsGoesBackInATrait* | ::= | *AbsMdDecl* |
| | $\mid$ | *PropertyDecl* |
| *AbsObjectDecl* | ::= | *AbsObjectMods*? *ObjectHeader* *AbsGoInAnObject*? `end` |
| *AbsGoInAnObject* | ::= | *AbsCoercions*? *AbsGoFrontInAnObject* *AbsGoBackInAnObject*? |
| | $\mid$ | *AbsCoercions*? *AbsGoBackInAnObject* |
| | $\mid$ | *AbsCoercions* |
| *AbsGoFrontInAnObject* | ::= | *AbsGoesFrontInAnObject*$^+$ |
| *AbsGoesFrontInAnObject* | ::= | *ApiFldDecl* |
| | $\mid$ | *AbsGetterSetterDecl* |
| | $\mid$ | *PropertyDecl* |
| *AbsGoBackInAnObject* | ::= | *AbsGoesBackInAnObject*$^+$ |
| *AbsGoesBackInAnObject* | ::= | *AbsMdDecl* |
| | $\mid$ | *PropertyDecl* |

## D.4 Variable Declarations

| | | |
|---|---|---|
| *VarDecl* | ::= | *VarMods*? *VarWTypes InitVal* |
| | \| | *VarMods*? *BindIdOrBindIdTuple* = *Expr* |
| | \| | *VarMods*? *BindIdOrBindIdTuple* : *Type* ... *InitVal* |
| | \| | *VarMods*? *BindIdOrBindIdTuple* : *TupleType InitVal* |
| *VarMods* | ::= | *VarMod*$^+$ |
| *VarMod* | ::= | *AbsVarMod* \| `private` |
| *VarWTypes* | ::= | *VarWType* |
| | \| | ( *VarWType*( , *VarWType*)$^+$ ) |
| *VarWType* | ::= | *BindId IsType* |
| *InitVal* | ::= | ( = \| := ) *Expr* |
| *AbsVarDecl* | ::= | *AbsVarMods*? *VarWTypes* |
| | \| | *AbsVarMods*? *BindIdOrBindIdTuple* : *Type* ... |
| | \| | *AbsVarMods*? *BindIdOrBindIdTuple* : *TupleType* |
| *AbsVarMods* | ::= | *AbsVarMod*$^+$ |
| *AbsVarMod* | ::= | `var` \| `test` |

## D.5 Function Declarations

| | | |
|---|---|---|
| *FnDecl* | ::= | *FnMods*? *FnHeaderFront FnHeaderClause* (= *Expr*)? |
| | \| | *FnSig* |
| *FnSig* | ::= | *SimpleName* : *Type* |
| *AbsFnDecl* | ::= | *AbsFnMods*? *FnHeaderFront FnHeaderClause* |
| | \| | *FnSig* |
| *FnHeaderFront* | ::= | *NamedFnHeaderFront* |
| | \| | *OpHeaderFront* |
| *NamedFnHeaderFront* | ::= | *Id StaticParams*? *ValParam* |

## D.6 Dimension, Unit, Type Alias, Test, Property, and External Syntax Declarations

| | | |
|---|---|---|
| *DimUnitDecl* | ::= | `dim` *Id* (= *Type*)? (`unit` \| `SI_unit`) *Id*$^+$ (= *Expr*)? |
| | \| | `dim` *Id* (= *Type*)? (`default` *Id*)? |
| | \| | (`unit` \| `SI_unit`) *Id*$^+$ (: *Type*)? (= *Expr*)? |
| *TypeAlias* | ::= | `type` *Id StaticParams*? = *Type* |
| *TestDecl* | ::= | `test` *Id* [*GeneratorClauseList*] = *Expr* |
| *PropertyDecl* | ::= | `property` (*Id* =)? (∀ *ValParam*)? *Expr* |
| *ExternalSyntax* | ::= | `syntax` *OpenExpander Id CloseExpander* = *Expr* |
| *OpenExpander* | ::= | *Id* |
| | \| | *LeftEncloser* |
| | \| | *Encloser* |
| *CloseExpander* | ::= | *Id* |
| | \| | *RightEncloser* |
| | \| | *Encloser* |
| | \| | `end` |
| *AbsExternalSyntax* | ::= | `syntax` *OpenExpander Id CloseExpander* |

## D.7 Headers

Each modifier should not appear multiple times in a declaration.

| | | |
|---|---|---|
| *IsType* | ::= | : *Type* |
| *ExtendsWhere* | ::= | `extends` *TraitTypeWheres* |
| *TraitTypeWheres* | ::= | *TraitTypeWhere* |
| | \| | { *TraitTypeWhereList* } |
| *TraitTypeWhereList* | ::= | *TraitTypeWhere*( , *TraitTypeWhere*)$^*$ |
| *TraitTypeWhere* | ::= | *TraitType Where*? |
| *Extends* | ::= | `extends` *TraitTypes* |
| *Excludes* | ::= | `excludes` *TraitTypes* |
| *Comprises* | ::= | `comprises` *TraitTypes* |
| *AbsComprises* | ::= | `comprises` *ComprisingTypes* |
| *TraitTypes* | ::= | *TraitType* |
| | \| | { *TraitTypeList* } |
| *TraitTypeList* | ::= | *TraitType*( , *TraitType*)$^*$ |
| *ComprisingTypes* | ::= | *TraitType* |
| | \| | { *ComprisingTypeList* } |
| *ComprisingTypeList* | ::= | . . . |
| | \| | *TraitType*( , *TraitType*)$^*$ ( , . . . )? |
| *Where* | ::= | `where` ⟦ *WhereBindingList* ⟧ ({ *WhereConstraintList* })? |
| | \| | `where` { *WhereConstraintList* } |
| *WhereBindingList* | ::= | *WhereBinding*( , *WhereBinding*)$^*$ |
| *WhereBinding* | ::= | *Id Extends*? |
| | \| | `nat` *Id* |
| | \| | `int` *Id* |
| | \| | `bool` *Id* |
| | \| | `unit` *Id* |
| *WhereConstraintList* | ::= | *WhereConstraint*( , *WhereConstraint*)$^*$ |
| *WhereConstraint* | ::= | *Id Extends* |
| | \| | *TypeAlias* |
| | \| | *Type* `coerces` *Type* |
| | \| | *Type* `widens` *Type* |
| | \| | *UnitConstraint* |
| | \| | *QualifiedName* = *QualifiedName* |
| | \| | *IntConstraint* |
| | \| | *BoolConstraint* |
| *UnitConstraint* | ::= | dimensionless = *Id* |
| | \| | *Id* = dimensionless |
| *IntConstraint* | ::= | *IntExpr* ≤ *IntExpr* |
| | \| | *IntExpr* < *IntExpr* |
| | \| | *IntExpr* ≥ *IntExpr* |
| | \| | *IntExpr* > *IntExpr* |
| | \| | *IntExpr* = *IntExpr* |
| *IntVal* | ::= | *IntLiteralExpr* |
| | \| | *QualifiedName* |

| | | |
|---|---|---|
| *IntExpr* | ::= | *IntVal* |
| | $\mid$ | *IntExpr* + *IntExpr* |
| | $\mid$ | *IntExpr* − *IntExpr* |
| | $\mid$ | *IntExpr* · *IntExpr* |
| | $\mid$ | *IntExpr* *IntExpr* |
| | $\mid$ | *IntExpr* ˆ *IntVal* |
| | $\mid$ | (*IntExpr*) |
| *BoolConstraint* | ::= | `NOT` *BoolExpr* |
| | $\mid$ | *BoolExpr* `OR` *BoolExpr* |
| | $\mid$ | *BoolExpr* `AND` *BoolExpr* |
| | $\mid$ | *BoolExpr* `IMPLIES` *BoolExpr* |
| | $\mid$ | *BoolExpr* = *BoolExpr* |
| *BoolVal* | ::= | *true* |
| | $\mid$ | *false* |
| | $\mid$ | *QualifiedName* |
| *BoolExpr* | ::= | *BoolVal* |
| | $\mid$ | *BoolConstraint* |
| | $\mid$ | (*BoolExpr*) |
| *FnHeaderClause* | ::= | *IsType*? *FnClauses* |
| *FnClauses* | ::= | *Throws*? *Where*? *Contract* |
| *Throws* | ::= | `throws` *MayTraitTypes* |
| *MayTraitTypes* | ::= | {} |
| | $\mid$ | *TraitTypes* |
| *Contract* | ::= | *Requires*? *Ensures*? *Invariant*? |
| *Requires* | ::= | `requires` { *ExprList*? } |
| *Ensures* | ::= | `ensures` { *EnsuresClauseList*? } |
| *EnsuresClauseList* | ::= | *EnsuresClause*( , *EnsuresClause*)* |
| *EnsuresClause* | ::= | *Expr* (`provided` *Expr*)? |
| *Invariant* | ::= | `invariant` { *ExprList*? } |
| *TraitMods* | ::= | $TraitMod^+$ |
| *TraitMod* | ::= | *AbsTraitMod* $\mid$ `private` |
| *AbsTraitMods* | ::= | $AbsTraitMod^+$ |
| *AbsTraitMod* | ::= | `value` $\mid$ `test` |
| *ObjectMods* | ::= | *TraitMods* |
| *AbsObjectMods* | ::= | *AbsTraitMods* |
| *MdMods* | ::= | $MdMod^+$ |
| *MdMod* | ::= | *FnMod* $\mid$ `override` |
| *AbsMdMods* | ::= | $AbsMdMod^+$ |
| *AbsMdMod* | ::= | *AbsFnMod* $\mid$ `override` |
| *FnMods* | ::= | $FnMod^+$ |
| *FnMod* | ::= | *AbsFnMod* $\mid$ `private` |
| *AbsFnMods* | ::= | $AbsFnMod^+$ |
| *AbsFnMod* | ::= | *LocalFnMod* $\mid$ `test` |
| *LocalFnMods* | ::= | $LocalFnMod^+$ |
| *LocalFnMod* | ::= | `atomic` $\mid$ `io` |
| *ParamFldMods* | ::= | $ParamFldMod^+$ |
| *ParamFldMod* | ::= | `var` $\mid$ `hidden` $\mid$ `settable` $\mid$ `wrapped` |
| *FldMods* | ::= | $FldMod^+$ |
| *FldMod* | ::= | `var` $\mid$ *AbsFldMod* |
| *AbsFldMods* | ::= | $AbsFldMod^+$ |
| *AbsFldMod* | ::= | *ApiFldMod* $\mid$ `wrapped` $\mid$ `private` |
| *ApiFldMods* | ::= | $ApiFldMod^+$ |
| *ApiFldMod* | ::= | `hidden` $\mid$ `settable` $\mid$ `test` |

| | | |
|---|---|---|
| *StaticParams* | ::= | ⟦*StaticParamList*⟧ |
| *StaticParamList* | ::= | *StaticParam*( , *StaticParam*)$^*$ |
| *StaticParam* | ::= | *Id Extends*? ( `absorbs unit` )? |
| | \| | `nat` *Id* |
| | \| | `int` *Id* |
| | \| | `bool` *Id* |
| | \| | `dim` *Id* |
| | \| | `unit` *Id* ( : *Type*)? ( `absorbs unit` )? |
| | \| | `opr` *Op* |
| *StaticArgs* | ::= | ⟦*StaticArgList* ⟧ |
| *StaticArgList* | ::= | *StaticArg*( , *StaticArg*)$^*$ |
| *StaticArg* | ::= | *Op* |
| | \| | Unity |
| | \| | dimensionless |
| | \| | 1 / *Type* |
| | \| | *DimPrefixOp Type* |
| | \| | ( *StaticArg* ) |
| | \| | *IntExpr* |
| | \| | *BoolExpr* |
| | \| | *Type* |
| | \| | *Expr* |

## D.8 Parameters

| | | |
|---|---|---|
| *ValParam* | ::= | *BindId* |
| | \| | (*Params*?) |
| *Params* | ::= | (*Param* , )$^*$ (*Varargs* , )? *Keyword*( , *Keyword*)$^*$ |
| | \| | (*Param* , )$^*$ *Varargs* |
| | \| | *Param*( , *Param*)$^*$ |
| *VarargsParam* | ::= | *BindId* : *Type* ... |
| *Varargs* | ::= | *VarargsParam* |
| *Keyword* | ::= | *Param* = *Expr* |
| *PlainParam* | ::= | *BindId IsType*? |
| | \| | *Type* |
| *Param* | ::= | *PlainParam* |
| *OpHeaderFront* | ::= | `opr` `BIG` ? ({ ↦ \| *LeftEncloser* \| *Encloser*) *StaticParams*? *Params* (*RightEncloser* \| *Encloser*) |
| | \| | `opr` *ValParam* (*Op* \| *ExponentOp*) *StaticParams*? |
| | \| | `opr` `BIG` ? (*Op* \| ^ \| *Encloser* \| $\sum$ \| $\prod$) *StaticParams*? *ValParam* |

## D.9 Method Declarations

| | | |
|---|---|---|
| *MdDecl* | ::= | *MdDef* |
| | \| | `abstract` ? *MdMods*? *MdHeaderFront FnHeaderClause* |
| *MdDef* | ::= | *MdMods*? *MdHeaderFront FnHeaderClause* = *Expr* |
| *AbsMdDecl* | ::= | `abstract` ? *AbsMdMods*? *MdHeaderFront FnHeaderClause* |
| *MdHeaderFront* | ::= | *NamedMdHeaderFront* |
| | \| | *OpMdHeaderFront* |
| *NamedMdHeaderFront* | ::= | *Id StaticParams*? *MdValParam* |
| *GetterSetterDecl* | ::= | *GetterSetterDef* |
| | \| | `abstract` ? *FnMods*? *GetterSetterMod MdHeaderFront FnHeaderClause* |

| | | |
|---|---|---|
| *GetterSetterDef* | ::= | *FnMods*? *GetterSetterMod MdHeaderFront FnHeaderClause* = *Expr* |
| *GetterSetterMod* | ::= | `getter` \| `setter` |
| *AbsGetterSetterDecl* | ::= | `abstract` ? *AbsFnMods*? *GetterSetterMod MdHeaderFront FnHeaderClause* |
| *Coercion* | ::= | `coerce` *StaticParams*? ( *BindId IsType* ) *CoercionClauses* `widens` ? = *Expr* |
| *AbsCoercion* | ::= | `coerce` *StaticParams*? ( *BindId IsType* ) *CoercionClauses* `widens` ? |
| *CoercionClauses* | ::= | *CoercionWhere*? *Ensures*? *Invariant*? |
| *CoercionWhere* | ::= | `where` ⟦ *WhereBindingList* ⟧ ({ *CoercionWhereConstraintList* })? |
| | \| | `where` { *CoercionWhereConstraintList* } |
| *CoercionWhereConstraintList* | ::= | *CoercionWhereConstraint*( , *CoercionWhereConstraint*)* |
| *CoercionWhereConstraint* | ::= | *WhereConstraint* |
| | \| | *Type* `widens or coerces` *Type* |

## D.10 Method Parameters

| | | |
|---|---|---|
| *MdValParam* | ::= | ( *MdParams*? ) |
| *MdParams* | ::= | (*MdParam* , )* (*Varargs* , )? *MdKeyword*( , *MdKeyword*)* |
| | \| | (*MdParam* , )* *Varargs* |
| | \| | *MdParam*( , *MdParam*)* |
| *MdKeyword* | ::= | *MdParam* = *Expr* |
| *MdParam* | ::= | *Param* |
| | \| | `self` |
| *OpMdHeaderFront* | ::= | `opr BIG` ? ({ $\mapsto$ \| *LeftEncloser* \| *Encloser*) *StaticParams*? *Params* |
| | | (*RightEncloser* \| *Encloser*) ( := ( *SubscriptAssignParam* ) )? |
| | \| | `opr` *ValParam* (*Op* \| *ExponentOp*) *StaticParams*? |
| | \| | `opr BIG` ? (*Op* \| ^ \| *Encloser* \| $\sum$ \| $\prod$) *StaticParams*? *ValParam* |
| *SubscriptAssignParam* | ::= | *Varargs* |
| | \| | *Param* |

## D.11 Field Declarations

| | | |
|---|---|---|
| *FldDecl* | ::= | *FldMods*? *FldWTypes InitVal* |
| | \| | *FldMods*? *BindIdOrBindIdTuple* = *Expr* |
| | \| | *FldMods*? *BindIdOrBindIdTuple* : *Type* ... *InitVal* |
| | \| | *FldMods*? *BindIdOrBindIdTuple* : *TupleType InitVal* |
| *FldWTypes* | ::= | *FldWType* |
| | \| | ( *FldWType*( , *FldWType*)$^+$ ) |
| *FldWType* | ::= | *BindId IsType* |

## D.12 Abstract Field Declarations

| | | |
|---|---|---|
| *AbsFldDecl* | ::= | *AbsFldMods*? *AbsFldWTypes* |
| | \| | *AbsFldMods*? *BindIdOrBindIdTuple* : *Type* ... |
| | \| | *AbsFldMods*? *BindIdOrBindIdTuple* : *TupleType* |
| *AbsFldWTypes* | ::= | *AbsFldWType* |
| | \| | ( *AbsFldWType*( , *AbsFldWType*)$^+$ ) |
| *AbsFldWType* | ::= | *BindId IsType* |
| *ApiFldDecl* | ::= | *ApiFldMods*? *BindId IsType* |

## D.13 Expressions

```
Expr                  ::=  AssignExpr
                       |   OpExpr
                       |   DelimitedExpr
                       |   FlowExpr
                       |   fn ValParam IsType? Throws? ⇒ Expr
                       |   Expr as Type
                       |   Expr asif Type
AssignExpr            ::=  AssignLefts AssignOp Expr
AssignLefts           ::=  ( AssignLeft( , AssignLeft)* )
                       |   AssignLeft
AssignLeft            ::=  SubscriptExpr
                       |   FieldSelection
                       |   QualifiedName
SubscriptExpr         ::=  Primary LeftEncloser ExprList? RightEncloser
FieldSelection        ::=  Primary . Id
OpExpr                ::=  EncloserOp OpExpr? EncloserOp?
                       |   OpExpr EncloserOp OpExpr?
                       |   Primary
EncloserOp            ::=  Encloser
                       |   Op
Primary               ::=  ArrayExpr
                       |   MapExpr
                       |   Comprehension
                       |   LeftEncloser ExprList? RightEncloser
                       |   ParenthesisDelimited
                       |   VarOrFnRef
                       |   LiteralExpr
                       |   self
                       |   Primary LeftEncloser ExprList? RightEncloser
                       |   Primary . Id StaticArgs? ParenthesisDelimited
                       |   Primary . Id
                       |   Primary ^ Exponent
                       |   Primary ExponentOp
                       |   Primary ArgExpr
                       |   Primary Primary
VarOrFnRef            ::=  Id StaticArgs?
ParenthesisDelimited  ::=  Parenthesized
                       |   ArgExpr
                       |   ()
Exponent              ::=  Id
                       |   ParenthesisDelimited
                       |   LiteralExpr
                       |   self
FlowExpr              ::=  exit Id? (with Expr)?
                       |   Accumulator StaticArgs? ([ GeneratorClauseList ])? Expr
                       |   atomic AtomicBack
                       |   tryatomic AtomicBack
                       |   spawn Expr
                       |   throw Expr
```

| | | |
|---|---|---|
| *AtomicBack* | ::= | *AssignExpr* |
| | \| | *OpExpr* |
| | \| | *DelimitedExpr* |
| *GeneratorClauseList* | ::= | *GeneratorBinding*( , *GeneratorClause*)$^*$ |
| *GeneratorBinding* | ::= | *BindIdOrBindIdTuple* ← *Expr* |
| *GeneratorClause* | ::= | *GeneratorBinding* |
| | \| | *Expr* |

## D.14 Expressions Enclosed by Keywords or Symbols

| | | |
|---|---|---|
| *DelimitedExpr* | ::= | *ArgExpr* |
| | \| | *Parenthesized* |
| | \| | `object` *ExtendsWhere*? *GoInAnObject*? `end` |
| | \| | *Do* |
| | \| | `label` *Id BlockElems* `end` *Id* |
| | \| | `while` *Expr Do* |
| | \| | `for` *GeneratorClauseList DoFront* `end` |
| | \| | `if` *Expr* `then` *BlockElems Elifs*? *Else*? `end` |
| | \| | ( `if` *Expr* `then` *BlockElems Elifs*? *Else* `end` ?) |
| | \| | `case` *Expr Op*? `of` *CaseClauses CaseElse*? `end` |
| | \| | `case most` *Op* `of` *CaseClauses* `end` |
| | \| | `typecase` *TypecaseBindings* `of` *TypecaseClauses CaseElse*? `end` |
| | \| | `try` *BlockElems Catch*? ( `forbid` *TraitTypes*)? ( `finally` *BlockElems*)? `end` |
| *Do* | ::= | (*DoFront* `also` )$^*$ *DoFront* `end` |
| *DoFront* | ::= | ( `at` *Expr*)? `atomic` ? `do` *BlockElems*? |
| *ArgExpr* | ::= | ( (*Expr* , )$^*$ (*Expr* ... , )? *KeywordExpr* ( , *KeywordExpr*)$^*$ ) |
| | \| | *TupleExpr* |
| | \| | ( (*Expr* , )$^*$ *Expr* ... ) |
| *TupleExpr* | ::= | ( (*Expr* , )$^+$ *Expr* ) |
| *KeywordExpr* | ::= | *BindId* = *Expr* |
| *Parenthesized* | ::= | ( *Expr* ) |
| *Elifs* | ::= | *Elif*$^+$ |
| *Elif* | ::= | `elif` *Expr* `then` *BlockElems* |
| *Else* | ::= | `else` *BlockElems* |
| *CaseClauses* | ::= | *CaseClause*$^+$ |
| *CaseClause* | ::= | *Expr* ⇒ *BlockElems* |
| *CaseElse* | ::= | `else` ⇒ *BlockElems* |
| *TypecaseBindings* | ::= | *TypecaseVars* ( = *Expr*)? |
| *TypecaseVars* | ::= | *BindId* |
| | \| | ( *BindId*( , *BindId*)$^+$ ) |
| *TypecaseClauses* | ::= | *TypecaseClause*$^+$ |
| *TypecaseClause* | ::= | *TypecaseTypes* ⇒ *BlockElems* |
| *TypecaseTypes* | ::= | ( *TypeList* ) |
| | \| | *Type* |
| *Catch* | ::= | `catch` *BindId CatchClauses* |
| *CatchClauses* | ::= | *CatchClause*$^+$ |
| *CatchClause* | ::= | *TraitType* ⇒ *BlockElems* |

| | | |
|---|---|---|
| *MapExpr* | ::= | { *EntryList*? } |
| *Comprehension* | ::= | `BIG` ? [ *StaticArgs*? *ArrayComprehensionClause*$^+$ ] |
| | $\vert$ | `BIG` ? { *StaticArgs*? *Entry* $\mid$ *GeneratorClauseList* } |
| | $\vert$ | `BIG` ? *LeftEncloser StaticArgs*? *Expr* $\mid$ *GeneratorClauseList RightEncloser* |
| *Entry* | ::= | *Expr* $\mapsto$ *Expr* |
| *ArrayComprehensionClause* | ::= | *ArrayComprehensionLeft* $\mid$ *GeneratorClauseList* |
| *ArrayComprehensionLeft* | ::= | *IdOrInt* $\mapsto$ *Expr* |
| | $\vert$ | ( *IdOrInt* , *IdOrIntList* ) $\mapsto$ *Expr* |
| *IdOrInt* | ::= | *Id* |
| | $\vert$ | *IntLiteralExpr* |
| *IdOrIntList* | ::= | *IdOrInt*( , *IdOrInt*)$^*$ |
| *ExprList* | ::= | *Expr*( , *Expr*)$^*$ |
| *EntryList* | ::= | *Entry*( , *Entry*)$^*$ |

## D.15 Local Declarations

| | | |
|---|---|---|
| *BlockElems* | ::= | *BlockElem*$^+$ |
| *BlockElem* | ::= | *LocalVarFnDecl* |
| | $\vert$ | *Expr*( , *GeneratorClauseList*)? |
| *LocalVarFnDecl* | ::= | *LocalFnDecl*$^+$ |
| | $\vert$ | *LocalVarDecl* |
| *LocalFnDecl* | ::= | *LocalFnMods*? *NamedFnHeaderFront FnHeaderClause* = *Expr* |
| *LocalVarDecl* | ::= | `var` ? *LocalVarWTypes InitVal* |
| | $\vert$ | `var` ? *LocalVarWTypes* |
| | $\vert$ | `var` ? *LocalVarWoTypes* = *Expr* |
| | $\vert$ | `var` ? *LocalVarWoTypes* : *Type* ... *InitVal*? |
| | $\vert$ | `var` ? *LocalVarWoTypes* : *TupleType InitVal*? |
| *LocalVarWTypes* | ::= | *LocalVarWType* |
| | $\vert$ | ( *LocalVarWType*( , *LocalVarWType*)$^+$ ) |
| *LocalVarWType* | ::= | *BindId IsType* |

| | | |
|---|---|---|
| *LocalVarWoTypes* | ::= | *LocalVarWoType* |
| | $\vert$ | ( *LocalVarWoType*( , *LocalVarWoType*)$^+$ ) |
| *LocalVarWoType* | ::= | *BindId* |
| | $\vert$ | *Unpasting* |
| *Unpasting* | ::= | [ *UnpastingElems* ] |
| *UnpastingElems* | ::= | *UnpastingElem RectSeparator UnpastingElems* |
| | $\vert$ | *UnpastingElem* |
| *UnpastingElem* | ::= | *BindId* ( [ *UnpastingDim* ] )? |
| | $\vert$ | *Unpasting* |
| *UnpastingDim* | ::= | *ExtentRange* ($\times$ *ExtentRange*)$^+$ |

## D.16 Literals

| | | |
|---|---|---|
| *LiteralExpr* | ::= | ( ) |
| | \| | *NumericLiteralExpr* |
| | \| | *CharLiteralExpr* |
| | \| | *StringLiteralExpr* |
| *ArrayExpr* | ::= | [ *RectElements* ] |
| *RectElements* | ::= | *Expr MultiDimCons** |
| *MultiDimCons* | ::= | *RectSeparator Expr* |

## D.17 Types

| | | |
|---|---|---|
| *Type* | ::= | *TypeRef* |
| | \| | *TraitType* |
| | \| | *TupleType* |
| | \| | ( *Type* ) |
| | \| | () |
| | \| | *ArgType* → *Type Throws*? |
| | \| | *Type DimType* |
| *TypeRef* | ::= | *DottedId StaticArgs*? |
| *TraitType* | ::= | *TypeRef* |
| | \| | *Type* [ *ArraySize*? ] |
| | \| | *Type* ˆ *IntExpr* |
| | \| | *Type* ˆ ( *ExtentRange* ( × *ExtentRange*)* ) |
| *ArgType* | ::= | ( (*Type* , )* (*Type* ... , )? *KeywordType*( , *KeywordType*)* ) |
| | \| | ( (*Type* , )* *Type* ... ) |
| | \| | *TupleType* |
| *KeywordType* | ::- | *BindId* = *Type* |
| *TupleType* | ::= | ( *Type* , *TypeList* ) |
| *TypeList* | ::= | *Type*( , *Type*)* |
| *ArraySize* | ::= | *ExtentRange*( , *ExtentRange*)* |
| *ExtentRange* | ::= | *StaticArg*? # *StaticArg*? |
| | \| | *StaticArg*? : *StaticArg*? |
| | \| | *StaticArg* |
| *DimType* | ::= | *DottedId* |
| | \| | ( *DimType* ) |
| | \| | 1 |
| | \| | *DimPrefixOp DimType* |
| | \| | *DimType DimInfixOp DimType* |
| | \| | *DimType DimPostfixOp* |
| | \| | *DimType* `in` *Expr* |
| *DimPrefixOp* | ::= | `square` \| `cubic` \| `inverse` |
| *DimInfixOp* | ::= | · \| / \| `per` |
| *DimPostfixOp* | ::= | `squared` \| `cubed` |

## D.18 Symbols and Operators

| | | |
|---|---|---|
| *EncloserPair* | ::= | (*LeftEncloser* \| *Encloser*) ·? (*RightEncloser* \| *Encloser*) |
| *ExponentOp* | ::= | `^T` \| ^*Op* |
| *AssignOp* | ::= | := \| *Op*= |
| *Accumulator* | ::= | $\sum$ \| $\prod$ \| `BIG` *Op* |

## D.19 Identifiers

| | | |
|---|---|---|
| *BindId* | ::= | *Id* |
| | \| | _ |
| *BindIdList* | ::= | *BindId*( , *BindId*)* |
| *BindIdOrBindIdTuple* | ::= | *BindId* |
| | \| | ( *BindId* , *BindIdList* ) |
| *SimpleName* | ::= | *Id* |
| | \| | `opr` *Op* |
| | \| | `opr` *EncloserPair* |
| *APIName* | ::= | *Id*( . *Id*)* |
| *QualifiedName* | ::= | *Id*( . *Id*)* |

## D.20 Spaces and Comments

| | | |
|---|---|---|
| *RectSeparator* | ::= | ;+ |
| | \| | *Whitespace* |

# Appendix E

# Full Grammar for Fortress Implementors

In this chapter, we provide a complete definition of the Fortress concrete syntax. It includes unimplemented Fortress language features such as dimensions and units, tests and properties, coercions, and where clauses. This syntax has been extracted from the parser-generator source files of an open source partial Fortress reference implementation [8], available at:

`http://projectfortress.sun.com`

The parser of this reference implementation is generated by Rats! [7], which generates "packrat parsers" that memoize intermediate results to ensure linear time performance in the presence of unlimited lookahead and backtracking. Rats! includes production naming, module modifications, and semantic predicates. Actions that generate semantic values are omitted for simplicity. See [7] for more information about the Rats! parser generator.

## E.1 Components and APIs

```
File =
     w CompilationUnit w EndOfFile
   / (w Imports w ";"?)? w Exports w ";"? (w Decls w ";"?)? w EndOfFile
   / (w Imports w ";"?)? w AbsDecls w ";"? w EndOfFile
   / w Imports w ";"? (w AbsDecls w ";"?)? w EndOfFile

CompilationUnit =
     Component
   / Api

Component = ("native" w)? "component" w APIName (w Imports w ";"?)? w Exports w ";"? (w Decls w ";"?)? w "end"

Api = ("native" w)? "api" w APIName (w Imports w ";"?)? (w AbsDecls w ";"?)? w "end"

Imports = Import (br Import)*

Import =
     "import" w ImportedNames
   / "import" w "api" w AliasedAPINames

ImportedNames =
     APIName "." w "{" w "..." w "}" (w "except" w SimpleNames)?
   / APIName "." w "{" w AliasedSimpleNameList (w "," w "...")? w "}"
   / QualifiedName (w "as" w Id)?

SimpleNames =
     SimpleName
   / "{" w SimpleNameList w "}"

SimpleNameList = SimpleName (w "," w SimpleName)*

AliasedSimpleName =
     Id (w "as" w Id)?
   / "opr" w Op (w "as" w Op)?
   / "opr" w EncloserPair (w "as" w EncloserPair)?

AliasedSimpleNameList = AliasedSimpleName (w "," w AliasedSimpleName)*

AliasedAPINames =
     AliasedAPIName
   / "{" w AliasedAPINameList w "}"

AliasedAPIName = APIName (w "as" w Id)?

AliasedAPINameList = AliasedAPIName (w "," w AliasedAPIName)*

Exports = Export (br Export)*

Export = "export" w APINames

APINames =
     APIName
   / "{" w APINameList w "}"

APINameList = APIName (w "," w APIName)*
```

## E.2 Top-level Declarations

```
Decls = Decl (br Decl)*

Decl =
     TraitDecl
   / ObjectDecl
   / VarDecl
   / FnDecl
   / DimUnitDecl
   / TypeAlias
   / TestDecl
   / PropertyDecl
   / ExternalSyntax

AbsDecls = AbsDecl (br AbsDecl)*

AbsDecl =
     AbsTraitDecl
   / AbsObjectDecl
   / AbsVarDecl
   / AbsFnDecl
   / DimUnitDecl
   / TypeAlias
   / TestDecl
   / PropertyDecl
   / AbsExternalSyntax
```

## E.3 Trait and Object Declarations

```
TraitDecl = TraitMods? TraitHeaderFront TraitClauses (w GoInATrait)? w "end"
```

```
TraitHeaderFront = "trait" w Id (w StaticParams)? (w ExtendsWhere)?

/* Each trait clause cannot appear more than once. */
TraitClauses = (w TraitClause)*

TraitClause =
     Excludes
   / Comprises
   / Where

GoInATrait =
     (Coercions br)? GoFrontInATrait (br GoBackInATrait)?
   / (Coercions br)? GoBackInATrait
   / Coercions

Coercions = Coercion (br Coercion)*

GoFrontInATrait = GoesFrontInATrait (br GoesFrontInATrait)*

GoesFrontInATrait =
     AbsFldDecl
   / GetterSetterDecl
   / PropertyDecl

GoBackInATrait = GoesBackInATrait (br GoesBackInATrait)*

GoesBackInATrait =
      MdDecl
    / PropertyDecl

ObjectDecl = ObjectMods? ObjectHeader (w GoInAnObject)? w "end"

ObjectHeader = "object" w Id (w StaticParams)? (w ObjectValParam)? (w ExtendsWhere)? FnClauses

ObjectValParam = "(" (w Params)? w ")"

Varargs := "transient" w VarargsParam

Param :=
     ParamFldMods? PlainParam
   / "transient" w PlainParam

GoInAnObject =
     (Coercions br)? GoFrontInAnObject (br GoBackInAnObject)?
   / (Coercions br)? GoBackInAnObject
   / Coercions

GoFrontInAnObject = GoesFrontInAnObject (br GoesFrontInAnObject)*

GoesFrontInAnObject =
     FldDecl
   / GetterSetterDef
   / PropertyDecl

GoBackInAnObject = GoesBackInAnObject (br GoesBackInAnObject)*

GoesBackInAnObject =
      MdDef
    / PropertyDecl

AbsTraitDecl = AbsTraitMods? TraitHeaderFront AbsTraitClauses (w AbsGoInATrait)? w "end"

/* Each trait clause cannot appear more than once. */
AbsTraitClauses = (w AbsTraitClause)*
```

```
AbsTraitClause =
     Excludes
   / AbsComprises
   / Where

AbsGoInATrait =
     (AbsCoercions br)? AbsGoFrontInATrait (br AbsGoBackInATrait)?
   / (AbsCoercions br)? AbsGoBackInATrait
   / AbsCoercions

AbsCoercions = AbsCoercion (br AbsCoercion)*

AbsGoFrontInATrait = AbsGoesFrontInATrait (br AbsGoesFrontInATrait)*

AbsGoesFrontInATrait =
     ApiFldDecl
   / AbsGetterSetterDecl
   / PropertyDecl

AbsGoBackInATrait = AbsGoesBackInATrait (br AbsGoesBackInATrait)*

AbsGoesBackInATrait =
      AbsMdDecl
    / PropertyDecl

AbsObjectDecl = AbsObjectMods? ObjectHeader (w AbsGoInAnObject)? w "end"

AbsGoInAnObject =
     (AbsCoercions br)? AbsGoFrontInAnObject (br AbsGoBackInAnObject)?
   / (AbsCoercions br)? AbsGoBackInAnObject
   / AbsCoercions

AbsGoFrontInAnObject = AbsGoesFrontInAnObject (br AbsGoesFrontInAnObject)*

AbsGoesFrontInAnObject =
     ApiFldDecl
   / AbsGetterSetterDecl
   / PropertyDecl

AbsGoBackInAnObject = AbsGoesBackInAnObject (br AbsGoesBackInAnObject)*

AbsGoesBackInAnObject =
      AbsMdDecl
    / PropertyDecl
```

## E.4 Variable Declarations

```
VarDecl =
     VarMods? NoNewlineVarWTypes w InitVal
   / VarMods? BindIdOrBindIdTuple w "=" w NoNewlineExpr
   / VarMods? BindIdOrBindIdTuple w ":" w Type w "..." w InitVal
   / VarMods? BindIdOrBindIdTuple w ":" w TupleType w InitVal

/* Each modifier cannot appear more than once. */
VarMods = (VarMod w)+

VarMod =
     AbsVarMod
   / "private"

VarWTypes =
     VarWType
   / "(" w VarWType (w "," w VarWType)+ w ")"

VarWType = BindId w IsType

InitVal = ("=" / ":=") w NoNewlineExpr

AbsVarDecl =
     AbsVarMods? VarWTypes
   / AbsVarMods? BindIdOrBindIdTuple w ":" w Type w "..."
   / AbsVarMods? BindIdOrBindIdTuple w ":" w TupleType

/* Each modifier cannot appear more than once. */
AbsVarMods = (AbsVarMod w)+

AbsVarMod =
     "var"
   / "test"
```

## E.5 Function Declarations

```
FnDecl =
     FnMods? FnHeaderFront FnHeaderClause (w "=" w NoNewlineExpr)?
   / FnSig

FnSig = SimpleName w ":" w NoNewlineType

AbsFnDecl =
     AbsFnMods? FnHeaderFront FnHeaderClause
   / FnSig

FnHeaderFront =
     NamedFnHeaderFront
   / OpHeaderFront

NamedFnHeaderFront = Id (w StaticParams)? w ValParam
```

## E.6 Dimension, Unit, Type Alias, Test, Property, and External Syntax Declarations

```
DimUnitDecl =
   / "dim" w Id (w "=" w NoNewlineType)? s ("unit" / "SI_unit") w Id (wr Id)*
       (w "=" w NoNewlineExpr)?
   / "dim" w Id (w "=" w NoNewlineType)? (w "default" w Id)?
   / ("unit" / "SI_unit") w Id (wr Id)* (w ":" w NoNewlineType)? (w "=" w NoNewlineExpr)?

TypeAlias = "type" w Id (w StaticParams)? w "=" w NoNewlineType

TestDecl = "test" w Id w "[" w GeneratorClauseList w "]" w "=" w NoNewlineExpr

PropertyDecl = "property" (w Id w "=")? (w "FORALL" w ValParam)? w NoNewlineExpr

ExternalSyntax = "syntax" w OpenExpander w Id w CloseExpander w "=" w NoNewlineExpr

OpenExpander =
     Id
   / LeftEncloser
   / Encloser

CloseExpander =
     Id
   / RightEncloser
   / Encloser
   / "end"

AbsExternalSyntax = "syntax" w OpenExpander w Id w CloseExpander
```

## E.7 Headers without Newlines

```
ExtendsWhere = "extends" w TraitTypeWheres

TraitTypeWheres =
     TraitTypeWhere
   / "{" w TraitTypeWhereList w "}"

TraitTypeWhereList = TraitTypeWhere (w "," w TraitTypeWhere)*

TraitTypeWhere = TraitType (w Where)?

Extends = "extends" w TraitTypes

Excludes = "excludes" w TraitTypes

Comprises = "comprises" w TraitTypes

AbsComprises = "comprises" w ComprisingTypes

TraitTypes =
     TraitType
   / "{" w TraitTypeList w "}"

TraitTypeList = TraitType (w "," w TraitType)*

ComprisingTypes =
     TraitType
   / "{" w ComprisingTypeList w "}"

ComprisingTypeList =
     "..."
   / TraitType (w "," w TraitType)* (w "," w "...")?

Where =
     "where" w "[\" w WhereBindingList w "\]" (w "{" w WhereConstraintList w "}")?
   / "where" w "{" w WhereConstraintList w "}"

WhereBindingList = WhereBinding (w "," w WhereBinding)*

WhereBinding =
     "nat" w Id
   / "int" w Id
   / "bool" w Id
   / "unit" w Id
   / Id (w Extends)?
   / "[\"       // Error production

FnHeaderClause = (w NoNewlineIsType)? FnClauses

FnClauses = (w Throws)? (w Where)? Contract

Throws = "throws" w MayTraitTypes

MayTraitTypes =
     "{" w "}"
   / TraitTypes

/* Each modifier cannot appear more than once. */
TraitMods = (TraitMod w)+

TraitMod =
     AbsTraitMod
   / "private"

/* Each modifier cannot appear more than once. */
AbsTraitMods = (AbsTraitMod w)+

AbsTraitMod =
     "value"
   / "test"

/* Each modifier cannot appear more than once. */
ObjectMods = TraitMods

/* Each modifier cannot appear more than once. */
AbsObjectMods = AbsTraitMods

/* Each modifier cannot appear more than once. */
MdMods = (MdMod w)+

MdMod =
     FnMod
   / "override"

/* Each modifier cannot appear more than once. */
AbsMdMods = (AbsMdMod w)+

AbsMdMod =
     AbsFnMod
   / "override"

/* Each modifier cannot appear more than once. */
FnMods = (FnMod w)+

FnMod =
     AbsFnMod
   / "private"

/* Each modifier cannot appear more than once. */
AbsFnMods = (AbsFnMod w)+

AbsFnMod =
     LocalFnMod
   / "test"

/* Each modifier cannot appear more than once. */
LocalFnMods = (LocalFnMod w)+

LocalFnMod =
     "atomic"
   / "io"

/* Each modifier cannot appear more than once. */
ParamFldMods = (ParamFldMod w)+

ParamFldMod =
     "var"
   / "hidden"
   / "settable"
   / "wrapped"
```

```
/* Each modifier cannot appear more than once. */
FldMods = (FldMod w)+

FldMod =
     "var"
   / AbsFldMod

/* Each modifier cannot appear more than once. */
AbsFldMods = (AbsFldMod w)+

AbsFldMod =
     ApiFldMod
   / "wrapped"
   / "private"

/* Each modifier cannot appear more than once. */
ApiFldMods = (ApiFldMod w)+

ApiFldMod =
     "hidden"
   / "settable"
   / "test"

StaticParams = "[\" w StaticParamList w "\]"

StaticParamList = StaticParam (w "," w StaticParam)*

StaticParam =
     "nat"  w Id
   / "int"  w Id
   / "bool" w Id
   / "dim"  w Id
   / "unit" w Id (w ":" w NoNewlineType)? (w "absorbs" w "unit")?
   / "opr"  w Op
   / Id (w Extends)? (w "absorbs" w "unit")?
   / "[\"       // Error production
```

## E.8 Headers Maybe with Newlines

```
IsType = ":" w Type

WhereConstraintList = WhereConstraint (w "," w WhereConstraint)*

WhereConstraint =
     Id w Extends
   / TypeAlias
   / Type w "coerces" w Type
   / Type w "widens" w Type
   / UnitConstraint
   / QualifiedName w "=" w QualifiedName
   / IntConstraint
   / BoolConstraint

UnitConstraint =
     "dimensionless" w "=" w Id
   / Id w "=" w "dimensionless"

IntConstraint =
     IntExpr w lessthanequal w IntExpr
   / IntExpr w lessthan w IntExpr
   / IntExpr w greaterthanequal w IntExpr
   / IntExpr w greaterthan w IntExpr
   / IntExpr w equals w IntExpr

IntVal =
     IntLiteralExpr
   / QualifiedName

IntExpr = IntExprFront IntExprTail*

IntExprFront =
     IntVal
   / "(" w IntExpr w ")"

IntExprTail =
     SumIntExpr
   / MinusIntExpr
   / ProductIntExpr
   / ExponentIntExpr

SumIntExpr = w "+" w IntExpr

MinusIntExpr = w "-" w IntExpr

ProductIntExpr = (w "DOT" w / sr) IntExpr

ExponentIntExpr = "^" IntVal

BoolConstraint = BoolConstraintFront BoolConstraintTail*

BoolConstraintFront =
     NOT w BoolExpr
   / BoolConstraintHead w OR w BoolExpr
   / BoolConstraintHead w AND w BoolExpr
   / BoolConstraintHead w IMPLIES w BoolExpr
   / BoolConstraintHead w "=" w BoolExpr

BoolConstraintHead =
     BoolVal
   / "(" w BoolExpr w ")"

BoolConstraintTail =
     OrBoolConstraint
   / AndBoolConstraint
   / ImpliesBoolConstraint
   / EqualsBoolConstraint

OrBoolConstraint = w OR w BoolExpr

AndBoolConstraint = w AND w BoolExpr

ImpliesBoolConstraint = w IMPLIES w BoolExpr

EqualsBoolConstraint = w "=" w BoolExpr

BoolVal =
     "true"
   / "false"
   / QualifiedName

BoolExpr = BoolExprFront BoolExprTail*

BoolExprFront =
     BoolVal
   / "(" w BoolExpr w ")"
   / NOT w BoolExpr
```

```
BoolExprTail =
     OrBoolExpr
   / AndBoolExpr
   / ImpliesBoolExpr
   / EqualsBoolExpr

OrBoolExpr = w OR w BoolExpr

AndBoolExpr = w AND w BoolExpr

ImpliesBoolExpr = w IMPLIES w BoolExpr

EqualsBoolExpr = w "=" w BoolExpr

Contract = (w Requires)? (w Ensures)? (w Invariant)?

Requires = "requires" w "{" (w ExprList)? w "}"

Ensures = "ensures" w "{" (w EnsuresClauseList)? w "}"

EnsuresClauseList = EnsuresClause (w "," w EnsuresClause)*

EnsuresClause = Expr (w "provided" w Expr)?

Invariant = "invariant" w "{" (w ExprList)? w "}"

StaticArgs = "[\" w StaticArgList w "\]"

StaticArgList = StaticArg (w "," w StaticArg)*

StaticArg =
     Op
   / "Unity"
   / "dimensionless"
   / "1" w "/" w Type
   / DimPrefixOp sr Type
   / "(" w StaticArg w ")"
   / !(QualifiedName (w "\]" / w "]" / w "," / w "[\" / w "[" / w "->" /
                      w "OR" / w "AND" / w "IMPLIES" / w "="))
     IntExpr
   / !(QualifiedName (w "\]" / w "]" / w "," / w "[\" / w "[" / w "->"))
     BoolExpr
   / !(QualifiedName (w "DOT" / w "/" / w "per" / w DimPostfixOp))
     Type
   / Expr
```

## E.9 Parameters

```
ValParam =
     BindId
   / "(" (w Params)? w ")"

Params =
     (Param w "," w)* (Varargs w "," w)? Keyword (w "," w Keyword)*
   / (Param w "," w)*  Varargs
   / Param (w "," w Param)*

VarargsParam = BindId w ":" w Type w "..."

Varargs = VarargsParam

Keyword = Param w "=" w Expr
```

```
PlainParam =
     BindId w IsType
   / !(BindId (w "\]" / w "]" / w "[\" / w "[" / w "->")) BindId
   / Type

Param = PlainParam

OpHeaderFront =
     "opr" (w "BIG")? w ("{" w "|->" / LeftEncloser / Encloser) (w StaticParams)? w Params w
     (RightEncloser / Encloser)
   / "opr" w ValParam w (Op / ExponentOp) (w StaticParams)?
   / "opr" (w "BIG")? w (Op / "^" / Encloser / SUM / PROD) (w StaticParams)? w ValParam
```

## E.10 Method Declarations

```
MdDecl =
     MdDef
   / ("abstract" w)? MdMods? MdHeaderFront FnHeaderClause

MdDef = MdMods? MdHeaderFront FnHeaderClause w "=" w NoNewlineExpr

AbsMdDecl = ("abstract" w)? AbsMdMods? MdHeaderFront FnHeaderClause

MdHeaderFront =
     NamedMdHeaderFront
   / OpHeaderFront

NamedMdHeaderFront = Id (w StaticParams)? w ValParam

GetterSetterDecl =
     GetterSetterDef
   / (abstract w)? FnMods? GetterSetterMod MdHeaderFront FnHeaderClause

GetterSetterDef =
     FnMods? GetterSetterMod MdHeaderFront FnHeaderClause w "=" w NoNewlineExpr

GetterSetterMod =
     "getter" w
   / "setter" w

AbsGetterSetterDecl = (abstract w)? AbsFnMods? GetterSetterMod MdHeaderFront FnHeaderClause

Coercion = "coerce" (w StaticParams)? w "(" w BindId w IsType w ")" CoercionClauses w
           ("widens" w)? "=" w NoNewlineExpr

AbsCoercion = "coerce" (w StaticParams)? w "(" w BindId w IsType w ")" CoercionClauses (w "widens")?

CoercionClauses = (w CoercionWhere)? (w Ensures)? (w Invariant)?

CoercionWhere =
     "where" w "[\" w WhereBindingList w "\]" (w "{" w CoercionWhereConstraintList w "}")?
   / "where" w "{" w CoercionWhereConstraintList w "}"

CoercionWhereConstraintList = CoercionWhereConstraint (w "," w CoercionWhereConstraint)*

CoercionWhereConstraint =
     WhereConstraint
   / Type w "widens" w "or" w "coerces" w Type
```

## E.11 Method Parameters

```
ValParam := "(" (w Params)? w ")"

Param :=
     PlainParam
   / "self"

OpHeaderFront :=
     "opr" (w "BIG")? w ("{" w "|->" / LeftEncloser / Encloser) (w StaticParams)? w Params w
       (RightEncloser / Encloser) (w ":=" w "(" w SubscriptAssignParam w ")")?
   / ...

SubscriptAssignParam =
     Varargs
   / Param
```

## E.12 Field Declarations

```
FldDecl = VarDecl

VarMods := FldMods
```

## E.13 Abstract Field Declarations

```
AbsFldDecl = AbsVarDecl

VarMods := AbsFldMods

ApiFldDecl = ApiFldMods? BindId w NoNewlineIsType
```

## E.14 Expressions

```
Expr = ExprFront ExprTail*

ExprFront =
     AssignExpr
   / OpExpr
   / DelimitedExpr
   / <Flow> FlowExpr
   / <Fn> "fn" w ValParam (w IsType)? (w Throws)? w "=>" w Expr

ExprTail =
     <As> As
   / <Asif> AsIf

As = w "as" w Type

AsIf = w "asif" w Type

AssignExpr = AssignLefts w AssignOp w Expr

AssignLefts =
     "(" w AssignLeft (w "," w AssignLeft)* w ")"
   / AssignLeft

AssignLeft =
     PrimaryFront AssignLeftTail+
   / QualifiedName

AssignLeftTail =
     SubscriptAssign
   / FieldSelectionAssign

SubscriptAssign = LeftEncloser (w ExprList)? w RightEncloser

FieldSelectionAssign = "." Id

OpExpr =
     OpExprNoEnc
   / OpExprLeftEncloser
   / Encloser

OpExprNoEnc =
     OpExprPrimary
   / OpExprPrefix
   / Op

TightInfixRight =
     Encloser OpExprPrimary
   / Encloser OpExprPrefix
   / <Primary>   Encloser wr OpExprPrimary
   / <Loose>     Encloser wr LooseInfix
   / <LeftLoose> Encloser wr LeftLooseInfix
   / Encloser

LeftLooseInfix =
     OpExprLeftEncloser
   / <Primary> Encloser wr OpExprPrimary
   / <Prefix>  Encloser wr OpExprPrefix
   / <Left>    Encloser wr OpExprLeftEncloser

OpExprLeftEncloser = Encloser OpExprNoEnc

OpExprPrimary =
     Primary TightInfixPostfix
   / Primary TightInfixRight
   / <Primary>   Primary wr OpExprPrimary
   / <Loose>     Primary wr LooseInfix
   / <LeftLoose> Primary wr LeftLooseInfix
   / Primary

OpExprPrefix =
     Op OpExprPrimary
   / Op OpExprPrefix
   / Op OpExprLeftEncloser
   / <Primary> Op wr OpExprPrimary
   / <Prefix>  Op wr OpExprPrefix
   / <Left>    Op wr OpExprLeftEncloser

TightInfixPostfix =
     Op OpExprPrimary
   / Op OpExprPrefix
   / Op OpExprLeftEncloser
   / <Primary> Op wr OpExprPrimary
   / <Prefix>  Op wr OpExprPrefix
   / <Left>    Op wr OpExprLeftEncloser
   / Op

LooseInfix =
     Op wr OpExprPrimary
   / Op wr OpExprPrefix
   / <Left> Op wr OpExprLeftEncloser

Primary = LeftAssociatedPrimary / MathPrimary

LeftAssociatedPrimary =
     DottedIdChain StaticArgs ParenthesisDelimited ParenthesisDelimitedLeft* Selector*
   / DottedIdChain SubscriptingLeft+ ParenthesisDelimitedLeft* Selector*
   / DottedIdChain ParenthesisDelimited ParenthesisDelimitedLeft+ Selector*
   / DottedIdChain ParenthesisDelimitedLeft* Selector*
   / PrimaryFront SubscriptingLeft* ParenthesisDelimitedLeft* Selector+

DottedIdChain = Id ("." w Id)+

MathPrimary = PrimaryFront MathItem*

PrimaryFront =
     ArrayExpr
   / MapExpr
   / Comprehension
   / LeftEncloser (w ExprList)? w RightEncloser
   / ParenthesisDelimited
   / VarOrFnRef
   / LiteralExpr
   / "self"

/* ArrayExpr is defined in Literal.rats */

VarOrFnRef = Id StaticArgs?

SubscriptingLeft =
     ("[" / "{") (w ExprList)? w ("]" / "}")
   / LeftEncloser (w ExprList)? w RightEncloser

ParenthesisDelimitedLeft = ParenthesisDelimited

ParenthesisDelimited =
     Parenthesized
   / ArgExpr
   / "(" w ")"

Selector =
     MethodInvocationSelector
   / FieldSelectionSelector

MethodInvocationSelector =
     "." w Id StaticArgs? ParenthesisDelimited ParenthesisDelimitedLeft*

FieldSelectionSelector = "." w Id SubscriptingLeft* ParenthesisDelimitedLeft*

MathItem =
     Subscripting
   / Exponentiation
   / ParenthesisDelimited
   / VarOrFnRef
   / LiteralExpr
   / "self"

Subscripting =
     ("[" / "{") (w ExprList)? w ("]" / "}")
   / LeftEncloser (w ExprList)? w RightEncloser

Exponentiation =
     "^" Exponent
   / ExponentOp
```

```
Exponent =
     Id
   / ParenthesisDelimited
   / LiteralExpr
   / "self"

FlowExpr =
     "exit" (w Id)? (w "with" w Expr)?
   / Accumulator StaticArgs? (w "[" w GeneratorClauseList w "]")? w Expr
   / "atomic" w AtomicBack
   / "tryatomic" w AtomicBack
   / "spawn" w Expr
   / "throw" w Expr

AtomicBack =
     AssignExpr
   / OpExpr
   / DelimitedExpr

GeneratorClauseList = GeneratorBinding (w "," w GeneratorClause)*

GeneratorBinding = BindIdOrBindIdTuple w "<-" w Expr

GeneratorClause =
     GeneratorBinding
   / Expr
```

## E.15 Expressions Enclosed by Keywords or Symbols

```
DelimitedExpr =
     ArgExpr
   / Parenthesized
   / "object" (w ExtendsWhere)? (w GoInAnObject)? w "end"
   / Do
   / "label" w Id w BlockElems w "end" w Id
   / "while" w Expr w Do
   / "for" w GeneratorClauseList w DoFront w "end"
   / "if" w Expr w "then" w BlockElems (w Elifs)? (w Else)? w "end"
   / "(" w "if" w Expr w "then" w BlockElems (w Elifs)? w Else (w "end")? w ")"
   / "case" w Expr (w Op)? w "of" w CaseClauses (w CaseElse)? w "end"
   / "case" w "most" w Op w "of" w CaseClauses w "end"
   / "typecase" w TypecaseBindings w "of" w TypecaseClauses (br CaseElse)? w "end"
   / "try" w BlockElems (w Catch)? (w "forbid" w TraitTypes)? (w "finally" w BlockElems)? w "end"

Do = (DoFront w "also" w)* DoFront w "end"

DoFront = ("at" w Expr w)? ("atomic" w)? "do" (w BlockElems)?

ArgExpr =
     "(" w (Expr w "," w)* (Expr w "..." w "," w)? KeywordExpr (w "," w KeywordExpr)* w ")"
   / "(" w (Expr w "," w)* Expr w "..." w ")"
   / TupleExpr

TupleExpr = "(" w (Expr w "," w)* Expr w ")"

KeywordExpr = BindId w "=" w Expr

Parenthesized = "(" w Expr w ")"

Elifs = Elif (w Elif)*

Elif = "elif" w Expr w "then" w BlockElems
```

```
Else = "else" w BlockElems

CaseClauses = CaseClause (br CaseClause)*

/* CaseClause is defined in LocalDecl.rats */

CaseElse = "else" w match w BlockElems

TypecaseBindings = TypecaseVars (w "=" w Expr)?

TypecaseVars =
     BindId
   / "(" w BindId (w "," w BindId)+ w ")"

TypecaseClauses = TypecaseClause (br TypecaseClause)*

TypecaseClause = TypecaseTypes w match w BlockElems

TypecaseTypes =
     "(" w TypeList w ")"
   / Type

Catch = "catch" w BindId w CatchClauses

CatchClauses = CatchClause (br CatchClause)*

CatchClause = TraitType w match w BlockElems

MapExpr = "{" (w EntrList)? w "}"

Comprehension =
     ("BIG" w)? "[" StaticArgs? w ArrayComprehensionClause (br ArrayComprehensionClause)* w "]"
   / ("BIG" w)? "{" StaticArgs? w Entry wr bar wr GeneratorClauseList w "}"
   / ("BIG" w)? LeftEncloser StaticArgs? w Expr wr bar wr GeneratorClauseList w RightEncloser

/* The operator "|->" should not be in the left-hand sides of map expressions
   and map/array comprehensions.
 */
mapstoOp = !("|->" w Expr (w mapsto / wr bar / w "}" / w ",")) "|->"

Entry = Expr w mapsto w Expr

ArrayComprehensionLeft =
     IdOrInt w mapsto w Expr
   / "(" w IdOrInt w "," w IdOrIntList w ")" w mapsto w Expr

/* ArrayComprehensionClause is defined in Symbol.rats */

IdOrInt =
     Id
   / IntLiteralExpr

IdOrIntList = IdOrInt (w "," w IdOrInt)*

ExprList = Expr (w "," w Expr)*

EntryList = Entry (w "," w Entry)*
```

## E.16 Local Declarations

```
BlockElems =
     BlockElem br BlockElems
   / BlockElem
```

```
     &(w Elifs / w Else / br CaseClause / br TypecaseTypes / br CaseElse
     / w "also" / w "end" / w Catch / w "forbid" / w "finally" / w ")")
   / BlockElem w ";"
     &(w Elifs / w Else / w CaseClause / w TypecaseTypes / w CaseElse
     / w "also" / w "end" / w Catch / w "forbid" / w "finally" / w ")")

BlockElem =
     LocalVarFnDecl
   / NoNewlineExpr (s "," w NoNewlineGeneratorClauseList)?

LocalVarFnDecl =
     LocalFnDecl (br LocalFnDecl)*
   / LocalVarDecl

LocalFnDecl = LocalFnMods? NamedFnHeaderFront FnHeaderClause w "=" w NoNewlineExpr

LocalVarDecl =
     ("var" w)? NoNewlineVarWTypes s InitVal
   / ("var" w)? NoNewlineVarWTypes
   / ("var" w)? VarWoTypes s "=" s NoNewlineExpr
   / ("var" w)? VarWoTypes s ":" s Type s "..." (s InitVal)?
   / ("var" w)? VarWoTypes s ":" s TupleType (s InitVal)?

VarWType := BindId s ":" s Type

VarWoTypes =
     VarWoType
   / "(" w VarWoType (w "," w VarWoType)+ w ")"

VarWoType =
     BindId
   / Unpasting

Unpasting = "[" w UnpastingElems w "]"

UnpastingElems =
     UnpastingElem RectSeparator UnpastingElems
   / UnpastingElem

UnpastingElem =
     BindId ("[" w UnpastingDim w "]")?
   / Unpasting

UnpastingDim = ExtentRange (w "BY" w ExtentRange)+

CaseClause = NoNewlineExpr w match w BlockElems
```

## E.17 Literals

```
LiteralExpr =
     "(" w ")"
   / NumericLiteralExpr
   / CharLiteralExpr
   / StringLiteralExpr

ArrayExpr = "[" w RectElements w "]"

RectElements = NoSpaceExpr MultiDimCons*

MultiDimCons = RectSeparator NoSpaceExpr

/* RectSeparator is defined in Spacing.rats */
```

```
NumericLiteralExpr =
     FloatLiteralExpr
   / IntLiteralExpr

FloatLiteralExpr = DigitString "." DigitString

IntLiteralExpr = DigitString

DigitString = [0-9]+

CharLiteralExpr = "’" CharLiteralContent "’"

StringLiteralExpr = ["] StringLiteralContent* ["]
StringLiteralContent =
     EscapeSequence
   / !["\\] _
EscapeSequence = ’\\’ [btnfr"\\]

CharLiteralContent   = ’\\’ [btnfr"\\] / !"’" _
```

## E.18 Expressions without Newlines

```
NoNewlineExpr = Expr
Expr := ...
ExprFront := ...
NoNewlineAssignExpr = AssignLefts s AssignOp w NoNewlineExpr

ExprTail :=
     <As> NoNewlineAs
   / <AsIf> NoNewlineAsIf

NoNewlineAs = s "as" w NoNewlineType

NoNewlineAsIf = s "asif" w NoNewlineType

TightInfixRight := ...
   / <Primary>   Encloser sr OpExprPrimary
   / <Loose>     Encloser sr LooseInfix
   / <LeftLoose> Encloser sr LeftLooseInfix

LeftLooseInfix := ...
   / <Primary> Encloser sr OpExprPrimary
   / <Prefix>  Encloser sr OpExprPrefix
   / <Left>    Encloser sr OpExprLeftEncloser

OpExprPrimary := ...
   / <Primary>   Primary sr OpExprPrimary
   / <Loose>     Primary sr LooseInfix
   / <LeftLoose> Primary sr LeftLooseInfix

OpExprPrefix := ...
   / <Primary> Op sr OpExprPrimary
   / <Prefix>  Op sr OpExprPrefix
   / <Left>    Op sr OpExprLeftEncloser

TightInfixPostfix := ...
   / <Primary> Op sr OpExprPrimary
   / <Prefix>  Op sr OpExprPrefix
   / <Left>    Op sr OpExprLeftEncloser

LooseInfix := ...
   / <Left> Op sr OpExprLeftEncloser
```

```
GeneratorClauseList := GeneratorBinding (s "," w GeneratorClause)*

NoNewlineGeneratorClauseList = GeneratorClauseList

NoNewlineVarWTypes =
     NoNewlineVarWType
   / "(" w NoNewlineVarWType (w "," w NoNewlineVarWType)+ w ")"

NoNewlineVarWType = BindId w NoNewlineIsType

NoNewlineIsType = ":" w NoNewlineType
```

## E.19 Expressions within Array Expressions

```
ExprFront -= <Flow> , <Fn>
NoSpaceExpr = ExprFront

TightInfixRight -= <Primary>, <Loose>, <LeftLoose>

OpExprPrimary -= <Primary>, <Loose>, <LeftLoose>

OpExprPrefix -= <Primary>, <Prefix>, <Left>

TightInfixPostfix -= <Primary>, <Prefix>, <Left>
```

## E.20 Types

```
Type = !("1") TypePrimary (w "in" w Expr)?

OpType =
     TypePrimary
   / TypePrefix

TypePrimary =
     TypePrimaryFront TightInfixPostfix
   / <LooseJuxt> TypePrimaryFront wr TypePrimary
   / <LooseInfix> TypePrimaryFront wr LooseInfix
   / TypePrimaryFront

TypePrefix =
     DimPrefixOp TypePrimary
   / DimPrefixOp TypePrefix
   / <Prefix> DimPrefixOp wr TypePrimary
   / <PrePrefix> DimPrefixOp wr TypePrefix

TightInfixPostfix =
     <Arrow> TypeInfixOp TypePrimary (w Throws)?
   / <ArrowPrefix> TypeInfixOp TypePrefix (w Throws)?
   / DimInfixOp TypePrimary
   / DimInfixOp TypePrefix
   / <Postfix> DimPostfixOp wr TypePrimary
   / <PostPrefix> DimPostfixOp wr TypePrefix
   / DimPostfixOp

LooseInfix =
     <Arrow> TypeInfixOp wr TypePrimary (w Throws)?
   / <ArrowPrefix> TypeInfixOp wr TypePrefix (w Throws)?
   / <Infix> DimInfixOp wr TypePrimary
   / <InPrefix> DimInfixOp wr TypePrefix
```

```
TypePrimaryFront = TypeFront TypeTail*

TypeFront =
     ParenthesizedType
   / ArgType
   / TypeRef
   / VoidType
   / "1"

ParenthesizedType = "(" w Type w ")"

ArgType =
     "(" w (Type w "," w)* (Type w "..." w "," w)? KeywordType (w "," w KeywordType)* w ")"
   / "(" w (Type w "," w)* Type w "..." w ")"
   / TupleType

KeywordType = BindId w "=" w Type

TupleType = "(" w Type w "," w TypeList w ")"

TypeList = Type (w "," w Type)*

TypeRef = Id StaticArgs?

VoidType = "(" w ")"

TypeTail =
     ArrayTypeSize
   / Exponentiation
   / ParenthesizedTypeLeft
   / IdLeft
   ;

ArrayTypeSize = "[" (w ArraySize)? w "]"

ArraySize = ExtentRange (w "," w ExtentRange)*

ExtentRange =
     (StaticArg w)? "#" (w StaticArg)?
   / (StaticArg w)? ":" (w StaticArg)?
   / StaticArg

Exponentiation =
     "^" IntExpr
   / "^" "(" w ExtentRange (w "BY" w ExtentRange)* w ")"

ParenthesizedTypeLeft = "(" w OpType w ")"

IdLeft = Id

TypeInfixOp = "->"

DimInfixOp = "DOT" / "/" / "per"

DimPrefixOp = "square" / "cubic" / "inverse"

DimPostfixOp = "squared" / "cubed"

TraitType =
     TraitTypeFront TraitTypeTail+
   / TypeRef

TraitTypeFront =
     ParenthesizedType
   / TupleType
   / TypeRef
   / VoidType

TraitTypeTail =
     ArrayTypeSizeTrait
   / ExponentiationTrait

ArrayTypeSizeTrait = "[" (w ArraySize)? w "]"

ExponentiationTrait =
     "^" IntExpr
   / "^" "(" w ExtentRange (w "BY" w ExtentRange)* w ")"
```

## E.21 Types without Newlines

```
Type Type := ...

NoNewlineType = !("1") TypePrimary (w "in" w NoNewlineExpr)?

TypePrimary := ...
   / <LooseJuxt> TypePrimaryFront sr TypePrimary
   / <LooseInfix> TypePrimaryFront sr LooseInfix

TypePrefix := ...
   / <Prefix> DimPrefixOp sr TypePrimary
   / <PrePrefix> DimPrefixOp sr TypePrefix

TightInfixPostfix := ...
   / <Arrow> TypeInfixOp TypePrimary (s Throws)?
   / <ArrowPrefix> TypeInfixOp TypePrefix (s Throws)?
   / <Postfix> DimPostfixOp sr TypePrimary
   / <PostPrefix> DimPostfixOp sr TypePrefix

LooseInfix := ...
   / <Arrow> TypeInfixOp sr TypePrimary (s Throws)?
   / <ArrowPrefix> TypeInfixOp sr TypePrefix (s Throws)?
   / <Infix> DimInfixOp sr TypePrimary
   / <InPrefix> DimInfixOp sr TypePrefix
```

## E.22 Symbols and Operators

```
Encloser = encloser

LeftEncloser = leftEncloser

RightEncloser = rightEncloser

ExponentOp = exponentOp

EncloserPair = (LeftEncloser / Encloser) (w DOT)? w (RightEncloser / Encloser)

bar = &("|" wr GeneratorClauseList closingComprehension) "|"
closingComprehension =
     w "}"
   / w "|>"
   / br ArrayComprehensionClause
   / w "]"
sd = [*.]?
bars = "|" (sd "|")*
slashes = "/" (sd "/")*
```

```
        / "\\" (sd "\\")*
lesses = "<" (sd "<")*
greaters = ">" (sd ">")*

encloser = !(bar) bars !([*.>/\\] / "->")

leftEncloser =
     leftEncloserMulti
   / !('|') _

leftEncloserMulti =
     "(" ("/"+ / "\\"+)
   / "[/\\/\\/" / "[/\\/"
   / "[" (sd slashes)
   / "{" (sd slashes)
   / lesses sd (slashes / bars)
   / bars sd slashes
   / "{*" /  "[*"
   / "((>" / "(<"

rightEncloser =
     rightEncloserMulti
   / !('|') _

rightEncloserMulti =
     "/"+ ")"
   / "\\"+ ")"
   / slashes sd (greaters / bars / [\]}])
   / bars sd greaters
   / "*]" / "*}"
   / "]" / "}"
   / ">)" / "<))"
   / "/\\/\\/]" / "/\\/]"

exponentOp = "^T" / "^" op

OpName = opn:id &{FortressUtil.validOp(opn) }

Op = condOp / op !(equalsOp) / compOp

compOp =
     "==="
   / "=/="
   / "<="
   / ">="

condOp =
     ":" op ":"
   / op ":"

multiOp =
     "-/->"
   / "<-/-"
   / "-->"
   / "==>"
   / ">>>"
   / mapstoOp
   / "<<<"
   / "<->"
   / leftarrow
   / "<=>"
   / "->"
   / doublerightarrow
   / ">>"
   / "<<"
   / "**"
   / "!!"
   / !(rightEncloserMulti) "///"
   / !(rightEncloserMulti) "//"

singleOp = !(encloser / leftEncloser / rightEncloser / multiOp / compOp / match) _

op = OpName
   / multiOp
   / singleOp

CompoundOp = op equalsOp

/* The operator "=>" should not be in the left-hand sides of case/typecase expressions. */
doublerightarrow = "=>" &(w BlockElems w match)
match = "=>"

/* The operator "BY" should not be used with ExtentRange. */
crossOp = "BY":OpName &(w ExtentRange)

leftarrow = "<-"

lessthanequal    = "<=":op / "LE":op
lessthan         = "<":op  / "LT":op
greaterthanequal = ">=":op / "GE":op
greaterthan      = ">":op  / "GT":op

NOT     = "NOT":op
OR      = "OR":op
AND     = "AND":op
IMPLIES = "->":op / "IMPLIES":op

equalsOp = "=":singleOp

AssignOp =
     ":="
   / CompoundOp

SUM = "SUM"

PROD = "PROD"

Accumulator =
     SUM
   / PROD
   / "BIG" w Op

ArrayComprehensionClause = ArrayComprehensionLeft wr bar wr GeneratorClauseList
```

## E.23 Identifiers

```
id      = s:(idstart idrest*) &{ !FORTRESS_KEYWORDS.contains(s) }
idstart = UnicodeIdStart / "_"
idrest  = UnicodeIdStart / "'" / UnicodeIdRest
IdText  = a1:id &{ !FortressUtil.validOp(a1) && !a1.equals("_") }

Id = IdText

BindId =
     Id
   / "_"

BindIdList = BindId (w "," w BindId)*
```

```
BindIdOrBindIdTuple =
     BindId
   / "(" w BindId w "," w BindIdList w ")"

SimpleName =
     Id
   / "opr" w Op
   / "opr" w EncloserPair

APIName = /* If we find ..., the dot doesn't count */
     Id &(w "...")
   / Id ("." Id)* &(w "...")
   / Id ("." Id)*

QualifiedName = /* If we find ..., the dot doesn't count */
     Id &(w "...")
   / Id ("." Id)* &(w "...")
   / Id ("." Id)*
```

## E.24 Spaces and Comments

```
EndOfFile       = !_
Whitespace =
     Space
   / "\t"       // Error production
   / Newline
Space =
     " "
   / "\f"
   / "\t"       // Error production
   / NoNewlineComment
Newline         = "\r\n" / "\r" / "\n" / NewlineComment
Comment         = "(*" CommentContents "*)"
CommentContents = CommentContent*
CommentContent  = Comment / '*' !')' / c:_ &{c != '*'}
NewlineComment = Comment
NoNewlineComment =
     Comment &{ !(yyValue.contains("\r\n") || yyValue.contains("\r") || yyValue.contains("\n")) }

wValue  = Whitespace*
wrValue = Whitespace+

w  = Whitespace*
wr = Whitespace+

s  = Space*
sr = Space+

nl = s Newline w
br = nl / s ";" w

RectSeparator =
     (w ";")+ w
   / sr
   / nl
```

# Appendix F

# Changes between Fortress 1.0 $\beta$ and 1.0 Specifications

- This release of the Fortress language specification is the first to be released in tandem with a compliant interpreter, available as open source and online at:

  `http://projectfortress.sun.com`

  Each example in the specification is automatically generated from a corresponding working Fortress program which is run by every test run of the interpreter.

- To synchronize the specification with the implementation, we temporarily dropped the following features from the specification:
  - Static checks (including static overloading checks)
  - Static type inference
  - Qualified names (including aliases of imported APIs)
  - Getters and setters
  - Array comprehensions
  - Keyword parameters and keyword arguments
  - Most modifiers
  - Dimensions and units
  - Type aliases
  - Where clauses
  - Coercion
  - Distributions
  - Parallel nested transactions
  - Abstract function declarations
  - Tests and properties
  - Syntactic abstraction

- Libraries have significantly changed (Part IV).
- Syntax and semantics of the following features have changed:
  - Tuple and functional arguments (Chapter 6, Chapter 9, and Section 13.26)
  - Operator rules: associativity, precedence, fixity, and juxtaposition (Chapter 16)
  - Operator declarations (Chapter 22)
  - Extremum expressions (Section 13.20)
  - Import statements (Section 20.2)
  - Multiple variable declarations (Chapter 8)
  - Typecase expressions (Section 13.21)
- The following features have been added to the language:
  - `native` modifier (Chapter 20)
  - Explicit static arguments to big operator applications (Section 13.17 and Section 13.28)
- The following features have been eliminated from the language:
  - Identifier parameters
  - Explicit self parameters of dotted methods
  - Local operator declarations
  - Shorthands for $\mathrm{Set}$, $\mathrm{List}$, and $\mathrm{Map}$ types
  - $\mathrm{Tuple}$ type encompassing all tuple types
- Significantly more examples have been added.

# Bibliography

[1] Ole Agesen, Lars Bak, Craig Chambers, Bay-Wei Chang, Urs Hlzle, John Maloney, Randall B. Smith, David Ungar, and Mario Wolczko. *The Self Programmer's Reference Manual*. `http://research.sun.com/self/release_4.0/Self-4.0/manuals/Self-4.1-Pgmers-Ref.pdf`, 2000.

[2] Eric Allen, Victor Luchangco, and Sam Tobin-Hochstadt. Encapsulated Upgradable Components, March 2005.

[3] Gilad Bracha, Guy Steele, Bill Joy, and James Gosling. *Java(TM) Language Specification, The (3rd Edition) (Java Series)*. Addison-Wesley Professional, July 2005.

[4] R. Cartwright and G. Steele. Compatible genericity with run-time types for the Java Programming Language. In *OOPSLA*, 1998.

[5] Stéphane Ducasse, Oscar Nierstrasz, Nathanael Schärli, Roel Wuyts, and Andrew P. Black. Traits: A mechanism for fine-grained reuse. *ACM Trans. Program. Lang. Syst.*, 28(2):331–388, 2006.

[6] Seth C. Goldstein, Klaus E. Schauser, and Dave E. Culler. Lazy Threads: Implementing a Fast Parallel Call. *Journal of Parallel and Distributed Computing*, 37(1), August 1996.

[7] Robert Grimm. Rats! – an easily extensible parser generator. `http://cs.nyu.edu/~rgrimm/xtc/rats.html`.

[8] Sun's Programming Language Research Group. Project fortress. `http://projectfortress.sun.com`.

[9] Ralf Hinze and Ross Paterson. Finger trees: a simple general-purpose data structure. *Journal of Functional Programming*, 16(2):197–217, March 2006.

[10] Atshushi Igarashi, Benjamin Pierce, and Philip Wadler. Featherweight Java: A minimal core calculus for Java and GJ. In Loren Meissner, editor, *Proceedings of the 1999 ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages & Applications (OOPSLA'99)*, volume 34(10), pages 132–146, N. Y., 1999.

[11] Richard Kelsey, William Clinger, and Jonathan Rees. Revised$^5$ report on the algorithmic language Scheme. *ACM SIGPLAN Notices*, 33(9):26–76, 1998.

[12] Xavier Leroy, Damien Doligez, Jacques Garrigue, Didier Rmy, and Jrme Vouillon. *The Objective Caml System, release 3.08*. `http://caml.inria.fr/distrib/ocaml-3.08/ocaml-3.08-refman.pdf`, 2004.

[13] J. Matthews, R. B. Findler, M. Flatt, and M. Felleisen. A Visual Environment for Developing Context-Sensitive Term Rewriting Systems (system description). In V. van Oostrom, editor, *Rewriting Techniques and Applications, 15th International Conference, RTA-04*, LNCS 3091, pages 301–311, Valencia, Spain, June 3-5, 2004. Springer.

[14] B. Meyer. *Object-oriented Software Construction*. Prentice Hall, 1988.

[15] Todd Millstein and Craig Chambers. Modular statically typed multimethods. *Information and Computation*, 175(1):76–118, May 2002.

[16] Robin Milner, Mads Tofte, Robert Harper, and David MacQueen. *The Definition of Standard ML (Revised)*. The MIT Press, 1997.

[17] Eric Mohr, David A. Kranz, and Robert H. Halstead, Jr. Lazy task creation: A technique for increasing the granularity of parallel programs. Technical Report TM-449, MIT/LCS, 1991.

[18] Martin Odersky, Philippe Altherr, Vincent Cremet, Burak Emir, Stphane Micheloud, Nikolay Mihaylov, Michel Schinz, Erik Stenman, and Matthias Zenger. *The Scala Language Specification*. `http://scala.epfl.ch/docu/files/ScalaReference.pdf`, 2004.

[19] Chris Okasaki. *Purely Functional Data Structures*. Cambridge University Press, Cambridge, UK, 1998.

[20] Simon Peyton-Jones. *Haskell 98 Language and Libraries*. Cambridge University Press, 2003.

[21] Barry N. Taylor. Guide for the use of the international system of units (si). Technical report, United States Department of Commerce, National Institute of Standards and Technology, April 1995.

[22] The Unicode Consortium. *The Unicode Standard, Version 5.0*. Addison-Wesley, 2006.