CSG111 January 16-17, 2007
Dr. Wand Readings: EOPL, Chap. 2

Lecture 2: Data Types and their Representations; Syntax: Scanning and Parsing

Key Concepts:

- Data types
  - The Data Abstraction Principle
  - Interface, specification, implementation
  - Constructors and Observers
  - Information-hiding
  - Representations: data-structure, procedural
- Environments
- Interpreter Recipe
- Scanning, Lexical Specification
  - token (lexical item)
  - Regular expressions
  - Scanning in SLLGEN
- Parsing
  - Context-free grammar
  - Parsing in SLLGEN
  - Concrete and abstract syntax
  - Abstract Syntax Tree
  - `arbno` and `separated-list`

2.1 Data Types

Every time we represent some set of things in a program, we're creating a data type for those things.

Want to arrange things so that we can change the representation without changing code throughout the program.

The Data Abstraction Principle

Divide a data type into two pieces:

1. *An* interface *that tells us what the data of the type represents, what the operations on the data are, and what properties these operations may be relied on to have. ("what")*
2. *An* implementation *that provides a specific representation of the data and code for the operations that depend on that data representation. ("how")*

This way you can change the implementation without changing the code that uses the data type (user = client; implementation = supplier/server). The client should be able to work without knowing how the data type is implemented. This is called *information hiding*. Later on, we'll see how to enforce this.

The vital part of the implementation is the specification of how the data is represented. We will use the notation $\lceil v \rceil$ for "the representation of data v".

2.2 Example: Arithmetic

Data to be Represented: the non-negative integers

Interface:

$$
\begin{aligned}
&(\texttt{zero}) = \lceil 0 \rceil \\
&(\texttt{is-zero?}\ \lceil n \rceil) = \begin{cases} \texttt{\#t} & n = 0 \\ \texttt{\#f} & n \neq 0 \end{cases} \\
&(\texttt{succ}\ \lceil n \rceil) = \lceil n+1 \rceil \\
&(\texttt{pred}\ \lceil n+1 \rceil) = \lceil n \rceil
\end{aligned}
$$

Now we can write procedures to do the other arithmetic operations, and these will work no matter what representation of the natural numbers we use.

```
(define plus
  (lambda (x y)
    (if (is-zero? x) y
      (succ (plus (pred x) y)))))
```

will satisfy $(\texttt{plus}\ \lceil x \rceil\ \lceil y \rceil) = \lceil x+y \rceil$, no matter what implementation of the integers we use.

Or similarly:

```
(define times
  (lambda (x y)
    (if (is-zero? x) (zero)
      (plus y (times (pred x) y)))))
```

Implementations:

1. Unary representation:

$$\lceil 0 \rceil = \texttt{()}$$
$$\lceil n+1 \rceil = \texttt{(cons \#t } \lceil n \rceil \texttt{)}$$

So the integer n is represented by a list of n `#t`'s. e.g., 0 is represented by `()`, 1 is represented by `(#t)`, 2 is represented by `(#t #t)`, etc.

It's easy to see that we can satisfy the specification by writing:

```
(define zero '())

(define is-zero? null?)

(define succ
  (lambda (n) (cons #t n)))

(define pred cdr)
```

2. Scheme number representation:

$\lceil n \rceil$ = the Scheme integer `n`

```
(define zero 0)
(define is-zero? zero?)
(define succ (lambda (n) (+ n 1)))
(define pred (lambda (n) (- n 1)))
```

3. Bignum representation (Base B, least-significant bigit first):

$$\lceil n \rceil = \begin{cases} \texttt{()} & n = 0 \\ \texttt{(cons } r \lceil q \rceil \texttt{)} & n = qB + r,\ 0 \le r < N \end{cases}$$

So if $N = 16$, $\lceil 33 \rceil$ = `(1 2)`, $\lceil 258 \rceil$ = `(2 0 1)` ($258 = 1 \times 16^2 + 0 \times 16^1 + 2 \times 16^0$). Exercise: write the operations in this implementation.

2.3 Environments

2.3.1 Interface

Data: An *environment* is a function whose domain is a finite set of *Scheme* symbols, and whose range is the set of all Scheme values.

Typical environment:

$$\{(s_1, v_1), \ldots, (s_n, v_n)\}$$

where the s_i are distinct symbols and the v_i are any Scheme values.

We sometimes call the value of s in an environment *env* its *binding* in *env*.

Three procedures in the interface:

$$\begin{array}{ll} \texttt{(empty-env)} & = \lceil \emptyset \rceil \\ \texttt{(apply-env } \lceil f \rceil \texttt{ } s\texttt{)} & = f(s) \\ \texttt{(extend-env } s \texttt{ } v \texttt{ } \lceil f \rceil\texttt{)} & = \lceil g \rceil, \\ & \text{where } g(s') = \begin{cases} v & \text{if } s' = s \\ f(s') & \text{otherwise} \end{cases} \end{array}$$

Example:

```
> (define e
    (extend-env 'd 6
      (extend-env 'y 8
        (extend-env 'x 7
          (extend-env 'y 14
            (empty-env))))))
```

constructs a scheme value `e` which is a representation of the environment

$$\{(\texttt{d}, 6), (\texttt{y}, 8), (\texttt{x}, 7)\}$$

There are lots of other ways to build representations of the same environment.

Constructors and Observers Can divide the functions in any interface into

- *constructors* that build elements of the data type, and
- *observers* that extract information from values of the data type.

Here `empty-env` and `extend-env` are the constructors, and `apply-env` is the only observer.

2.3.2 Data Structure Representation

We can obtain a representation of environments by observing that every environment can be built by starting with the empty environment and applying `extend-env` n times, for some $n \geq 0$, *e.g.*,

```
(extend-env s_n v_n
   ...
   (extend-env s_1 v_1
     (empty-env))...)
```

So every environment can be built by an expression in the following grammar:

Env-exp ::= `(empty-env)`
::= `(extend-env` *Symbol Scheme-value Env-exp*`)`

Can use the same grammar to describe a set of lists. This gives the following implementation:

```
;; env ::= (empty-env) | (extend-env sym val env)

empty-env : () -> env
(define empty-env
  (lambda () '(empty-env)))

extend-env : sym * val * env -> env
(define extend-env
  (lambda (sym val saved-env)
    (list 'extend-env sym val saved-env)))

apply-env : env * sym -> val
(define apply-env
  (lambda (env sym)
    (cond
      ((eqv? (car env) 'empty-env)
       (eopl:error 'apply-env
         "No binding for ~s" sym))
      ((eqv? (car env) 'extend-env)
       (let ((saved-sym  (cadr env))
             (saved-val (caddr env))
             (saved-env  (cadddr env)))
         (if (eqv? saved-sym sym)
           saved-val
           (apply-env saved-env sym))))
      (else
        (eopl:error 'apply-env
          "Bad environment: ~s" env)))))
```

The procedure `apply-env` looks at the data structure `env` representing an environment, determines what kind of environment it represents, and does the right thing. If it represents the empty environment, then an error is reported. If it represents an environment built by `extend-env`, then it checks to see if the symbol it is looking for is the same as the one bound in the environment. If it is, then the saved value is returned. Otherwise, the symbol is looked up in the saved environment.

This is a very common pattern of code. We call it the *interpreter recipe*:

The Interpreter Recipe

1. *Look at a piece of data.*
2. *Decide what kind of data it represents.*
3. *Extract the components of the datum and do the right thing with them.*

We call this dead-simple data structure representation the *abstract syntax tree* or *AST* representation because it represents the syntax of the expression that built the environment.

Of course we could use lots of different data structures instead. For example, we could say

```
env ::= () | (sym val . env)
```

In this representation, we'd write:

```
(define empty-env
  (lambda ()
    '()))

(define extend-env
  (lambda (sym val saved-env)
    (cons sym
      (cons val saved-env))))
```

[Easy puzzle: what would `apply-env` look like for this representation?]

2.3.3 Procedural Representation

The environment interface has an important property: it has only one observer, `apply-env`. This allows us to represent an environment as a *Scheme* procedure that takes a symbol and returns its associated value.

To do this, define `empty-env` and `extend-env` to return procedures that, when applied, do the same thing that `apply-env` did in the preceding section.

```
(define empty-env
  (lambda ()
    (lambda (sym)
      (eopl:error 'apply-env "No binding for ~s" sym))))

(define extend-env
  (lambda (saved-sym saved-val saved-env)
    (lambda (sym)
      (if (eqv? saved-sym sym)
          saved-val
          (apply-env saved-env sym)))))

(define apply-env
  (lambda (env sym)
    (env sym)))
```

If the empty environment, created by invoking `empty-env`, is passed any symbol whatsoever, it indicates with an error message that the given symbol is not in its domain. The procedure `extend-env` returns a new procedure that represents the extended environment. This procedure, when passed a symbol `sym`, checks to see if the symbol it is looking for is the same as the one bound in the environment. If it is, then the saved value is returned. Otherwise, the symbol is looked up in the saved environment.

We call this a *procedural representation*, in which the data is represented by its *action under* `apply-env`.

This turns out to be a useful technique in more situations than you might think, eg whenever the set of values being represented is a set of mathematical functions.

2.4 Lexical Scanning

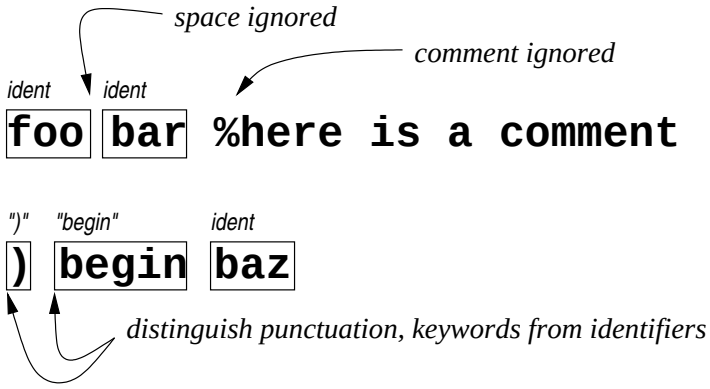
Programs are just strings of characters. Need to group these characters into meaningful units. This grouping is divided into two stages: *scanning* and *parsing*.

Scanning is dividing the sequence of characters into words, punctuation, etc. These units are called *lexical items*, *lexemes*, or most often *tokens*. Refer to this as the *lexical structure* of the language.

Parsing is organizing the sequence of tokens into hierarchical syntactic structures such as expressions, statements, and blocks. This is like organizing (diagramming) an English sentence into clauses, etc. Refer to this as the *syntactic* or *grammatical* structure of the language.

Typical pieces of lexical specification:

- Any sequence of spaces and newlines is equivalent to a single space.
- A comment begins with `%` and continues until the end of the line.
- An identifier is a sequence of letters and digits starting with a letter, and a variable is an identifier that is not a keyword.



2.4.1 What's in a token?

Data structure for token consists of three pieces:

- A *class*, a Scheme symbol that describes what kind of a token you've found. The set of classes is part of the lexical specification.
- A piece of data describing the particular token. The nature of this data is also part of the lexical specification. For our system, the data will be as follows: For identifiers, the datum is a Scheme symbol built from the string in the token. For a number, the datum is the number described by the number literal. For a literal string, the datum is the string (used for keywords and punctuation)

  In a language that didn't have symbols, we might use a string (the name of the identifier), or an entry into a hash table indexed by identifiers (a *symbol table*) instead. Using Scheme spares us these annoyances.
- Debugging information, such as line and character numbers.

Job of the scanner is to go through the input and analyze it to produce these tokens. Generally the scanner is a subroutine that, when called, produces the "next" token of the input.

Could write scanner from scratch, but that's tedious and error-prone. Better idea: write down the lexical specification in a specialized language. Such a specialized language is called a *domain-specific language*, or *DSL*.

2.4.2 Specification via regular expressions

Most common language is a language of *regular expressions*. Regular expressions are widely used for pattern matching (eg in Perl, grep, etc.).

$$\begin{array}{rcl} R & ::= & \mathit{character} \mid R\,R \mid R \cup R \mid R^* \\ & \mid & \neg \mathit{character} \end{array}$$

What do these mean?

A character c matches the string consisting of the character c.

$\neg c$ matches any 1-character string other than c.

RS matches any string that consists of a string matching R followed by a string matching S. (This is called *concatenation*).

$R \cup S$ matches any string that either matches R or matches S. (This is sometimes written $R \mid S$).

R^* matches any number n of copies ($n \geq 0$) of strings that match R. This is called the *Kleene closure* of R.

Examples:

ab matches only the string `ab`.

$ab \cup cd$ matches the strings `ab` and `cd`.

$(ab \cup cd)(ab \cup cd \cup ef)$ matches `abab`, `abcd`, `abef`, `cdab`, `cdcd`, or `cdef`.

$(ab)^*$ matches the empty string, `ab`, `abab`, `ababab`, `abababab`, ...

$(ab \cup cd)^*$ matches the empty string, `ab`, `cd`, `abab`, `abcd`, `cdab`, `cdcd`, `ababab`, ... `cdcdcd`, ...

Regular expressions for our example:

$$\mathit{whitespace} = (\mathit{space} \cup \mathit{newline})\ (\mathit{space} \cup \mathit{newline})^*$$

$$\mathit{comment} = \texttt{\%}\ (\neg \mathit{newline})^*$$

$$\mathit{identifier} = \mathit{letter}\ (\mathit{letter} \cup \mathit{digit})^*$$

Rule for scanners: always take the *longest* match. This way `xyz` comes out as one identifier, not 3.

2.4.3 Specifying scanners in SLLGEN

SLLGEN is a package for generating scanners and parsers in Scheme.

In SLLGEN, scanners are specified by regular expressions, which are written as lists in Scheme. Here's our example:

```
(define lex0
  '((whitespace (whitespace)                        skip)
    (comment    ("%" (arbno (not #\newline)))       skip)
    (identifier (letter (arbno (or letter digit))) symbol)
    (number     (digit (arbno digit))              number)))
```

Don't have to worry about keywords when writing a lexical specification: the parser-generator will add those automatically.

2.4.4 Testing the scanner

`sllgen:make-string-scanner` generates a scanning procedure. You'll probably only need it for testing. First argument is the lexical specification, second is the grammar (that's empty here).

Demo: here I've put `lex0` in a module and opened it in DrScheme.

```
Welcome to DrScheme, version 299.400p1.
Language: (module ...).
drscheme-init.scm plt209.1.5 10feb2005
> (define scan0 (sllgen:make-string-scanner lex0 '()))
> (scan0 "foo bar %here is a comment
begin     baz")
((identifier foo 1)                    ; 1 is a line number
 (identifier bar 1)
 (identifier begin 2)                  ; no keywords in lex0
 (identifier baz 2))
```

2.4.5 The SLLGEN scanner specification language

```
scanner ::= (regexp-and-action ...)
regexp-and-action ::= (name (regexp ...) outcome)
name ::= symbol
regexp ::= tester
   | (or regexp ...)     matches the OR of the regexps
   | (arbno regexp)      STAR of the regexp
   | (concat regexp ...) concatenation of the regexps
tester ::= string     matches the string
   | LETTER          matches any letter
   | DIGIT           matches any digit
   | WHITESPACE      matches any Scheme whitespace character
   | ANY             matches any character
   | (NOT char)      matches any character other than the given char
outcome :: = SKIP | SYMBOL | NUMBER | STRING
```

A scanner is a list. Each item in the list is a specification of a regular expression, consisting of a name, a sequence of regular expressions, and an action to be taken on success.

The name is a Scheme symbol. It is the name of lexical class being defined.

The second part of the specification is a *sequence* of regular expressions, because the top level of a regexp in a scanner is almost always a concatenation. Each regular expression in the sequence follows the obvious Scheme-like syntax: we use `or` and `concat` for union and concatenation, and `arbno` for Kleene star.

The base cases for the regular expressions are Scheme strings, negation (of a character, not a string– see our example), and four predefined testers: `letter` (matches any letter), `digit` (matches any digit), `whitespace` (matches any Scheme whitespace character), and `any` (matches any character).

As the scanner works, it collects characters into a buffer. When the scanner determines that it has found the longest possible match of all the regular expressions in the specification, it executes the *outcome* of the corresponding regular expression.

An outcome can be one of the following:

- The symbol `skip`. This means this is the end of a token, but no token is emitted. The scanner continues working on the string to find the next token. This action is used for whitespace and comments. The class name is ignored.
- The symbol `symbol`. The characters in the buffer are converted into a Scheme symbol and a token is emitted, with the class name as its class and with the symbol as its datum.
- The symbol `number`. The characters in the buffer are converted into a Scheme number, and a token is emitted, with the class name as its class and with that number as its datum.
- The symbol `string`. The characters in the buffer are converted into a Scheme string, and a token is emitted, with the class name as its class and with that string as its datum.

If there is a tie for longest match between two regular expressions, `string` takes precedence over `symbol` (so keywords that would otherwise be identifiers show up as keywords).

2.5 Parsing

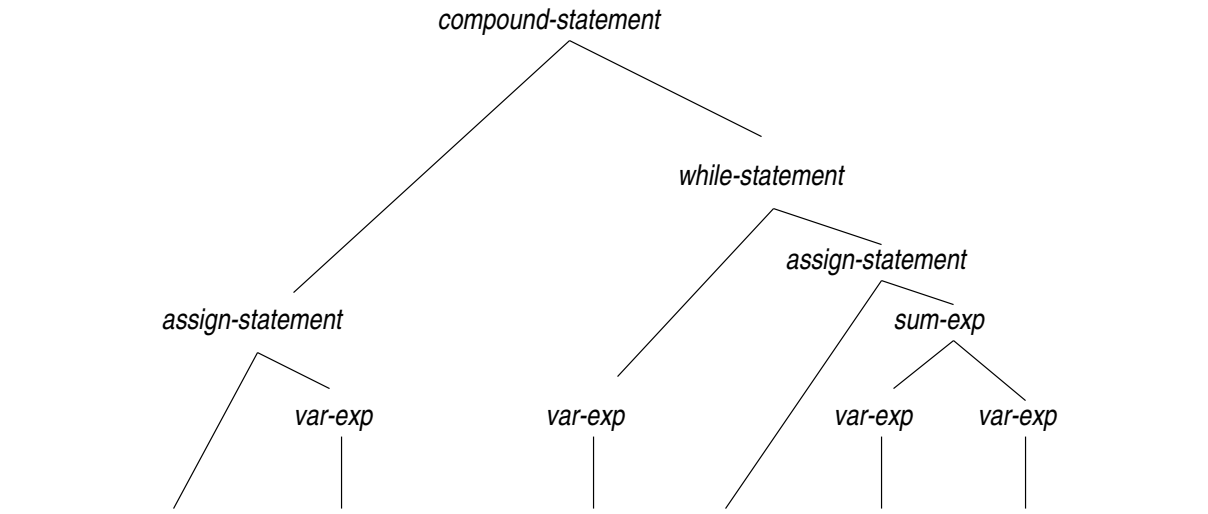
Parsing is organizing the sequence of tokens into hierarchical syntactic structures such as expressions, statements, and blocks. This is like organizing (diagramming) an English sentence into clauses, etc. Refer to this as the *syntactic* or *grammatical* structure of the language.

Output of parser is a *tree*, called *parse tree* or *abstract syntax tree* (AST). Mostly we'll call these AST's.

Example:

```
(define-datatype statement statement?
  (compound-statement
    (stmt1 statement?)
    (stmt2 statement?))
  (while-statement
    (test expression?)
    (body statement?))
  (assign-statement
    (lhs symbol?)
    (rhs expression?)))

(define-datatype expression expression?
  (var-exp
    (id symbol?))
  (sum-exp
    (exp1 expression?)
    (exp2 expression?)))
```




```
;;; input:

begin x := foo; while x do x := (x + bar) end

;;; output:

#(struct:compound-statement
  #(struct:assign-statement x #(struct:var-exp foo))
  #(struct:while-statement
    #(struct:var-exp x)
    #(struct:assign-statement x
       #(struct:sum-exp
          #(struct:var-exp x)
          #(struct:var-exp bar)))))
```

2.5.1 Specification via context-free grammars

We've seen lots of grammars so far. These are called *context-free* grammars. (Why context-free? To find out, go take automata theory).

Our example,

$$
\begin{aligned}
\textit{Statement} &::= \texttt{begin}\ \textit{Statement}\ \texttt{;}\ \textit{Statement}\ \texttt{end} \\
&::= \texttt{while}\ \textit{Expression}\ \texttt{do}\ \textit{Statement} \\
&::= \textit{Identifier}\ \texttt{:=}\ \textit{Expression} \\
\textit{Expression} &::= \textit{Identifier} \\
&::= \texttt{(}\textit{Expression}\ \texttt{+}\ \textit{Expression}\texttt{)}
\end{aligned}
$$

A non-terminal can produce a string by applying a production, eg:

```
expression -> ( expression + expression )     sum-exp
           -> ( identifier + expression )     var-exp
           -> ( X + expression)
           -> ( X + ( expression + expression))  sum-exp
           -> ( X + ( identifier + expression))  var-exp
           -> ( X + ( Y          + expression))
           -> ( X + ( Y          + identifier))  var-exp
           -> ( X + ( Y          + Z         ))
```

We associate a *name* with each production; these become the variants for our AST's; the AST documents the productions that were used to build the string. We don't have to keep non-terminal names or punctuation in the AST, because this information is already in the production.

2.5.2 Specifying grammars in SLLGEN

SLLGEN includes a sublanguage for specifying grammars. Here's our example again:

```
(define simple-while-grammar
  '((statement
      ("begin" statement ";" statement "end")
      compound-statement)
    (statement
      ("while" expression "do" statement)
      while-statement)
    (statement
      (identifier ":=" expression)
      assign-statement)
    (expression
      (identifier)
      var-exp)
    (expression
      ("(" expression "+" expression ")")
      sum-exp)))
```

This is in `http://www.ccs.neu.edu/course/csg111/interps/lecture02/grammar1.scm`.

Testing the grammar

```
;;; Scaffolding

;; build the datatypes from the grammar
(sllgen:make-define-datatypes lex0 simple-while-grammar)

;; build a scanner (optional)
(define just-scan
  (sllgen:make-string-scanner lex0 simple-while-grammar))

;; build a parser
(define scan&parse
  (sllgen:make-string-parser lex0 simple-while-grammar))

(define show-the-datatypes
  (lambda () (sllgen:show-define-datatypes lex0 simple-while-grammar)))

(define stmt1 "begin x := foo; while x do x := (x + bar) end")
```

The procedure `sllgen:make-string-parser` generates both a scanner and parser. It takes all the literal strings from the grammar and adds them to the regexps of the scanner, so you don't have to worry about this.

2.5.3 Running the tests

```
> (scan&parse stmt1)
#(struct:compound-statement
  #(struct:assign-statement x #(struct:var-exp foo))
  #(struct:while-statement
    #(struct:var-exp x)
    #(struct:assign-statement x
       #(struct:sum-exp
          #(struct:var-exp x)
          #(struct:var-exp bar)))))
> (show-the-datatypes)
(define-datatype statement statement?
  (compound-statement
    (compound-statement13 statement?)
    (compound-statement14 statement?))
  (while-statement
    (while-statement15 expression?)
    (while-statement16 statement?))
  (assign-statement
    (assign-statement17 symbol?)
    (assign-statement18 expression?)))
(define-datatype expression expression?
  (var-exp
    (var-exp19 symbol?))
  (sum-exp
    (sum-exp20 expression?)
    (sum-exp21 expression?)))
```

The field names are uninformative because the information isn't in the grammar; if you want better field names you can always write out the `define-datatype` by hand.

These `define-datatypes` show what goes in the node for a production: a subtree for each non-terminal, and the data field for each data-bearing terminal (identifiers and numbers). Literal strings aren't stored, because they are the same for every instance of the production.

2.5.4 The SLLGEN grammar specification language

```
grammar ::= (production ...)      ; nonterm of first prod is
                                  ; start symbol.

production ::= (lhs rhs prod-name)

lhs ::= symbol                    becomes a datatype name

rhs ::= (rhs-item ...)

rhs-item ::= symbol | string
  | (ARBNO rhs-item ...) | (SEPARATED-LIST rhs-item ... token)

prod-name ::= symbol  becomes a variant name
```

A grammar is a list of productions. The left-hand side of the first production is the start symbol for the grammar.

Each production consists of a left-hand side (a non-terminal symbol), a right-hand side (a list of rhs-item's) and a production name.

The right-hand side of a production is a list of symbols or strings (or `arbno`'s or `separated-list`'s – we'll talk about those later). Symbols are non-terminals (if they don't appear on lhs, or as a lexical class, an error will be reported); strings are literal strings.

The production name is a symbol which becomes the variant of the `define-datatype`.

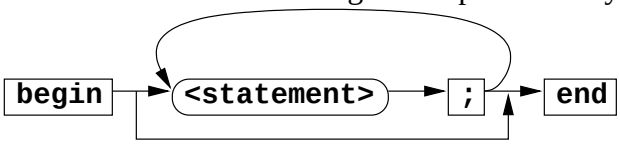
Important: The parser must be able to figure out which production to use knowing only (1) what non-terminal it's looking for and (2) the first symbol (token) of the string being parsed. Grammars in this form are called *LL*(1) grammars (SLLGEN stands for Scheme LL(1) parser GENerator). This will be good enough for our purposes. SLLGEN will warn you if your grammar does not meet this restriction.

2.5.5 `arbno`'s and `separated-list`'s

An `arbno` is a Kleene star in the grammar: it matches an abitrary number of repetitions of its entry. Example: Change our grammar to say

```
(define the-grammar
  '((statement
      ("begin" (arbno statement ";") "end")
      compound-statement)
    ...))
```

This makes a compound statement a sequence of (an arbitrary number of) semicolon-*terminated* statements. Such a statement might be represented by a diagram like



This `arbno` generates a single field in the AST, which will contain a *list* of the data for the thing inside the `arbno`.

Our example (in `grammar2.scm`) generates the following datatypes:

```
> (sllgen:show-define-datatypes lex0 the-grammar)
(define-datatype
  statement
  statement?
  (compound-statement
    (compound-statement32 (list-of statement?)))  ;; <<<------
  (while-statement
    (while-statement29 expression?)
    (while-statement28 statement?))
  (assign-statement
    (assign-statement31 symbol?)
    (assign-statement30 expression?)))
(define-datatype
  expression
  expression?
  (var-exp
    (var-exp27 symbol?))
  (sum-exp
    (sum-exp26 expression?)
    (sum-exp25 expression?)))
```

Here's another example:

```
> (define scan&parse
    (sllgen:make-string-parser lex0 the-grammar))

> (scan&parse "begin x := foo; y := bar; z := uu; end")
#(struct:compound-statement
  (#(struct:assign-statement x #(struct:var-exp foo))
   #(struct:assign-statement y #(struct:var-exp bar))
   #(struct:assign-statement z #(struct:var-exp uu))))
```

The `compound-statement` has 1 data field, which contains a *list* of the statements inside the begin-end.

Another example We can put more complicated things inside an `arbno`. For example:

```
(define grammar3
  '((expression (number) lit-exp)
    (expression (identifier) var-exp)
    (expression
      ("let" (arbno identifier "=" expression) "in" expression)
      let-exp)
    (expression
      ("(" "-" (arbno expression) ")")
      sum-exp)
    ))

(sllgen:make-define-datatypes lex0 grammar3)

(define scan&parse
  (sllgen:make-string-parser lex0 grammar3))

> (scan&parse "let x = (- y z) u = (- g h) v = (- i j)
                 in (- (- x u) (- v k))")
#(struct:let-exp
  (x u v)
  (#(struct:sum-exp #(struct:var-exp y) #(struct:var-exp z))
   #(struct:sum-exp #(struct:var-exp g) #(struct:var-exp h))
   #(struct:sum-exp #(struct:var-exp i) #(struct:var-exp j)))
  #(struct:sum-exp
    #(struct:sum-exp #(struct:var-exp x) #(struct:var-exp u))
    #(struct:sum-exp #(struct:var-exp v) #(struct:var-exp k))))
```

Here `(arbno identifier "=" expression)` generated 2 lists: a list of identifiers and a list of expressions. This is handy because it will let our interpreters get at the pieces of the expression directly.

Let's look at the datatypes generated for `grammar3`:

```
(define grammar3
  '((expression (number) lit-exp)
    (expression (identifier) var-exp)
    (expression
      ("let" (arbno identifier "=" expression) "in" expression)
      let-exp)
    (expression
      ("(" expression (arbno expression) ")")
      app-exp)
    ))

>  (sllgen:show-define-datatypes lex0 grammar3)
(define-datatype
  expression
  expression?
  (lit-exp
    (lit-exp10 number?))
  (var-exp
    (var-exp11 symbol?))
  (let-exp
    (let-exp12 (list-of symbol?))       ; <---
    (let-exp13 (list-of expression?))   ; <---
    (let-exp14 expression?))
  (sum-exp
    (sum-exp15 (list-of expression?))))
```

The rule is that a rhs-item inside an `arbno` generates a list of whatever it would generate if the `arbno` weren't there. So if we wrote

```
(define grammar-3a
  '((nt0
      ("begin" nt5 (arbno nt1 "foo" nt2 "bar" nt3 "thing") nt4)
      prod-1)
    ...))
```

We would generate one list for each `rhs-item` in the `arbno`. But of course strings don't count, so this example will generate 3 lists. So the `define-datatype` would look like:

```
(define-datatype nt0 nt0?
  (prod-1
    (field1 nt5?)
    (field2 (list-of nt1?))
    (field3 (list-of nt2?))
    (field4 (list-of nt3?))
    (field5 nt4?))
  ...)
```

We can also have nested `arbno`'s in which case you'll get a list of lists. We'll get to those a little later.

Separated Lists

Sometimes you want lists with separators, not terminators. This is common enough that it is a built-in operation in SLLGEN. Just write:

```
(define grammar4
  '((statement
      ("begin" (separated-list statement ";") "end")
      compound-statement)
    ;...
    ))

> (sllgen:show-define-datatypes lex0 grammar4)
(define-datatype
  statement
  statement?
  (compound-statement
    (compound-statement103 (list-of statement?))) ; <<<-- same as for ARBNO
  (while-statement
    (while-statement100 expression?)
    (while-statement99 statement?))
  (assign-statement
    (assign-statement102 symbol?)
    (assign-statement101 expression?)))
...

> (define scan&parse
   (sllgen:make-string-parser lex0 grammar-4))
> (scan&parse "begin end")
#(struct:compound-statement ())
> (scan&parse "begin x := foo; y := bar; z := uu end")
#(struct:compound-statement
  (#(struct:assign-statement x #(struct:var-exp foo))
   #(struct:assign-statement y #(struct:var-exp bar))
   #(struct:assign-statement z #(struct:var-exp uu))))
> (scan&parse "begin x := foo; y := bar; z := uu; end")
parsing: at line 1:
nonterminal <seplist12> can't begin with literal-string14 "end"
```

Nested arbnos

We will occasionally use nested `arbno`'s and `separated-list`'s. A non-terminal inside an `arbno` generates a list, so a non-terminal inside an `arbno` inside an `arbno` generates a list of lists.

As an example, consider a `begin` like the last one, except that we have parallel assignments:

```
(define grammar5
  '((statement
      ("begin"
        (separated-list
          (separated-list identifier ",")
          ":="
          (separated-list expression ",")
          ";")
        "end")
      compound-statement)
    (expression (number) lit-exp)
    (expression (identifier) var-exp)
    ))

> (define scan&parse
   (sllgen:make-string-parser lex0 grammar5))
> (scan&parse "begin x,y := u,v ; z := 4; t1, t2 := 5, 6 end")
#(struct:compound-statement
  ((x y) (z) (t1 t2))    ;; <-- list of lhs's, each lhs is a list of ids.
  ((#(struct:var-exp u)  ;; <-- list of rhs's, each rhs is a list of exps.
    #(struct:var-exp v))
   (#(struct:lit-exp 4))
   (#(struct:lit-exp 5)
    #(struct:lit-exp 6))))
```

Here of course I've used `separated-list` instead of `arbno`, but they generate the same data.

```
> (sllgen:show-define-datatypes lex0 grammar5)
(define-datatype
  statement
  statement?
  (compound-statement
    (compound-statement4 (list-of (list-of symbol?)))
    (compound-statement3 (list-of (list-of expression?)))))
(define-datatype
  expression
  expression?
  (lit-exp (lit-exp2 number?))
  (var-exp (var-exp1 symbol?)))
```

This looks hairy, but it turns out to be very natural.

2.5.6 SLLGEN error messages

SLLGEN can generate a bunch of error messages. These may occur at parser-generation time or at parse time.

The most common errors arise at parse time, when your string doesn't match the grammar.

```
> (define simple-while-grammar
    '((statement
        ("begin" statement ";" statement "end")
        compound-statement)
      (statement
        ("while" expression "do" statement)
        while-statement)
      (statement
        (identifier ":=" expression)
        assign-statement)
      (expression
        (identifier)
        var-exp)
      (expression
        ("(" expression "+" expression ")")
        sum-exp)))
> (define scan&parse
    (sllgen:make-string-parser lex0 simple-while-grammar))
> (scan&parse "begin x := foo; while x do x := (x + bar); end")
Error reported by sllgen during parsing:
at line 1
Looking for "end", found literal-string22 ";" in production
((string "begin") (non-term statement) (string ";")
(non-term statement) (string "end") (reduce compound-statement))
debug> r
>
```

This is an error during parsing. It was trying to match the production that it represents as:

```
((string "begin")
 (non-term statement)
 (string ";")
 (non-term statement)
 (string "end")
 (reduce compound-statement))
```

It was looking for the string `end`, but it found something else (the `;`). This could mean one of several things:

1. The most likely explanation is that the string you gave it was wrong: it didn't fit the grammar. This is the kind of message you get when you feed the parser an ungrammatical string.
2. Another possibility is that the grammar was incorrect. (If the string and the grammar don't agree, one of them is wrong, and either one could be the culprit).
3. A third possibility is that the scanner is wrong and gave the wrong scan of the string. This is possible, though relatively unlikely.

In this case, the string was wrong: it had a terminating `;` where it was not legal.

Here's another one:

```
> (scan&parse "begin x:= foo; while x do x := (x + 1) end")
Error reported by sllgen during parsing:
at line 1
Nonterminal <expression> can't begin with number 1
debug> r
>
```

Here it was looking for an expression, and it found the number `1`. But there is no production that allows an expression to begin with a number!

```
(define grammar-34
  '((statement
      ("begin" (separated-list statement ";") "end")
      compound-statement)
    ...))
> (define scan&parse4
   (sllgen:make-string-parser lex0 grammar-34))
> (scan&parse4 "begin x:= y; u := v ; z := t ; end")
Error reported by sllgen during parsing:
at line 1
Nonterminal <seplist45> can't begin with literal-string43 "end"
debug> r
>
```

`<seplist45>` is a new non-terminal generated by a `separated-list`. So it was looking for an `(separated-list` *`something`*`)`, where *something* is something that can't begin with an `end`, and it found an `end` in the input string when it got to this point. Alas, the name of the nonterminal here is often uninformative, but the line number can help. Similar nonterminals are generated for an `arbno`.

Of course, in real life, it may not be so simple to find the error!

Errors may also come during the parser generation process:

```
> (define grammar-34a
    '((statement
        ("begin" (separated-list statement ";") "end")
        compound-statement)
      (statement
        ("while" expression do statement)
        while-statement)
      (statement (identifier ":=" expression) assign-statement)
      (expression (identifier) var-exp)
      (expression ("(" expression "+" expression ")") sum-exp)))
> (sllgen:make-define-datatypes lex0 grammar-34a)
Error reported by sllgen during defining-datatypes:

Illegal item do (unknown symbol) in rhs ("while" expression do statement)
debug> r
>
```

Here `sllgen:make-define-datatypes` got confused because it couldn't figure out what `do` was: it wasn't a terminal, because it didn't occur in `lex0`, and it wasn't a non-terminal, because it didn't occur on the left-hand side of a production in `grammar-34a`.

Similar errors will arise if you do things like misspell nonterminals in a grammar, or if the scanner and the parser disagree on the spelling of the lexical classes.

```
> (define grammar-35
    '((statement
        ("begin" statement ";" statement "end"))))
> (define scan&parse
    (sllgen:make-string-parser lex0 grammar-35))

Error in caddr: incorrect list structure
(statement ("begin" statement ";" statement "end")).
```

Here the grammar itself is syntactically incorrect: I forgot to put a name on this production.

```
> (define grammar-35
    '((statement
        ("begin" statement ";" statement "end")
        prod-1)
      (statement
        ("begin" (separated-list statement ";") "end")
        prod-2)))
> (define scan&parse
    (sllgen:make-string-parser lex0 grammar-35))
Error reported by sllgen during parser-generation:

Grammar not LL(1): Shift conflict detected for class "begin"
in nonterminal statement:
(("begin") (string "begin") (arbno seplist7 1) (string "end") (reduce prod-2))
(("begin") (string "begin") (non-term statement) (string ";")
(non-term statement) (string "end") (reduce prod-1))
```

This means that the grammar failed the LL(1) test: there are two productions for `statement` that both start with `begin`.

There are many more error messages, but these are the most common. Please let me know if you get an error message you can't figure out. I'll share these with the class (anonymously) so that everyone will share the results.