CSG111 January 9-10, 2007
Dr. Wand Readings: EOPL, Chap. 1, Sec 2.4

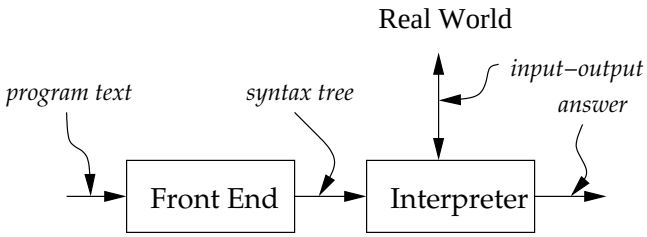
Lecture 1: Background

Key Concepts:

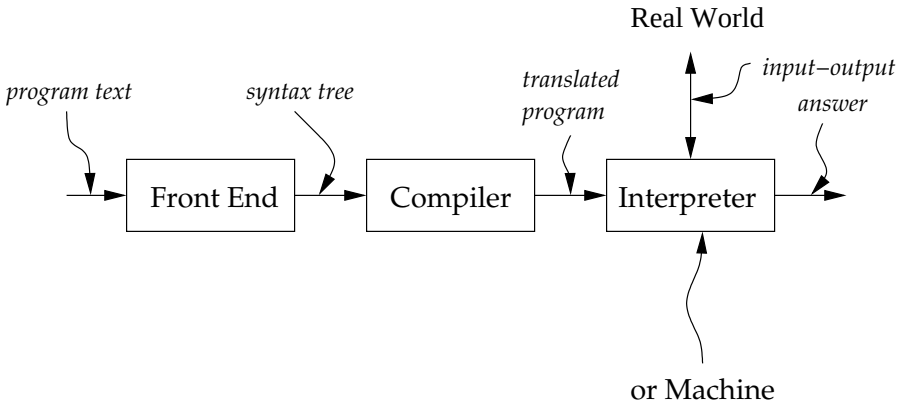
Front End, Interpreter, Compiler
Analyzer, Translator
Specification, Implementation
The Four Questions
Inductive Definitions
  Top-Down, Bottom-Up
  Rules of Inference, Grammars
Recursive Procedures
  The Smaller-Subproblem Principle
  Follow the Grammar!
  Generalization
  Auxiliary Procedures and Context Arguments No Mysterious Auxiliaries!
Tail Recursion
Testing
Representing Trees
  `define-datatype`
  `cases`
`letrec`
Sequencing

1.1 The Big Picture

How a program gets executed:

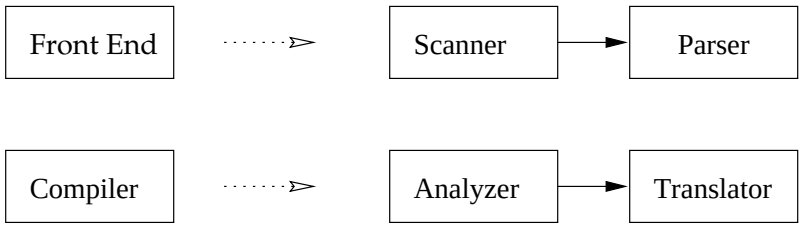


(a) Execution via interpreter

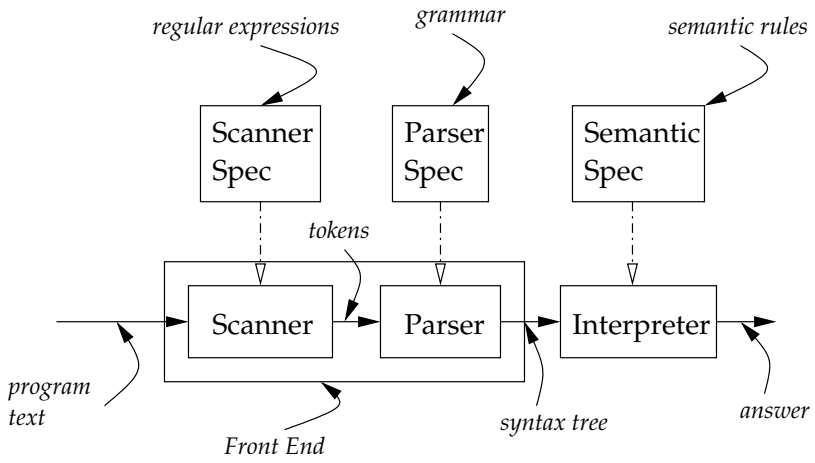


(b) Execution via Compiler

These components may get refined:



We will be concerned with language *specification*. We write each part of the specification in an appropriate specification language.



Sometimes you can automatically generate the component from the specification (scanners, parsers), sometimes it's impossible; occasionally it's possible but better to do it by hand. But usually it's better to use a tool.

There are standard tools for generating scanners and parsers; we will use our own tool, called SLLGEN, for that.

We will, however, write our interpreters by hand, in Scheme.

It's always important to distinguish between a language we are studying or implementing, sometimes called the *object language*, and the language we use to implement it, called the *implementation language*. We will have many small object languages, but our implementation language will always be Scheme.

1.2 The Four Questions

When looking at a language, we will always ask four questions. As we proceed through the course, we will ask these questions in more and more sophisticated ways; I'll show some of these subquestions now, even though we haven't yet covered enough to understand what they mean:

1. **What are the values in the language?**
    - What are the values manipulated by the language, and what operations on those values are represented in the language?
    - What are the expressed and denoted values in the language?
    - What are the types in the language?
2. **What are the scoping rules of the language?**
    - How are variables bound? How are they used?
    - What variables are in scope where?
3. **What are the effects in the language?**
    - Are there side-effects in the language?
    - Can execution of programs have effects in the world?
    - Can execution of programs have effects on other programs?
    - Can execution of a program fail to terminate?
    - Are there non-local control effects in the language?
4. **What are the static properties of the language?**
    - What can we predict about the behavior of a program without knowing the run-time values?
    - How can we analyze a program to predict this behavior?

1.3 Inductive Definitions of Sets

Know Your Sets

When writing code for a procedure, you must know precisely what kinds of values may occur as arguments to the procedure, and what kinds of values it is legal for the procedure to return.

Definition 1 (Example of a Top-Down Definition) *A non-negative integer n is in S whenever*

1. $n = 0$, *or*
2. $n - 3 \in S$.

S is the set of multiples of 3. This definition leads to a test:

```
in-S? : non-negative-integers -> boolean
usage: (in-S? n) = #t if n is in S, #f otherwise
(define in-S?
  (lambda (n)
    (if (zero? n) #t
      (and
        (>= (- n 3) 0)
        (in-S? (- n 3))))))
```

Here we have written a recursive procedure in Scheme that follows the definition. To determine whether $n \in S$, we first ask whether $n = 0$. If it is, then the answer is true. Otherwise we need to see whether $n - 3 \in S$. To do this, we first check to see whether $(n - 3) \geq 0$. If it is, then we can use our procedure to see whether it is in S.

Two alternative ways of writing the same definition of S:

Definition 2 (Example of a Bottom-Up Definition) *Define the set S to be the smallest set contained in N and satisfying the following two properties:*

1. $0 \in S$, *and*
2. *whenever $n \in S$, then $n + 3 \in S$.*

Definition 3 (Example of definition by rules of inference)

$$\frac{}{0 \in S}$$

$$\frac{n \in S}{(n+3) \in S}$$

- Each entry is called a *rule of inference*, or just a *rule*; the horizontal line is read as an "if-then".
- The part above the line is called the *hypothesis* or the *antecedent*; the part below the line is called the *conclusion* or the *consequent*.
- When there are two or more hypotheses listed, they are connected by an implicit "and."
- A rule with no hypotheses is called an *axiom*.
- The rules are interpreted as saying that an integer n is in S if and only if the statement "$n \in S$" can be derived from the axioms by using the rules of inference finitely many times.

These three definitions are entirely equivalent. We'll move back and forth between them freely.

Definition 4 (list of integers, top-down) *A Scheme list is a* list of integers *if and only if either*

1. *it is the empty list, or*
2. *it is a pair whose car is a integer and whose cdr is a list of integers.*

We use *Int* to denote the set of all integers, and *List-of-Int* to denote the set of lists of integers.

Definition 5 (list of integers, bottom-up) *The set List-of-Int is the smallest set of Scheme lists satisfying the following two properties:*

1. $() \in$ *List-of-Int, and*
2. *if* $n \in$ *Int and* $l \in$ *List-of-Int, then* $(n \ . \ l) \in$ *List-of-Int.*

The phrase $(n \ . \ l)$ denotes a Scheme list whose car is n and whose cdr is l.

Definition 6 (list of integers, rules of inference)

$$() \in \textit{List-of-Int}$$

$$\frac{n \in \textit{Int} \qquad l \in \textit{List-of-Int}}{(n \ . \ l) \in \textit{List-of-Int}}$$

1. `()` is a list of integers, because of property 1 of definition 5 or the first rule of definition 6.

2. `(14 . ())` is a list of integers, because of property 2 of definition 5, since `14` is a integer and `()` is a list of integers. We can also write this as an instance of the second rule for *List-of-Int* .

$$\frac{\texttt{14} \in \textit{Int} \qquad \texttt{()} \in \textit{List-of-Int}}{\texttt{(14 . ())} \in \textit{List-of-Int}}$$

3. `(3 . (14 . ()))` is a list of integers, because of property 2, since `3` is a integer and `(14 . ())` is a list of integers. We can write this as another instance of the second rule for *List-of-Int* .

$$\frac{\texttt{3} \in \textit{Int} \qquad \texttt{(14 . ())} \in \textit{List-of-Int}}{\texttt{(3 . (14 . ()))} \in \textit{List-of-Int}}$$

4. `(-7 . (3 . (14 . ())))` is a list of integers, because of property 2, since `-7` is a integer and `(3 . (14 . ()))` is a list of integers. Once more we can write this as an instance of the second rule for *List-of-Int* .

$$\frac{\texttt{-7} \in \textit{Int} \qquad \texttt{(3 . (14 . ()))} \in \textit{List-of-Int}}{\texttt{(-7 . (3 . (14 . ())))} \in \textit{List-of-Int}}$$

5. Nothing is a list of integers unless it is built in this fashion.

1.4 Defining Sets Using Grammars

Normally use grammars to define sets of strings, but can use them to define sets as well.

$$\begin{array}{l} \textit{List-of-Int} ::= \texttt{()} \\ \textit{List-of-Int} ::= \texttt{(}\textit{Int} \quad \texttt{.} \quad \textit{List-of-Int}\texttt{)} \end{array}$$

two rules, corresponding to the two properties in Definition 5 above.

In this definition we have

- **Nonterminal Symbols**. These are the names of the sets being defined. In this case there is only one such set, but in general, there might be several sets being defined. These sets are sometimes called *syntactic categories*.

  We will use the convention that nonterminals and sets have names that are capitalized, but we will use lower-case names when referring to their elements in prose. This is simpler than it sounds. For example, *Expression* is a nonterminal, but we will write $e \in \textit{Expression}$ or "*e* is an expression."

  Another common convention, called *Backus-Naur Form* or *BNF*, is to surround the word with angle brackets, *e.g.* ⟨expression⟩.

- **Terminal Symbols**. These are the characters in the external representation, in this case `.`, `(`, and `)`. We typically write these using a typewriter font, *e.g.* `lambda`.

- **Productions**. The rules are called *productions*. Each production has a left-hand side, which is a nonterminal symbol, and a right-hand side, which consists of terminal and nonterminal symbols. The left- and right-hand sides are usually separated by the symbol ::=, read *is* or *can be*. The right-hand side specifies a method for constructing members of the syntactic category in terms of other syntactic categories and *terminal symbols*, such as the left parenthesis, right parenthesis, and the period.

Shorthand notation in grammars:

Alternatives in a single line:

$$\textit{List-of-Int} \ ::= \texttt{()} \ \mid \texttt{(}\textit{Int} \ \texttt{.} \ \textit{List-of-Int}\texttt{)}$$

Omitting repeated LHS's:

$$\begin{aligned} \textit{List-of-Int} \ &::= \texttt{()} \\ &::= \texttt{(}\textit{Int} \ \texttt{.} \ \textit{List-of-Int}\texttt{)} \end{aligned}$$

Kleene star:

$$\textit{List-of-Int} \ ::= \texttt{(}\{\textit{Int}\}^{*}\texttt{)}$$

Separated Lists:

Comma-separated strings in $\{\textit{Int}\}^{*(,)}$:

```
                <---- the empty string is included
8
14, 12
7, 3, 14, 16
```

Semicolon-separated strings in $\{\textit{Int}\}^{*(;)}$:

```
                <---- the empty string is included
8
14; 12
7; 3; 14; 16
```

1.5 Defining Procedures Recursively

The Smaller-Subproblem Principle

If we can reduce a problem to a smaller subproblem, we can call the procedure that solves the subproblem to solve the problem.

Imagine we want to define a function to find powers, eg. $e(n,x) = x^n$ for any non-negative integer n. (We define $e(0,x) = 1$).

e : non-negative-int * int -> int
usage: (e n x)
produces: x^n

```
(define e
  (lambda (n x)
    (if (zero? n)
        1
        (* x
           (e (- n 1) x)))))
```

Let's watch this work. We can use good old-fashioned algebraic reasoning.

```
  (e 3 5)
= (* 5 (e 2 5))
= (* 5 (* 5 (e 1 5)))
= (* 5 (* 5 (* 5 (e 0 5))))
= (* 5 (* 5 (* 5 1)))
= 125
```

What's the moral? If we can reduce the problem to a smaller subproblem, then we can call the procedure itself ("recursively") to solve the smaller subproblem. Then, as we call the procedure, we ask it to work on smaller and smaller subproblems, so eventually we will ask it about something that it can solve directly (eg $n = 0$, the basis step), and then it will terminate successfully.

1.6 Manipulating lists of symbols

```
;; a list of symbols is either
;;     -- empty
;;     -- of the form (cons sym los), where sym is a symbol and los is
;;        another list of symbols.
```

1.6.1 remove-first

remove-first : sym * (list-of sym) -> (list-of sym)
usage: (remove-first sym los)
produces: a list like los, but without the first occurrence (if any) of sym.

```
;; examples:
;;   (remove-first 'a '()) = ()
;;   (remove-first 'a '(a b a c)) = (b a c)
;;   (remove-first 'a '(b a b a c)) = (b b a c)

(define remove-first
  (lambda (sym los)
    (if (null? los) '()
      (if (eqv? sym (car los))
        (cdr los)
        (cons (car los)
          (remove-first sym (cdr los)))))))
```

1.6.2 remove

remove : sym * (list-of sym) -> (list-of sym)
usage: (remove sym los)
produces: a list like los, but without all occurrences (if any)
of sym.

```
;; examples:
;;   (remove 'a '()) = ()
;;   (remove 'a '(a b a c)) = (b c)
;;   (remove 'a '(b a b a a c)) = (b b c)

(define remove
  (lambda (sym los)
    (if (null? los) '()
      (if (eqv? sym (car los))
        (remove sym (cdr los))
        (cons (car los)
          (remove sym (cdr los)))))))
```

1.7 Follow the Grammar!

Follow the Grammar!

When defining a procedure that operates on inductively-defined data, the structure of the program should be patterned after the structure of the data.

More precisely:

- Write one procedure for each nonterminal in the grammar. The procedure will be responsible for handling the data corresponding to that nonterminal.
- In each procedure, you will have one `cond` line for each production corresponding to that nonterminal. You may have additional case structure, but this will get you started on the right foot.

Procedure `subst`

Input: Two symbols: `new` and `old`, and an list `slst`

Output: a list similar to `slst` but with all occurrences of `old` replaced by instances of `new`.

Example:

```
> (subst 'a 'b '((b c) (b () d)))
((a c) (a () d))
```

But wait: need to be more precise about what kind of lists are possible inputs. Let's write a grammar to describe these nested lists of symbols:

Definition 7 (s-list, s-exp)

$$\begin{aligned} \textit{S-list} &::= (\{\textit{S-exp}\}^* \ ) \\ \textit{S-exp} &::= \textit{Symbol} \mid \textit{S-list} \end{aligned}$$

An s-list is a list of s-exps, and an s-exp is either an s-list or a symbol. Here are some s-lists:

```
(a b c)
(an (((s-list)) (with () lots) ((of) nesting)))
```

The Kleene star gives a concise description of the set of S-lists, but it is not so helpful for writing programs.

So rewrite the grammar to eliminate the use of the Kleene star.

$$
\begin{aligned}
\textit{S-list} &::= \texttt{()} \\
&::= (\textit{S-exp} \ . \ \textit{S-list}) \\
\textit{S-exp} &::= \textit{Symbol} \mid \textit{S-list}
\end{aligned}
$$

The follow-the-grammar pattern says we should have two procedures, one for dealing with *S-list* and one for dealing with *S-exp*:

subst `: sym * sym * s-list -> s-list`
usage: `(subst new old slist)`
produces: `a copy of slist with all occurrences of old replaced by new`

```
(define subst
  (lambda (new old slist)
    ...))
```

subst-in-s-exp `: sym * sym * s-exp -> s-exp`
usage: `(subst-in-sexp new old sexp)`
produces: `a copy of sexp with all occurrences of old replaced by new`

```
(define subst-in-s-exp
  (lambda (new old sexp)
    ...))
```

Let us first work on `subst`. The grammar tells us there should be two alternatives:

```
subst : sym * sym * s-list -> s-list
(define subst
  (lambda (new old slist)
    (if (null? slist)                ; is slist empty?
      ...                            ; answer for empty slist
      ...)))                         ; answer for non-empty slist
```

If the list is empty, there are no occurrences of `old` to replace.

```
subst : sym * sym * s-list -> s-list
(define subst
  (lambda (new old slist)
    (if (null? slist)
      '()
      ...)))
```

If `slist` is non-empty, its car is an *S-exp*, and its cdr is an *S-list*. So we recur on the car using `subst-in-s-exp`, and we recur on the cdr using `subst`.

```
subst : sym * sym * s-list -> s-list
(define subst
  (lambda (new old slist)
    (if (null? slist)
      '()
      (cons
        (subst-in-s-exp new old (car slist))
        (subst new old (cdr slist))))))
```

Now we can move on to `subst-in-s-exp`. From the grammar, we know that the symbol expression `sexp` is either a symbol or an s-list. If it is a symbol, we need to ask whether it is the same as the symbol `old`. If it is, the answer is `new`; if it is some other symbol, the answer is the same as `sexp`. If `sexp` is an s-list, then we can recur using `subst` to find the answer.

subst-in-s-exp `: sym * sym * s-exp -> s-exp`

```
(define subst-in-s-exp
  (lambda (new old sexp)
    (if (symbol? sexp)
      (if (eqv? sexp old) new sexp)
      (subst new old sexp))))
```

Since we have strictly followed the definition of *S-list* and *S-exp*, this recursion is guaranteed to halt. Since `subst` and `subst-in-s-exp` call each other recursively, we say they are *mutually recursive*.

The decomposition of `subst` into two procedures, one for each syntactic category, is an important technique. It allows us to think about one syntactic category at a time, which simplifies things tremendously in more complicated situations.

1.8 Auxiliary Procedures and Context Arguments

Sometimes you need to *generalize* the problem before you can solve it.

number-elements : (list-of scheme-value)
                      -> (list-of (list int val))
usage: (number-elements-from '(v_0 v_1 v_2 ...))
        = ((0 v_0) (1 v_1) (2 v_2) ...)

```
;; analysis: generalize the problem by considering
```

number-elements-from : (list-of scheme-value) * int
                      -> (list-of (list int val))
usage: (number-elements-from '(v_0 v_1 v_2 ...) n)
        = ((n v_0) ($n+1$ v_1) ($n+2$ v_2) ...)

```
;; example:
;; (number-elements-starting-from '(a b c) 3) = ((3 a) (4 b) (5 c))

;; analysis: a list is either empty, or the cons of a value and a list.

(define number-elements-starting-from
  (lambda (lst n)
    (if (null? lst) '()
      (cons
        (list n (car lst))
        (number-elements-starting-from (cdr lst) (+ 1 n))))))

;; now (number-elements lst) is just
;; (number-elements-starting-from lst 0).

(define number-elements
  (lambda (lst)
    (number-elements-starting-from lst 0)))
```

There are two important observations to be made here:

- `number-elements-starting-from` has an *independent* specification. It's not random! This gives us a slogan:

No Mysterious Auxiliaries!

When defining an auxiliary procedure, always specify what it does on all *arguments, not just the initial values.*

- The two arguments to `number-elements-starting-from` play two different roles.

  The first argument is the list we are working on. It gets smaller at every recursive call.

  The second argument is an abstraction of the *context* in which we are working. `number-elements-starting-from` always works on a sublist of the original list. The second argument represents the information that was in the original list but would now be invisible to us because we are looking at the sublist. In this case, the only information we need is the number of elements that are above the current sublist. This need not decrease at a recursive call; indeed it grows, because we are passing over another element of the original list. We sometimes call this a *context argument* or *inherited attribute*.

1.9 Tail Recursion

How could we talk about recursion without doing factorial?

fact `: int -> int`
usage: `(fact n)`
produces: `n!`

```
(define fact
  (lambda (n)
    (if (zero? n) 1 (* n (fact (- n 1))))))
```

We can model a calculation with `fact`:

```
  (fact 4)
= (* 4 (fact 3))
= (* 4 (* 3 (fact 2)))
= (* 4 (* 3 (* 2 (fact 1))))
= (* 4 (* 3 (* 2 (* 1 (fact 0)))))
= (* 4 (* 3 (* 2 (* 1 1))))
= (* 4 (* 3 (* 2 1)))
= (* 4 (* 3 2))
= (* 4 6)
= 24
```

This is the natural recursive definition of factorial. Each call of `fact` is made with a promise that the value returned will be multiplied by the value of `n` at the time of the call. Thus `fact` is invoked in larger and larger control contexts as the calculation proceeds.

Compare this behavior with that of the following procedures.

fact-iter `: int -> int`
usage: `(fact-iter n)`
produces: `n!`

```
(define fact-iter
  (lambda (n)
    (fact-iter-acc n 1)))
```

fact-iter-acc `: int * int -> int`
usage: `(fact-iter-acc n a)`
produces: `n! * a`

```
(define fact-iter-acc
  (lambda (n a)
    (if (zero? n) a (fact-iter-acc (- n 1) (* n a)))))
```

With these definitions, we calculate:

```
  (fact-iter 4)


= (fact-iter-acc 4  1)


= (fact-iter-acc 3  4)


= (fact-iter-acc 2  12)


= (fact-iter-acc 1  24)


= (fact-iter-acc 0  24)


= 24
```

In keeping with our preceding slogan ("No Mysterious Auxiliaries!"), notice that `fact-iter-acc` has its own purpose statement: it calculates $n! \times a$. This is easy to check, and also justifies the initial call from `fact-iter` to `fact-iter-acc`. It also makes `fact-iter-acc` a generalization of `fact`, just as `number-elements-starting-from` was a generalization of `number-elements`.

But we digress....

Here, `fact-iter-acc` is always invoked in the same context (in this case, no context at all). When `fact-iter-acc` calls itself, it does so at the "tail end" of a call to `fact-iter-acc`. That is, no promise is made to do anything with the returned value other than return it as the result of the call to `fact-iter-acc`. This is called *tail recursion*.

Thus each step in the derivation above has the form (`fact-iter-acc` n a). Because the program is tail recursive, it behaves just like a flowchart written in Scheme: no control stack is necessary, even though we are doing almost nothing but procedure calls.

1.10 Testing

You've always gotta have a test plan! The software to set up and run the test is sometimes called the "test scaffolding" or the "test harness". The test harness is in `drscheme-init.scm`. Here's how to use the test harness:

```
;; mp1-soln.scm

(module mp1-soln (lib "eopl.ss" "eopl")

  (provide subst)

  ;;;;;;;; Solution ;;;;;;;;;;;;;;

  ;; data definitions:
  ;;  slist ::= () OR (cons sexp slist)     -- ie, (list-of sexp).
  ;;  sexp    = symbol OR slist

  ;; subst : slist * symbol * slist -> slist
  ;; usage : (subst new old slist)
  ;; produces : a copy of slist with all occurrences of old
  ;; replaced by new.

  (define subst
    (lambda (new old slist)
      (cond
        [(null? slist)   ; is the list empty?
         '()]            ; yes: nothing to substitute
        [(pair? slist)   ; no: work on the pieces
         (cons
          (subst-in-sexp new old (car slist))
          (subst new old (cdr slist)))])))

  ;; ... more procedures, with purpose headers...

)
```

```
;; top.scm
(module top  (lib "eopl.ss" "eopl")

  (require "drscheme-init.scm")
  (require "mp1-soln.scm")

  ;; test-subst! : sym * sym * sym * s-list * s-list -> unspecified
  ;; prints either "test-name passed" or "test-name FAILED".
  ;; uses (run-experiment fn args correct-answer equal-answer?)
  (define test-subst!
    (lambda (test-name new old slist ans)
      (eopl:printf "~s " test-name)
      (let ((outcome
                  (run-experiment subst (list new old slist) ans
                                  equal?)))
        (if (car outcome)
          (eopl:printf "passed ~%")
          (eopl:printf "FAILED~%")))))

  (test-subst! 'subst-test-1 'a 'b '((b c) (b () d))
    '((a c) (a () d)))

  (test-subst! 'subst-test-2 'a 'b '() '())

  (test-subst! 'subst-test-3 'a 'b '(c b d a) '(c a d a))

  ;; ...more like this

  )
```

Here's what happens when we run `top.scm`:

```
Welcome to DrScheme, version 352.
Language: (module ...).
drscheme-init.scm plt209.1.5 10feb2005
subst-test-1 passed
subst-test-2 passed
subst-test-3 passed
>
```

If any of these failed, you could go back and call `subst` again to see what it did. Or you could modify `test-subst!` to print out more information when a test fails.

The basic testing procedure provided by `test-utils.scm` is the procedure `run-experiment`.

```
;; run-experiment :  procedure * (list-of scheme-value)
                     * scheme-value * (scheme-value * scheme-value -> bool)
                     -> (cons bool b)

;; usage: (run-experiment fn args correct-answer equal-answer?)
;; Applies fn to args.  Compares the result to correct-answer using
;; equal-answer?
;; Returns a pair consisting of a boolean (#t if the answer is
;; correct, otherwise #f) and the result.
```

When we test interpreters, we'll have a slightly more elaborate setup, in which the solution and tests are spread out over several modules.

Note that the three examples above test every branch in our procedure.

These files are at

`http://www.ccs.neu.edu/course/csg111/interps/lecture1/`

All our code will be in subfolders of `interps`.

1.11 Trees

[Note: This is based on Section 2.4 in EOPL3.]

We will do a lot a manipulation of trees. For this we've created some extensions of Scheme. For PLT Scheme, these are loaded by selecting the EOPL language level. To see how this is done, consider a definition of binary trees

$$\textit{Bintree} ::= \textit{Int} \mid (\textit{Symbol}\ \ \textit{Bintree}\ \ \textit{Bintree})$$

But we would like to manipulate trees *independent* of their representation as Scheme data structures. For this we need an interface. What should the interface look like? We'll need:

- *constructors* that allow us to build each kind of binary tree,
- a predicate that tests to see if a value is a representation of a binary tree, and
- some way of determining, given a binary tree, whether it is a leaf or an interior node, and of extracting its components.

We create this interface using `define-datatype`:

```
(define-datatype bintree bintree?
  (leaf-node
    (datum number?))
  (interior-node
    (key symbol?)
    (left bintree?)
    (right bintree?)))
```

This says that a `bintree` is either

- a `leaf-node` consisting of a number called the `datum` of the `bintree` or
- an `interior-node` consisting of a `key` that is a symbol, a `left` that is a `bintree`, and a `right` that is also a `bintree`.

It creates a data type with the following interface:

- a procedure, `leaf-node`, for constructing a `leaf-node`. The procedure `leaf-node` tests its argument with `number?`; if the argument does not pass this test, an error is reported.
- a 3-argument procedure, `interior-node`, for building an `interior-node`. This procedure tests its first argument with `symbol?` and its second and third arguments with `bintree?` to ensure that they are appropriate values.
- a 1-argument predicate `bintree?` that when passed a `leaf-node` or an `interior-node` returns true. For all other arguments, it returns false.

General form of `define-datatype`:

`(define-datatype` *type-name type-predicate-name*
  {`(`*variant-name* {`(`*field-name predicate*`)`}*`)`}$^+$ `)`

This creates a variant-record data type, named *type-name*. There are one or more *variants*, and every value of the data type is a value of exactly one of the variants. Each variant has a *variant-name* and zero or more fields, each with its own *field-name* and associated *predicate*.

Having zero fields is useful:

```
(define-datatype maybe-int maybe-int?
  (absent)
  (present (contents integer?)))
```

In languages like C or Java, a variable of type `t` can be either null or a "real" `t`, so these languages confuse `t` and `(maybe-t)`. Then you have to remember when you have to check for `null` and when can get away without it. We'll use constructions like this to keep them nice and separate.

In PLT Scheme, trees are represented using a PLT Scheme data type called a *structure*:

```
> (define tree-1 (leaf-node 3))
> tree-1
#(struct:leaf-node 3)
> (define tree-2
    (interior-node 'key1 tree-1 (leaf-node 4)))
> tree-2
#(struct:interior-node
   key1
   #(struct:leaf-node 3)
   #(struct:leaf-node 4))
> (define tree-4 (interior-node 'key2 tree-2 tree-1))
> tree-4
#(struct:interior-node
  key2
  #(struct:interior-node key1
     #(struct:leaf-node 3)
     #(struct:leaf-node 4))
  #(struct:leaf-node 3))
>
```

However, we said we wanted an interface that would allow us to manipulate trees *independent* of their representation as Scheme data structures. This means that you may *not* rely on this representation in your code. You may not use the PLT selector functions to get at the components. (Indeed, you cannot: we've carefully designed the system that way.)

Instead, we will decompose trees using the special form `cases`. All of the knowledge about the representation of trees is encapsulated inside `define-datatype` and `cases`, so if we ever decide to change the representation of trees, we only need to change these two special forms.

`cases` works by doing very simple pattern-matching on trees. Here's an example:

```
;; leaf-sum : bintree -> number
;; usage: (leaf-sum tree)
;; produces: the sum of the leaf node values in tree

(define leaf-sum
  (lambda (tree)
    (cases bintree tree
      (leaf-node (datum) datum)
      (interior-node (key left right)
        (+ (leaf-sum left)
           (leaf-sum right))))))
```

`cases` branches on `tree` to see which variant it belongs to, and takes the corresponding branch. Then each of the variables in the branch is bound to the corresponding field of `tree`, and the expression in the branch is evaluated. Thus, if `tree` were bound to

`(interior-node key17 (leaf-node 3) (leaf-node 4))`,

then

1. The `interior-node` branch would be selected,
2. `left` would be bound to `(leaf-node 3)` and `right` would be bound to `(leaf-node 4)`, and
3. the expression `(+ (leaf-sum left) (leaf-sum right))` would be evaluated.

We will write lots of code like this.

`cases` binds its variables *positionally*: the *i*-th variable is bound to the value in the *i*-th field. So we could just as well have written:

```
;; leaf-sum : bintree -> number
;; usage: (leaf-sum tree)
;; produces: the sum of the leaf node values in tree

(define leaf-sum
  (lambda (tree)
    (cases bintree tree
      (leaf-node (n) n)
      (interior-node (k l r)
        (+ (leaf-sum l)
           (leaf-sum r))))))
```

The general form of a `cases` expression:

`(cases` *type-name expression*
  {(*variant-name* ({*field-name*}*) *consequent*)}*
  `(else` *default*`))`

A `cases`-clause may have more than one expression; the expressions are evaluated left to right, but only the value of the last one is returned. This is handy for inserting tracing printouts, etc.

The final clause may be an `else` clause, like the `else` clause in a `cond`. However, we typically do not use `else` clauses. That way if we leave something out, we'll get an error immediately.

```
;; double-tree : bintree -> bintree
;; usage: (double-tree tree)
;; produces: a bintree just like the original, but with all the numbers
;; in the leaf nodes doubled.

(define double-tree
  (lambda (tree)
    (cases bintree tree
      (leaf-node (n)
        (leaf-node
          (* 2 n)))
      (interior-node (key left right)
        (interior-node
          key
          (double-tree left)
          (double-tree right))))))
```

Note how this procedure follows the grammar for trees!

Note the distinction between the nodes names and the procedures that construct them. We could write

```
(define foo
  (lambda (n) (leaf-node n)))

(define double-tree
  (lambda (tree)
    (cases bintree tree
      (leaf-node (n)
        (foo
          (* 2 n)))
      ...)))
```

but not

```
(define foo
  (lambda (n) (leaf-node n)))

(define double-tree
  (lambda (tree)
    (cases bintree tree
      (foo (n)    ;; this must be leaf-node, not foo.
        (leaf-node
          (* 2 n)))
      ...)))
```

number-leaves : bintree -> bintree
usage: (number-leaves t)
produces: a tree like the original, but in which all leaves are
numbered starting from 0

```
(define number-leaves
  (lambda (t)
    (car
      (number-leaves-starting-from t 0))))
```

number-leaves-starting-from : bintree * int -> bintree * int
usage: (number-leaves t n)
produces: a pair whose car is a tree like t, but with leaves numbered
starting from n, and whose cdr is the first ‘‘unused’’ node number

```
(define number-leaves-starting-from
  (lambda (t n)
    (cases bintree t
      (leaf-node (d)
        (cons
          (leaf-node n)
          (+ n 1)))
      (interior-node (key left right)
        (let ((ans1 (number-leaves-starting-from left n)))
          (let ((ans2 (number-leaves-starting-from right (cdr ans1))))
            (cons
              (interior-node
                key
                (car ans1)
                (car ans2))
              (cdr ans2)))))))))
```

Note the use of `let` to avoid retraversing the subtrees. [Puzzle: this algorithm runs in time proportional to the size of the input tree. If we didn’t use `let`, so that each subtree were traversed twice, what would the running time be?]

1.12 A Larger Example

Consider a set of trees given by the following grammar:

Red-blue-tree ::= *Red-blue-subtree*
Red-blue-subtree ::= `(red-node` *Red-blue-subtree* *Red-blue-subtree*`)`
::= `(blue-node` $\{$*Red-blue-subtree*$\}^*$ `)`
::= `(leaf-node` *integer*`)`

We wish to write a procedure that takes a tree and builds a tree of the same shape, except that each leaf node is replaced by a leaf node that contains the number of *red* nodes above it in the tree.

Why did we make *Red-blue-tree* a separate non-terminal? Because the top of a tree never has any nodes above it (especially no red nodes!), so it will be treated specially.

```
(define tree1
  (a-tree               ; these are procedure calls, not quoted lists!
    (red-node
      (blue-node
        (list (leaf-node 3)
              (leaf-node 4)
              (leaf-node 5)))
      (red-node
        (leaf-node 6)
        (red-node
          (leaf-node 7)
          (leaf-node 8))))))

> tree1
#(struct:a-tree
  #(struct:red-node
    #(struct:blue-node
      (#(struct:leaf-node 3)
       #(struct:leaf-node 4)
       #(struct:leaf-node 5)))
    #(struct:red-node
      #(struct:leaf-node 6)
      #(struct:red-node
        #(struct:leaf-node 7)
        #(struct:leaf-node 8)))))

> (mark-leaves-with-red-depth tree1)
#(struct:a-tree
  #(struct:red-node
    #(struct:blue-node
      (#(struct:leaf-node 1)
       #(struct:leaf-node 1)
       #(struct:leaf-node 1)))
    #(struct:red-node
      #(struct:leaf-node 2)
      #(struct:red-node
        #(struct:leaf-node 3)
        #(struct:leaf-node 3)))))
```

This grammar has three non-terminals: *tree*, *subtree*, and *list of subtrees*, so we follow the grammar and write three procedures, `mark-leaves-with-red-depth`, `mlwrd-subtree` (`mark-leaves-with-red-depth-subtree` is too long a procedure name, even for me) and `mlwrd-list-of-subtrees`.

```
(define-datatype red-blue-tree red-blue-tree?
  (a-tree
    (root red-blue-subtree?)))

(define-datatype red-blue-subtree red-blue-subtree?
  (red-node
    (lson red-blue-subtree?)
    (rson red-blue-subtree?))
  (blue-node
    (sons (list-of red-blue-subtree?)))
  (leaf-node
    (datum number?)))
```

We could have defined a separate `datatype` for *list of subtrees*, instead we represent these as Scheme lists, and use the predicate constructor `list-of`:

```
;; list-of : predicate -> predicate
(define list-of
  (lambda (pred)
    (lambda (val)
      (or (null? val)
        (and (pair? val)
          (pred (car val))
          ((list-of pred) (cdr val)))))))
```

This is included in the EOPL language level, so you don't need to define it.

mark-leaves-with-red-depth : red-blue-tree -> red-blue-tree
usage: (mark-leaves-with-red-depth tree)
produces: a red-blue-tree like tree, but with each leaf node
replaced by a leaf node containing the number of red nodes
above it.

```
(define mark-leaves-with-red-depth
  (lambda (tree)
    (cases red-blue-tree tree
      (a-tree (root)
        (a-tree
          (mlwrd-red-blue-subtree root 0))))))
```

mlwrd-red-blue-subtree : red-blue-subtree * int
-> red-blue-tree-subtree
usage: (mlwrd-red-blue-subtree subtree n), where n is the
number of red nodes above this subtree in the entire tree.
produces: a red-blue-subtree like this subtree,
but with each leaf node replaced by a leaf node
containing the number of red nodes above it.

```
(define mlwrd-red-blue-subtree
  (lambda (subtree n)                ; n is the number of red nodes
                                     ; above the subtree
    (cases red-blue-subtree subtree
      (leaf-node (d) (leaf-node n))
      (red-node (lson rson)
        (red-node
          (mlwrd-red-blue-subtree lson (+ n 1)) ; one more red node
          (mlwrd-red-blue-subtree rson (+ n 1))))
      (blue-node (sons)
        (blue-node
          (mlwrd-list-of-red-blue-subtrees sons n))) ; no more red nodes
      )))
```

mlwrd-list-of-red-blue-subtrees
: (list-of red-blue-subtree) * int -> (list-of red-blue-subtree)
usage: (mlwrd-list-of-red-blue-subtrees sons n)
produces: a list of subtrees at ‘‘red-depth’’ n with each leaf node replaced by a leaf node containing the number of red nodes above it

```
(define mlwrd-list-of-red-blue-subtrees
  (lambda (subtrees n)                ; n still the number of red
                                      ; nodes above these trees
    (if (null? subtrees) '()
      (cons (mlwrd-red-blue-subtree (car subtrees) n)
        (mlwrd-list-of-red-blue-subtrees (cdr subtrees) n)))))
```

1.13 letrec

Lots of times we'd like to define recursive procedures locally, like `fact-iter` above.

fact-iter `: int -> int`
usage: `(fact-iter n)`
produces: `n!`

```
(define fact-iter
  (lambda (n)
    (fact-iter-acc n 1)))
```

fact-iter-acc `: int * int -> int`
usage: `(fact-iter-acc n a)`
produces: `n! * a`

```
(define fact-iter-acc
  (lambda (n a)
    (if (zero? n) a (fact-iter-acc (- n 1) (* n a)))))
```

You'd like to write

```
(define fact-iter
  (lambda (n)
    (let ((fact-iter-acc
            (lambda (n a)
              (if (zero? n) a (fact-iter-acc (- n 1) (* n a))))))
      (fact-iter-acc n 1))))
```

But we can't do that, because the scope of `fact-iter-acc` doesn't include it's definition. Instead, we use `letrec`:

```
(letrec
 ((name1 proc1)
  (name2 proc2)
  ...)
 body)
```

creates a set of mutually recursive procedures and makes their names available in the body.

So we can write:

fact-iter `: int -> int`
usage: `(fact-iter n)`
produces: `n!`

```
(define fact-iter
  (lambda (n)
    (letrec ((fact-iter-acc   ; : int * int -> int
               (lambda (n a)  ; produces: n! * a
                 (if (zero? n) a
                   (fact-iter-acc (- n 1) (* n a))))))
      (fact-iter-acc n 1))))
```

Note how we've put in the documentation for the local procedure. Just because it's local doesn't mean you don't have to document it!

The most common use of letrec is to define a single procedure and then start it. For this there is a special form called *named let*. This overloads the ordinary `let` form. For example,

```
(define fact-iter
  (lambda (n)
    (letrec ((fact-iter-acc
               (lambda (n a)
                 (if (zero? n) a
                   (fact-iter-acc (- n 1) (* n a))))))
      (fact-iter-acc n 1))))
```

could be replaced by

```
(define fact-iter
  (lambda (n)
    (let fact-iter-acc        ; : int * int -> int
      ((n n) (a 1))           ; produces: n! * a
      (if (zero? n) a
        (fact-iter-acc (- n 1) (* n a))))))
```

`letrec` is handy because it allows us to think about recursion locally. It will be an important feature of the languages we define.

1.14 Sequencing

Want to have some finer control over the order in which things are done in Scheme. We need this for worrying about (a) side-effects and (b) termination. In general, there is only one rule about sequencing in Scheme:

Arguments are evaluated before procedure bodies

So in

```
((lambda (x y z) body) exp1 exp2 exp3)
```

`exp1`, `exp2`, and `exp3` are guaranteed to be evaluated before `body`, but we don't know in what order `exp1`, `exp2`, and `exp3` are going to be evaluated, but they will all be evaluated before `body`.

This is **precisely** the same as

```
(let ((x exp1) (y exp2) (z exp3))
  body)
```

In both cases, we evaluate `exp1`, `exp2`, and `exp3`, and then we evaluate `body` in an environment in which `x`, `y`, and `z` are bound to the values of `exp1`, `exp2`, and `exp3`.

Think about this. It is important.

We can use this to control the order of evaluation when that's important:

```
(begin exp1 exp2) = ((lambda (d) exp2) exp1)
                  = (let ((d exp1)) exp2)
```

where `d` is a variable that does not occur in `exp2`. (Think `d` for dummy).