An Introduction to Scheme

This handout is an introduction to programming in the Scheme programming language. Scheme is a block-structured, lexically-scoped, properly tail-recursive dialect of Lisp that supports first-class functions and continuations. (Don't worry if you don't know what all these terms mean yet -- you will by the end of the course.) It was invented by Gerald Jay Sussman and Guy Steele, Jr. at MIT in 1975, and has gained popularity as a teaching language and as an extension language (a language for expressing abstractions built on top of a given set of primitives).

1. Scheme = Kernel + Syntactic Sugar + Standard Library

The Scheme language is constructed out of three parts:

1. A small kernel language.
2. A collection of derived forms (a.k.a. **syntactic sugar**) rewritable into kernel constructs.
3. A library of standard procedures. (Unlike in other languages (such as Pascal), we will use the terms "procedure" and "function" interchangeably in the context of Scheme.)

For the most part, we will focus on the kernel language, but will mention syntactic sugar and library procedures as needed. For full details of the language, see *the Revised[5] Report on Scheme* (*R5RS*), which is available as Handout #6 and is also linked from the CS251 home page.

2. Syntax of the Scheme Kernel

The **syntax** of a programming language describes the *form* of phrases in the language. In contrast, the **semantics** of the language describes the *meaning* of these phrases. We describe the syntax of Scheme in this section and the semantics of Scheme in the next section.

The syntax of a language is often specified by a **grammar**. A grammar consists of

- a set of **terminals**, the primitive tokens out of which phrases are constructed.
- a set of **nonterminals**, each of which ranges over a class of compound phrases.
- a set of **productions** that specify the form of the phrases for each nonterminal.

2.1 Terminals

In the case of Scheme, the terminal tokens are as follows:

- *N* Numbers : e.g. `3`, `-17`, `3.141`, `6.023e23`
- *B* Booleans: `#t` (true) and `#f` (false)
- *Y* Symbols: `'a`, `'captain`, `'fib_n-2`
- *C* Characters: e.g. `#\a`, `#\b`, `#\c`, ... , `#\space`, `#\tab`, `#\newline`
- *R* Strings: e.g., `"A sample string"`, `"x"`, `""`
- *I* Identifiers: e.g., `x`, `square`, `fact_n-2`
- reserved words: `lambda`, `if`, `quote`, `letrec`, `begin`, `set!`, `define`, `let`, `cond`, `and`, `or`
- parentheses: `(`, `)`

We use the metavariables *N* , *B*, *Y*, *C*, *G* and *I* to range over numbers, booleans,

symbols, characters, strings, and identifiers, respectively. For example, we use N in contexts where we wish to refer to "any number" rather than a particular number.

2.2 Nonterminals

Nonterminals specify phrases that are essentially trees whose leaves are terminals. Nonterminals are designated by a metavariable that ranges over a class of phrases. The nonterminals for Scheme are:

- L Literals ("self-evaluating" phrases)
- E Expressions (phrases that denote a value)
- D Definitions (phrases that associate a name with a value)
- F Top-Level Forms (phrases understood by the interpreter; Scheme files are sequences of these phrases)

2.3 Productions

The form of the phrases for each nonterminal are specified by productions, that describe the tree structure of the phrases. Each production can be viewed as a rewrite rule. Any phrase consisting only of terminals that can be derived by a sequence of rewrite steps from a given nonterminal is in the syntactic class of that nonterminal.

The productions for the Scheme nonterminals are as follows:

```
L    N
L    B
L    Y
L    C
L    R

E    L                         ; Literal
E    I                         ; Variable Reference
E    (lambda (I1 ... In) E)    ; Abstraction
E    (E0 E1 ... En)            ; Application (Call)
E    (if E1 E2 E3)             ; Conditional

D    (define I E)

F    E
F    D
```

Note: for simplicity, a few of the productions for expressions have been omitted above. We will meet them later in the term.

2.4 Notes on Scheme Syntax

Terminology

Expressions that are terminals (literals and identifiers) are **primitive expressions**. Expressions that are not terminals are **compound expressions**. In order to be able to talk about compound expressions, it is important to know their names and the names of their subparts.

- A compound expression beginning with the reserved word **`lambda`**, as in `(lambda (`I_1 `...` I_n`)` E`)`, is an **abstraction**. The identifiers I_1 ... I_n are the **formal parameters** (or just **formals**), and E is the **body**.

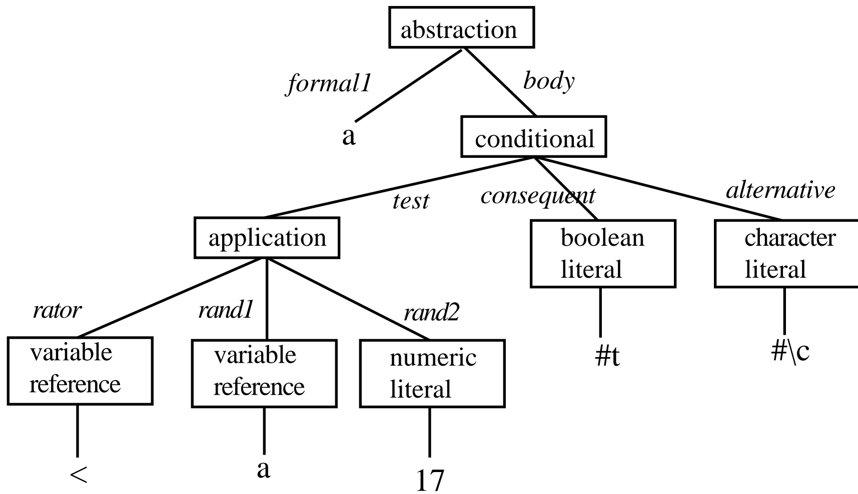
- A compound expression that does not begin with a reserved word, as in `(E0 E1 ... En)`, is an **application**. E_0 is the **operator** (or just **rator**) and $E_1 \ ... \ E_n$ are the **operands** (or just **rands**).

- A compound expression beginning with the reserved word **`if`**, such as `(if E1 E2 E3)`, is a **conditional**. E_1 is the **test** , E_2 is the **consequent** (or **then expression**), and E_3 is the **alternative** (or **else expression**).

A compound expression beginning with a reserved word is called a **special form**. For example, conditionals and abstractions are special forms. Applications are the only compound expressions that are not special forms. If you look at the list of reserved words in Section 2.1, you will see that certain special forms have not yet been introduced. Some will be introduced later in this document; others will be introduced later in the term.

Parenthesized Expressions Stand for Trees

We will see later in the term that a compound phrases in any language can be viewed as a tree (a so-called **abstract syntax tree**) whose leaves are terminals. The parenthesized form of compound expressions in Scheme emphasizes the tree-like nature of compound expressions. A parenthesized expression with *n* subexpressions represents a tree node with *n* subtrees. For example, the expression `(lambda (a) (if (< a 17) #t #\b))` is a sequence of characters that represents the following tree:



You should get into the habit of thinking of expressions as trees. We'll make heavy use of this notion later in the course.

Note that parentheses in Scheme are essential for indicating the tree structure of an expression. Unlike in most other languages, where parentheses are often an optional means of specifying grouping and precedence, parentheses in Scheme are never optional; each one has a very particular meaning!

Definitions

It is worth emphasizing that definitions are *not* expressions and cannot be used in the subexpression positions of a compound expression. For example, the following phrase is not legal in Scheme: `(if (< a 17) (define b 1) (define c 2))`. Definitions can only appear at "top-level" -- i.e., submitted directly to the interpreter or included as one of the top-level phrases of a file.

Strings vs. Symbols

Both strings and symbols are forms of alphabetic data. A string literal is delimited by double quotes, and may contain any characters, including whitespace characters (spaces, tabs, newlines). A symbol literal, on the other hand, is introduced by a single quote but does not have a corresponding closing quote; it consists of a sequence of non-whitespace characters.

From the abstract data type perspective, strings are more versatile than symbols because strings support numerous operations for manipulating the component characters of a string. On the other hand, symbols are treated as atomic (indivisible) entities; the only interesting thing you can do with a symbol is test it for equality with another symbol. Nevertheless, later on we will see that symbols are very convenient for specifying compound data structures.

Note: if `I` is an identifier, then `'I` and `(quote I)` are synonymous. Quotation plays an important role in the specification of compound data structures in Scheme; see Section 6.3 for details.

Case Sensitivity

Scheme is not case sensitive. This means that the identifiers `fact`, `Fact`, and `FACT` are all treated identically. The same holds true for symbols.

Comments

Comments are introduced in Scheme with a semicolon; the comment continues to the end of the line. In MIT Scheme, multi-line comments are opened with `#|` and are closed with `|#`. These are useful for commenting out blocks of code. Multi-line comments nest properly.

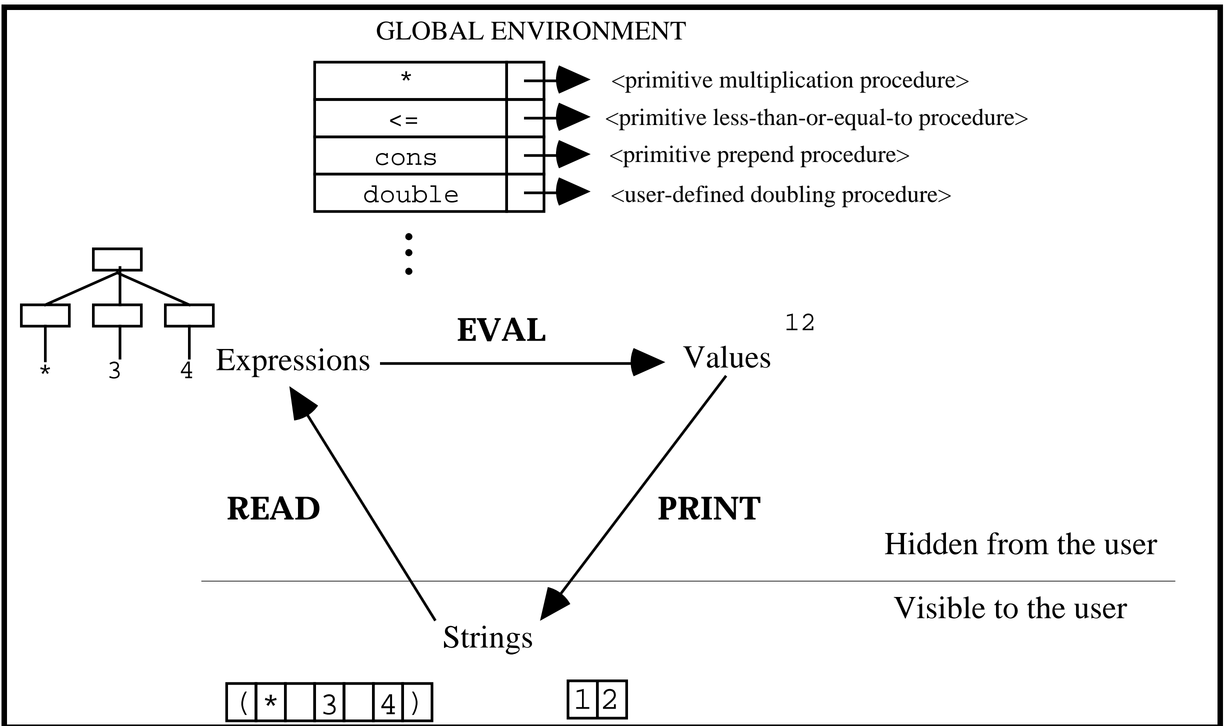
3. The Semantics of the Scheme Kernel

In this section, we explore the *meaning* of each of Scheme's syntactic phrases.

3.1 The Read-Eval-Print Loop

Scheme is an interpreted language, which means that its syntactic phrases can be directly executed by a virtual machine known as a Scheme interpreter. The interpreted nature of Scheme makes it straightforward to evaluate Scheme expressions and name Scheme values in an interactive manner. The entity with which you interact is known as a **read-eval-print loop**, or **REPL** for short.

In order to understand certain details about Scheme evaluation, it is important to have a good model of the REPL. Below is a diagram that depicts a Scheme REPL:



The user interacts with the REPL by typing in a string of characters that represents an expression. The system responds with a sequence of characters that represents the value that results from evaluating the given expression. So all the user sees when interacting with the REPL is character strings. We will represent such interactions as follows:

```
scheme> (* 3 4)
12
```

The top line contains a prompt (`scheme>`) that designates the name of the language we are interacting with. (This is helpful for distinguishing between the various languages we will be studying.) The prompt is followed by an expression to be evaluated, in this case `(* 3 4)`. The line following the end of the expression is the result printed by the interpreter, in this case `12`.

In order to have a firm understanding of this sort of interaction, it is important to have some idea of what is going on "under the hood". There are three phases to the REPL:

1. **Read**: In this phase, the character string representation of the expression is **parsed** into an abstract syntax tree.

2. **Eval**: In this phase, the tree representation of the expression is evaluated according to the evaluation rules presented later in this section.

3. **Print**: In this phase, the value resulting from the Eval phase is **unparsed** into a character string representation.

The first two phases are fairly uniform among Scheme implementations, but Scheme implementations sometimes differ in how they print out values. For example, some Scheme systems we will use this semester print the symbol `red` as `red` (without a quotation mark) and some will print it as `'red` (with a quotation mark). This is potentially rather confusing, but it helps to realize that the issue is just one of printed representation and not something more fundamental!

In order to evaluate most expressions, the Eval phase must know the meaning of global variable names, which are stored in a table called the **global environment**. When a REPL is first

created, this table is initialized with the standard bindings for Scheme: `*` is initially bound to the primitive multiplication procedure, `<=` with the primitive less-than-or-equal-to procedure, and so on. Whenever the user executes a definition -- i.e., a phrase of the form `(define I E)` -- the REPL evaluates *E* to a value *v*, and then associates the name *I* with *v* in the global environment. This way, the name is available in the evaluation of subsequent expressions.

3.2 Evaluation Rules (Substitution Model)

The semantics of Scheme is embodied in the Eval phase of the REPL. This phase determines the value of every expression according to a set of evaluation rules. This semester, we will study several different formulations of these evaluation rules. The formulation presented here, known the **substitution model**, is based on a notion of rewriting an expression according to a set of rewrite rules until it becomes a value. In this context, a value is defined as follows:

Definition: A **value** is an expression that is either:
- a literal;
- an abstraction; or
- a variable reference that is the name of a Scheme primitive procedure (e.g. `*`, `<=`).

Below, the evaluation rules are defined by case analysis on the type of expression being evaluated. We shall give examples of each rule using the `subst>` prompt to emphasize that we are using the substitution model. The `subst>` prompt will be followed by the entire sequence of rewritten expressions that lead to the final value.

Literals: A literal expression is self-evaluating, so it need not be rewritten: it is already a value.

```
subst> 17
17

subst> #t
#t

subst> #f
#f   ; Note: some versions of Scheme (e.g. MIT Scheme) do not distinguish
     ; falsity and the empty list. In such versions, #f prints as ().

subst> 'a
'a   ; Note: in most Scheme interpreters, this prints as a, not 'a.

subst> #\c
#\c

subst> "Hi there!"
"Hi there!"
```

Abstractions: Since abstraction expressions are values, they are self-evaluating:

```
subst> (lambda (x) (* x x))
(lambda (x) (* x x))
; Note Scheme interpreters differ greatly in terms of how they print
; procedure values. They usually do not print lambda notation. In MIT
; Scheme, user-defined procedures print as #[compound-procedure ZZZ],
; where ZZZ is an identifying number. System-defined procedures (e.g.,
; primitives such as +) have different printed representation

subst> (lambda (a b) (/ (+ a b) 2))
(lambda (a b) (/ (+ a b) 2))

subst> (lambda () 17)
(lambda () 17)
```

Variable References: A variable reference expression `I` is evaluated as follows:

1) If `I` is the name of a standard Scheme primitive procedure, it is self evaluating.
2) If `I` is the name of a user-defined value, it rewrites to the most recent value defined by the user.
3) If `I` is neither the name of a Scheme primitive procedure nor a user-defined value, the rewriting process halts with an error.

For example, suppose that the following session occurs after invoking a fresh REPL:

```
; In a new REPL, the only names bound in the global environment
; are the standard primitive procedure names of Scheme.

subst> +
+

; Extend the global environment with a new binding.
subst> (define double (lambda (x) (* x 2)))
double ; The "value" printed after a define is the defined name.

subst> double
(lambda (x) (* x 2))

subst> triple
Error! Unbound variable triple
```

Conditionals: A conditional expression `(if` E_1 E_2 E_3`)` is evaluated as follows:

1) Rewrite the test expression E_1 into a value V_1.
2) If V_1 is not `#f`, then the conditional expression rewrites to the then expression E_2.
3) If V_1 is `#f`, then the conditional expression rewrites to the else expression E_3.

```
subst> (if (< 1 2) (+ 3 4) (* 5 6))
(if #t (+ 3 4) (* 5 6))
(+ 3 4)
7

subst> (if (> 1 2) (+ 3 4) (* 5 6))
(if #f (+ 3 4) (* 5 6))
(* 5 6)
30

subst> (if (< 1 2) (+ 3 4) (5 * 6))
(if #t (+ 3 4) (5 * 6))
(+ 3 4)
7  ; Note: the error is never discovered in the branch not taken.

subst> (if 1 2 3)
; Note: any non-false value counts as true in a conditional test
```

Applications: An application expression `(E0 E1 ... En)` is evaluated as a two stage process:

1) In the **subexpression evaluation stage**, all subexpressions E_i are evaluated to their values V_i. In this stage, the operator subexpression is not treated any differently than the operand subexpressions. The fact that the operator position is an expression that is evaluated like any other is a source of great power in Scheme that we will witness when we study higher-order functions.
2) In the **application stage**, the operator value V_0 is applied to the operand values V_1 through V_n. The meaning of application depends on the what kind of value V_0 is:
   - If V_0 denotes a primitive Scheme procedure *P*, and there are an appropriate number of arguments of the appropriate type for *P*, the application is rewritten to the result of applying *P* to the arguments.
   - If V_0 is an abstraction, and the number of arguments matches the number of formal parameters, the application rewrites the application is rewritten to a copy of the body of the abstraction in which the argument values have been substituted for the corresponding formal parameters. (There are some tricky aspects to substitution that are described later.) The value of the application is determined by evaluating the copied body according to the substitution model.
   - In all other cases (i.e., V_0 is not a procedure, or the number or types of the arguments is not appropriate), the rewriting process halts with an error.

For example:

```
subst> (+ 1 2)
3

subst> (* (+ 1 2) (- 3 4))
(* 3 -1)
-3

subst> (not (< 3 2))
(not ())
#t

subst> (+ 1 (< 2 3))
(+ 1 #t)
Error! Can't add a boolean to a number.

subst> (not #t #f)
Error! Not expects exactly 1 argument, but was given 2.

subst> (2 + 3)
Error! 2 is not a procedure.

subst> ((lambda (x) (* x x)) (+ 2 3))
((lambda (x) (* x x)) 5) ; No substitution until arg is value
(* 5 5)
25

subst> (define average (lambda (a b) (/ (+ a b) 2)))
average

subst> (average (+ 2 4) (* (+ 1 2) 4))
((lambda (a b) (/ (+ a b) 2)) 6 (* 3 4))
((lambda (a b) (/ (+ a b) 2)) 6 12) ; No substitution until
(/ (+ 6 12) 2)                      ; both args are values
(/ 18 2)
9
```

```
subst> (define sum-of-squares   ; This method of formatting code
         (lambda (x y)          ; is called pretty-printing.
           (+ (square x)        ; Note that it's OK that square
              (square y))))     ; is not yet defined
sum-of-squares

subst> (define square (lambda (x) (* x x)))
square

subst> (sum-of-squares (+ 1 2) (/ 12 3))
((lambda (x y) (+ (square x) (square y))) 3 4)
(+ (square 3) (square 4))
(+ ((lambda (x) (* x x)) 3) ((lambda (x) (* x x)) 4))
(+ (* 3 3) (* 4 4))
(+ 9 16)
25

subst> (define add-a (lambda (x) (+ a x)))
add-a

subst> (define a 1)
a

subst> (add-a 3)
((lambda (x) (+ a x)) 3)
(+ a 3)
(+ 1 3)
4

subst> (define a 17) ; Overwrites previous definition of a

subst> (add-a 3)
((lambda (x) (+ a x)) 3)
(+ a 3)
(+ 17 3)
20

subst> ((+ 1 2) (* 3 4) (- 5 6))
(3 12 -1)
Error! 3 is not a procedure.

subst> ((lambda (x) (x x)) (lambda (x) (x x))
((lambda (x) (x x)) (lambda (x) (x x))
((lambda (x) (x x)) (lambda (x) (x x))
((lambda (x) (x x)) (lambda (x) (x x))
...  ; This shows that some rewriting processes may not terminate
```

3.3 Substitution

The substitution model gets its name from the substitution step that takes place in the application of an abstraction to argument values. Although substitution seems fairly intuitive, there are some pitfalls to watch out for:

- In cases where the same identifier names logically distinct variables, we must be careful to only substitute for the appropriate variable references. For example, in the expression,

```
((lambda (a) (if (< 3 a)
                 (lambda (b) (* a b))
                 (lambda (a) (* a a))))
 2)
```

there are really two distinct variables named a: the one declared by the outer lambda and the one declared by the inner lambda. Subscripting the formal parameters and their associated variable references would yield:

```
((lambda (a1) (if (< 3 a1)
                  (lambda (b) (* a1 b))
                  (lambda (a2) (* a2 a2))))
 2)
```

The result of substitution should therefore be:

```
(if (< 3 2) (lambda (b) (* 2 b)) (lambda (a) (* a a)))
```

That is, we should not substitute 2 for the as in `(lambda (a) (* a a))` because they stand for a different variable than the one we are substituting for. We shall formalize this notion when we study the scope of names in programs.

- We must be careful that Scheme primitive procedures passed as arguments are not accidentally capture by formal parameters that happen to have the same name. For example, in the following expression

  ```
  ((lambda (f) (lambda (+) f)) +)
  ```

  a naive substitution yields an identity function, which is incorrect:

  ```
  (lambda (+) +)
  ```

  The correct result is a procedure that always ignores its argument and returns +:

  ```
  (lambda (+_1) +)
  ```

  This result can be achieved by consistently renaming the formal parameter (and any associated variable references) of an abstraction when substituting a value into the abstraction that might accidentally be "captured". Again, we shall formalize this notion when we study the scope of names in programs.

3.4 A Substitution Model Interpreter

Normally, a Scheme REPL only shows you the final result of evaluating an expression. It does not show you the intermediate sequence of rewritten expressions.

I have developed an alternative Scheme REPL that displays all the intermediate rewritten expressions as implied by the substitution model. I shall refer to this REPL as the substitution model REPL. The substitution model REPL is a pedagogical tool that will help you to understand the details of the substitution model I encourage you to use the substitution model REPL at the beginning of the course whenever you have questions about the evaluation of an expression.

You are strongly encouraged to experiment with the substitution model interpreter, which is located in the CVS-controlled directory `cs251/subst-model`. Once you have created your local CS251 CVS repository (see Handout #10 for details), you can use the substitution model interpreter as follows:

1. Evaluate `(cd "~/cs251/subst-model")` in the Scheme REPL. This changes the default file directory to be the directory containing the implementation of the substitution model REPL.

2. Evaluate `(load "subst-model.scm")` in the Scheme REPL. This loads the definitions

that implement the substitution model REPL.

3. Evaluate `(subst-repl)` in the Scheme REPL. This will enter a new REPL whose prompt is `subst>` (to distinguish it from the normal Scheme REPL).

Usage notes for the substitution model REPL:

- As in normal Scheme, you can define values and load files containing definitions and expressions.
- If you evaluate an expression that signals an error, the error will exit the substitution model REPL and return to the normal Scheme REPL. To restart the substitution model REPL, evaluate `(subst-repl)` again. Unfortunately, all definitions from your previous session will have been lost. For this reason, it is a good idea to keep all your definitions in a file, so it is easy to reload them via `load`.
- To exit the substitution model REPL, evaluate the special token `exit`.

The substitution model REPL implements a design decision that is worth explaining. Where possible, it performs rewrite steps on the subexpressions of an application in parallel. For example, consider the first step the `average` example from above

```
subst> (average (+ 2 4) (* (+ 1 2) 4))
((lambda (a b) (/ (+ a b) 2)) 6 (* 3 4))
```

In one "step" , the interpreter has rewritten `average` to its associated value, `(+ 2 4)` to `6`, and `(* (+ 1 2) 4)` to `(* 3 4)`. The substitution model allows, but does not require, all of these rewrites to occur in the same step. There are many alternative strategies. For example, they could be performed one-by-one in a left-to-right manner:

```
subst> (average (+ 2 4) (* (+ 1 2) 4))
((lambda (a b) (/ (+ a b) 2)) (+ 2 4) (* (+ 1 2) 4)) ; Rewrite average
((lambda (a b) (/ (+ a b) 2)) 6 (* (+ 1 2) 4))       ; Rewrite (+ 2 4)
((lambda (a b) (/ (+ a b) 2)) 6 (* 3 4))             ; Rewrite (+ 1 2)
((lambda (a b) (/ (+ a b) 2)) 6 12)                  ; Rewrite (* 3 4)
```

Another approach would be a right-to-left order:

```
subst> (average (+ 2 4) (* (+ 1 2) 4))
(average (+ 2 4) (* (+ 1 2) (* 3 4)))                ; Rewrite (+ 1 2)
(average (+ 2 4) 12)                                 ; Rewrite (* 3 4)
(average 6 12)                                       ; Rewrite (+ 2 4)
((lambda (a b) (/ (+ a b) 2)) 6 12)                  ; Rewrite average
```

Many other orders of evaluation are possible. It turns out that the particular order taken cannot affect the final result in the purely functional subset of Scheme (i.e., the subset in which no side-effects are allowed).

3.5 Procedure Tracing

The detailed step-by-step information provided by the substitution model interpreter can be overwhelming for all but the simplest examples. Sometimes you just want to focus on the invocation of a small number of procedures. The procedure tracing facility of MIT Scheme is extremely handy for this purpose. It allows you to select any number of procedures you wish to "trace", and then prints out information about these procedures each time they are called or return. See Section 4.3 of the MIT Scheme handout (#7) or Section 5.4 of *the MIT Scheme User's Manual* for more information.

4. Standard Library and Syntactic Sugar and

The Scheme grammar defined in Section 2 is extremely small and simple. In fact, it seems *too* small. Is it really enough to support the writing of arbitrarily complex programs? The answer is yes -- as long as these kernel language constructs are used in conjunction with a suitable library of standard functions and some convenient syntactic sugar.

4. 1 The Scheme Standard Library

Scheme comes equipped with a number of dataypes and operations on those datatypes. One way to extend the language is to add a new datatype with its associated operations;
for these kinds of extensions the kernel remains unmodified. Following are some of the standard datatypes and their common operations; for more examples, see section 6 of *R4RS*.

Numbers: `+, -, *, /, max, min, quotient, remainder, <, <=, =, >, >= ...`
Booleans: `not`
Symbols: `eq?, symbol->string, string->symbol`
Characters: `char<?, char=?, char>?, char->integer, integer->char`
Strings: `string<?, string=?, string-ref, , ...`
Procedures: `apply`
Pairs: `cons, car, cdr, ...`
Lists: `list, list-ref, list-tail, append, reverse, map, ...`
Vectors: `vector, vector-ref, ...`

For each standard datatype, there is a predicate that tests to see if an element is of that datatype -- e.g., `number?, boolean?, symbol?, string?, procedure?, pair?, list?, null?, vector?, ...`

Most dataypes are equipped with an equality predicate that tests whether two elements of the datatype are the same. Here are the standard equality predicates for the atomic (i.e., not compound) datatypes:

Numbers: `=`
Booleans: `eq?`
Symbols: `eq?`
Characters: `char=?`
Strings: `string=?`

Rather than remember the above, you can just use the `eqv?` procedure, which dispatches to the appropriate equalilty testing procedure depending on the type of its argument.

Equality testing is trickier for compound data structures (pairs, lists, vectors) and procedures. For now, we will assume that two compound data structures are the same if they have the same shape and all their corresponding atomic components are pairwise `eqv?` This sort of equality can be tested with the `equal?` procedure.

As for procedures, what does equality on procedures mean? Ideally we would like what is called an "extensional equality": two procedures are equal if they return the same results on all inputs. Unfortunately, such an equality test is uncomputable; it is a prime example of a mathematical function that is well-defined, but for which it is impossible to write a program. A much easier equality to test is whether two procedures were created with the evaluation of the same `lambda` expression at the same point in time; this can be tested with `eq?`

4.2 Scheme Syntactic Sugar

Syntactic sugar specifies new special forms via rewrites into the kernel language. This allows the language to be extended with new special forms without affecting its essential core. Below are

some of the most important desugarings; for more, see Section 7.3 of *R5RS*. Note that it is possible to write any Scheme program without using any syntactic sugar at all. However, syntactic sugar helps to makes programs more concise and more readable.

```
(define (I_name I_param-1 ... I_param-n) E)
   desugars-to (define I_name (lambda (I_param-1 ... I_param-n) E))

(let ((I_name-1 E_def-1) ... (I_name-n E_def-n)) E_body )
   desugars-to ((lambda (I_name-1  ...I_name-n) E_body ) E_def-1 ...E_def-n)

(cond) desugars-to  <unspecified>
(cond (else E)) desugars-to  E
(cond (E_test E_action) ...)
   desugars-to (if E_test  E_action (cond ...))

(and) desugars-to  #t
(and E) desugars-to E
(and E_1 E_2 …) desugars-to  (if E_1  (and E_2 …) #f)

(or) desugars-to  #f
(or E) desugars-to E
(or E_1 E_2 …) desugars-to  (let ((I E_1 )) (if I I (or E_2 …)),
```

where `I` is a "fresh" identifier (i.e., not mentioned elsewhere in the program.

5. Recursion and Iteration

It is straightforward to express recursive definitions in Scheme. For example, here is a recursive definition in Scheme of the classic factorial function:

```
(define fact-rec
  (lambda (n)
    (if (= n 0)
        1
        (* n (fact-rec (- n 1))))))
```

The substitution model explains the evaluation of calls to `fact-rec` without a hitch. For example:

```
subst> (fact-rec 3)
(fact-rec 3)
((lambda (n) (if (= n 0) 1 (* n (fact-rec (- n 1))))) 3)
(if (= 3 0) 1 (* 3 (fact-rec (- 3 1))))
(if #f 1 (* 3 (fact-rec (- 3 1))))
(* 3 (fact-rec (- 3 1)))
(* 3 ((lambda (n) (if (= n 0) 1 (* n (fact-rec (- n 1))))) 2))
(* 3 (if (= 2 0) 1 (* 2 (fact-rec (- 2 1)))))
(* 3 (if #f 1 (* 2 (fact-rec (- 2 1)))))
(* 3 (* 2 (fact-rec (- 2 1))))
(* 3 (* 2 ((lambda (n) (if (= n 0) 1 (* n (fact-rec (- n 1))))) 1)))
(* 3 (* 2 (if (= 1 0) 1 (* 1 (fact-rec (- 1 1))))))
(* 3 (* 2 (if #f 1 (* 1 (fact-rec (- 1 1))))))
(* 3 (* 2 (* 1 (fact-rec (- 1 1)))))
(* 3 (* 2 (* 1 ((lambda (n) (if (= n 0) 1 (* n (fact-rec (- n 1))))) 0))))
(* 3 (* 2 (* 1 (if (= 0 0) 1 (* 0 (fact-rec (- 0 1)))))))
(* 3 (* 2 (* 1 (if #t 1 (* 0 (fact-rec (- 0 1)))))))
(* 3 (* 2 (* 1 1)))
(* 3 (* 2 1))
(* 3 2)
6
```

Note that there is no need to introduce notions of function activations or stacks, nor is there any need to handle recursive functions applications any differently than non-recursive ones. The same rules that handle "regular" function applications handle recursive ones.

It's worth noting that a sort of stack-like structure natural emerges out of the rules for evaluating an application. Recall that all subexpressions of an application must be rewritten to a value before the operator can be applied to the resulting values. This means that operator applications remain "pending" while their arguments are being calculated; this gives rise to what is normally known as the call stack. This stack-like nature of recursive procedure call evaluation is more apparent in a more stylized rewrite sequence that only highlights certain intermediate expressions:

```
subst> (fact-rec 5)
(fact-rec 5)
(* 5 (fact-rec 4))
(* 5 (* 4 (fact-rec 3)))
(* 5 (* 4 (* 3 (fact-rec 2))))
(* 5 (* 4 (* 3 (* 2 (fact-rec 1)))))
(* 5 (* 4 (* 3 (* 2 (* 1 (fact-rec 0))))))
(* 5 (* 4 (* 3 (* 2 (* 1 1)))))
(* 5 (* 4 (* 3 (* 2 1))))
(* 5 (* 4 (* 3 2)))
(* 5 (* 4 6))
(* 5 24)
120
```

The growing and shrinking "bulge" above is characteristic of the stack-like nature of a recursive process.

Of course, it is possible to express tree recursions as well as linear recursions. Here is the Scheme version of the classic Fibonacci function

```
(define fib-rec
  (lambda (n)
    (if (< n 2)
        n
        (+ (fib-rec (- n 1)) (fib-rec (- n 2))))))
```

and here is a manually formatted transcript of the call `(fib-rec 4)`:

```
subst> (fib-rec 4)
(+ (fib-rec 3) (fib-rec 2))
(+ (+ (fib-rec 2) (fib-rec 1))
   (+ (fib-rec 1) (fib-rec 0)))
(+ (+ (+ (fib-rec 1) (fib-rec 0))
      1)
   (+ 1 0))
(+ (+ (+ 1 0)
      1)
   1)
(+ (+ 1
      1)
   1)
(+ 2
   1)
3
```

What about iteration? Since Scheme does not provide any mutable variables or iterative constructs like **`while`**, **`for`**, or **`repeat/until`**, is it possible to express iteration in Scheme? For example, an iterative solution to computing the factorial of 5 would be to update two state

variables in a loop as follows:

Number	Result
5	1
4	5
3	20
2	60
1	120
0	120

It turns out that it is possible to express this sort of iterative process in Scheme without mutable variables or iteration constructs. All that is needed is procedure calls! The iterative factorial process is expressed in Scheme using two procedures:

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

(define fact-loop
  (lambda (number result)
    (if (= number 0)
        result
        (fact-loop (- number 1) (* result number)))))
```

The `fact-iter` procedure is a one-argument procedure that is the entry point for starting the process. All it does is to call `fact-loop`, a procedure whose two arguments represent the state variables in the above table, with the appropriate initial values. Here is a stylized trace of the substitution model on a sample call:

```
subst> (fact-iter 5)
(fact-loop 5 1)
(fact-loop 4 5)
(fact-loop 3 20)
(fact-loop 2 60)
(fact-loop 1 120)
(fact-loop 0 120)
120
```

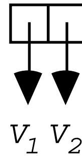
Each line corresponds precisely to one row of the above table. Note how Scheme avoids the need for mutable variables in this example. Rather than creating two state variables and changing their contents over time, each call to fact-loop effectively introduces new copies of the `number` and `result` parameters. Scheme avoids the need for iteration constructs by simply using an explicit procedure call to reenter the loop. Note that this process does not have a growing and shrinking bulge but takes a constant amount of space throughout. The reason is that there are no pending operations that are waiting for `fact-loop` to return. A procedure call that is not nested within pending operations is said to be a **tail call**. Scheme implementations are required to be **properly tail recursive**, which means that they must implement tail calls without pushing any information on the run-time stack of execution frames. This guarantees that processes like those generated by `fact-loop` will exhibit the same desirable constant space behavior as `while` and `for` loops in Java, C, and other languages with special looping constructs.

6. Data Structures in Scheme (Lists, Trees, S-Expressions)

This section covers compound data structures in Scheme. Pairs are the usual way of composing data in Scheme; they can be used in idiomatic ways to construct lists and s-expressions (trees). Scheme also supports vectors, which are an array-like data structure; see *R5RS* for details. First-class functions are another kind of compound data in Scheme; these will be covered in a later handout.

6.1. Pairs

In Scheme, any two values V_1 and V_2 can be glued together into a pair via the primitive `cons` function. Pictorially, the result of `(cons` V_1 V_2 `)`.is shown as a box-and-pointer diagram:



In the substitution model, `cons` itself is treated as a primitive function like `+` and `<=`, but `(cons` V_1 V_2 `)` is treated as a new kind of value that can be the result of an evaluation. For example:

```
subst> (cons (+ 1 2) (- (* 3 4) 5))
(cons 3 (- 12 5))
(cons 3 7)

subst> (cons (* 6 7) (= 8 8))
(cons 42 #t)
```

The second example illustrates that pairs are **heterogeneous** data structures -- i.e., their components need not have the same type.

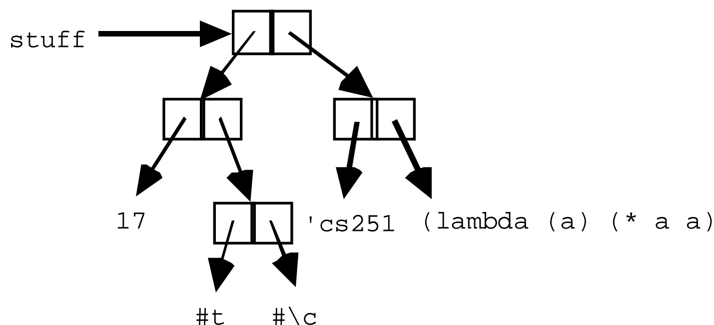
The left and right components of a pair are extracted, respecitively, by the `car` and `cdr` functions:

```
subst> (car (cons V1 V2 ))
V1

subst> (cdr (cons V1 V2 ))
V2
```

(The names `car` and cdr are vestiges of an early Lisp implementation on the IBM 704. `car` stands for "contents of address register" and `cdr` stands for "contents of decrement register".)

Arbitrarily complex data structures can be built with nested `cons`es and their components can be extracted by sequences of nested `car`s and `cdr`s. For example, consider the following box-and-pointer structure, named `stuff`:



Below, give an expression that generates the data structure depicted by the above diagram:

```
(define stuff


 )
```

For each of the components of `stuff`, given an expression that extracts that component:

```
17

#t

#\c

'cs251

(lambda (a) (* a a))
```

Notes:

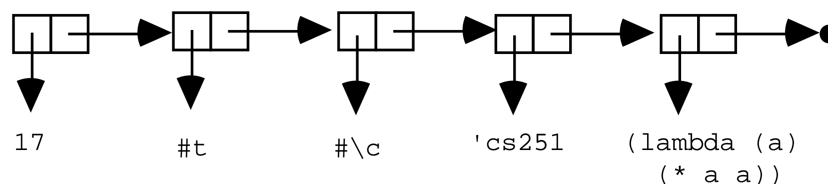
- The fact that functions can be stored in data structures is hallmark of functional programming and a powerful feature that we will exploit throughout this course.
- Nested combinations of `car`s and `cdr`s are so common that most Scheme implementations provide primitives that abbreviate up to four such nested applications. The abbreviations are equivalent to the following definitions:

```
(define caar (lambda (x) (car (car x))))
(define cadr (lambda (x) (car (cdr x))))
(define cdar (lambda (x) (cdr (car x))))
(define cddr (lambda (x) (cdr (cdr x))))
....
(define cdddar (lambda (x) (cdr (cdr (cdr (car x))))))
(define cddddr (lambda (x) (cdr (cdr (cdr (cdr x))))))
```

- Scheme interpreters usually print pairs using the "dotted pair" notation. For example, the printed representation of the pair `(cons 17 42)` is `(17 . 42)`. However, due to the list printing conventions described in the next section, the story is more complex. For instance, we might expect that the printed representation of `(cons (cons 1 2) (cons 3 4))` should be `((1 . 2) . (3 . 4))`, but it is in fact `((1 . 2) 3 . 4)`.

6.2. Lists

In Scheme programs it is rare to use `cons` to pair two arbitrary values. Instead, it is much more common to use sequences of pairs connected by their `cdr`s to represent linked lists (which in Scheme are simply called “lists”). For example, here is a list of the five values from the example of the previous section.



The sequence of pairs connected by their `cdr`s is known as the **spine** of the list.

A list is defined inductively as follows. A list is either

- An empty list (shown as the solid black circle in the above diagram).
- A pair whose `car` is the head of the list and whose cdr is the `tail` of the list.

The Scheme notation for an empty list is `'()`; the primitive predicate `null?` returns true for the empty list and false otherwise. Thus, the above list can be constructed as follows:

```
(cons 17
      (cons #t
            (cons #\c
                  (cons 'cs251
                        (cons (lambda (a) (* a a))
                              '()))))))
```

Such nested sequences of `cons`es are very difficult to read. They can be abbreviated using the primitive `list` procedure, which takes *n* argument values and constructs a list of those *n* values. For example, the above expression can be written more succinctly as follows:

```
(list 17 #t #\c 'cs251 (lambda (a) (* a a)) )
```

In fact, it helps to think that list desugars into a nested sequence of `cons`es (it turns out that this is not quite true, but is close enough to being true that it's not harmful to think this way):

(list V_1 V_2 ... V_n) desugars to (cons V_1 (cons V_2 ... (cons V_n '()) ...))

The elements of a list can be accessed by cars and cdrs, just as before.

These list operations should seem very familiar from CS111 and CS230! Indeed, the list material in these courses was patterned after Scheme list operations, so the similarity is not coincidental. Here is the correspondence between the Java list operations you've seen before and the Scheme list operations:

Java list operation	Scheme list operation
`prepend(x,L)`	`(cons x L)`
`head(L)`	`(car L)`
`tail(L)`	`(cdr L)`
`empty()`	`'()`
`IsEmpty(L)`	`(null? L)`

The following idioms are so common that you should commit them to memory (some Scheme systems supply these as primitives; if they don't, you can define them on your own):

```
(define first (lambda (lst) (car lst)))
(define second (lambda (lst) (car (cdr lst))))
(define third (lambda (lst) (car (cdr (cdr lst)))))
(define fourth (lambda (lst) (car (cdr (cdr (cdr lst))))))
(define but-first (lambda (lst) (cdr lst)))
(define but-second (lambda (lst) (cdr (cdr lst))))
(define but-third (lambda (lst) (cdr (cdr (cdr lst)))))
(define but-fourth (lambda (lst) (cdr (cdr (cdr (cdr lst))))))
```

Most Scheme system print a list a sequence of values delimited by parenthesis. Thus, the list value produced by `(list 3 #t #\c)` is usually printed as `(3 #t #\c)`. One way to think of this is that it is a form of the dotted-pair notation in which a dot "eats" a following open parenthesis. That is, `(3 . (#t . (#\c . ())))` becomes `(3 #t #\c)`. In most systems, symbols appear without a quotation mark, so the result of evaluating `(list 'a 'b 'c)` is printed (a b c). Procedural values are printed in an ad hoc way that varies greatly from system to system.

In addition to the `list` construct, there are two other standard list-building functions.

- `(cons` *`value list`*`)` prepends *`value`* to the front of *`list`*. If *`list`* has *n* elements, then `(cons` *`value list`*`)` has *n+1* elements.

- `(append` *`list1 list2`*`)` returns a list containing all the elements of *`list1`* followed by all

the elements of *list2*. If *list1* has *m* elements and *list2* has *n* elements, then (append *list1 list2*) has *m*+*n* elements.

For example, suppose that we have the following definitions:

```
(define L1 (list 1 2 3))
(define L2 (list 4 5))
```

What are the results of the following manipulations involving these lists?

```
> (list L1 L2)


> (cons 17 L1)


> (cons L1 L2)


> (append L1 L2)


> (append L1 L1)
```

We emphasize that all the list operations we're studying now are **non-destructive** -- they never change the contents of an existing list, but always make a new list. So when we say that "cons prepends a value to the front of a list", this is an abuse of language that really means "cons returns a list that is formed by prepending the given value to the given list." Later in the course, destructive list operators will be introduced in the context of imperative programming.

6.3 List Quotation

Scheme supports some syntactic sugar involving quotation to simplify writing complex list structures. Informally, '*elts* constructs the list structure whose printed representation is that of *elts*.

```
> '(1 a #t #\c)
(1 a #t #\c)

> '((a b) c (d (e f)))
((a b) c (d (e f)))
```

(Note that '*elts* is itself an abbreviation for (quote *elts*).)

More formally, you can imagine that Scheme has the following syntactic sugar for quoted list structure. (This is not really the case, but it's close enough to being true that you can believe it.)

'*N desugars-to N*
'*B desugars-to B*
'*C desugars-to C*
'*R desugars-to R*
'(S_1 … S_n) *desugars-to* (list 'S_1 … 'S_n)

6.4 List Recursion

The inductive definition of lists naturally leads to functions on lists that are defined recursively. Below are some representative functions, some of which we will define in class. Those that we don't get to you should define on your own (see me or the tutor if you need help!)

Note: In CS111 and CS230, you've already seen versions of many of the following functions in Java. Two great advantages of Scheme over Java are that (1) Scheme lists are **heterogeneous**: a single list may contain elements of many different types) and (2) Scheme list functions are **polymorphic**: they work on any list, regardless of the types of its elements.

In contrast, Java lists must contain elements that are all (subtypes of) a given type, and Java list functions work only for lists whose elements are a particular type. For instance, in Java, we defined classes like `IntList` and `BoolList` for representing lists of integers and lists of booleans, and it was necessary to write different `length()`, `append()`, `reverse()`, etc. methods for each such class even though the code for these methods never examines the elements.

One way of finessing these problems in Java is to use an `ObjectList` class in which every element is an `Object`. Then all list methods can be defined exaclty once on `ObjectList`. However, as we saw in CS111 and CS230, there are two problems with this approach: (1) Java's type system requires an explicit cast to be applied to elements extracted from an ObjectList; and (2) since primitive datatypes like `int`, `boolean`, etc. are not objects, they must be packaged into and unpackaged from wrapper classes like `Integer`, `Boolean`, etc. These infelicities make the Java list programs less readable and less general than their Scheme counterparts.

(**length** *list*)

Return the number of elements in the list.

```
> (length '(a b c))
3

> (length '(17))
1

> (length `())
0
```

(**sum** *list*)

Sum the elements of a list of numbers

```
> (sum '(2 4 6))
12

> (sum `())
0
```

(**from-to** *lo hi*)

Return a list of all the integers between lo and hi, inclusive

```
> (from-to 3 7)
(3 4 5 6 7)

> (from-to 7 3)
()
```

(**squares** *list*)

Return a list whose values are the squares of the given list of numbers

```
> (squares '(5 1 3 2))
(25 1 9 4)
```

(**evens** *list*)

Return a list containing only even numbers in the given list of numbers

```
> (evens '(5 1 4 2 7 6))
(4 2 6)

> (evens '(5 1 3 7))
()
```

(**member?** *value list*)

Determine if *value* is a member of *list*. (Assume `equal?` is used to test equality.)

```
> (member? 2 '(5 1 4 2 7 6))
#t

> (member? 17 '(5 1 4 2 7 6))
()
```

(**remove** *value list*)

Remove all occurrences of *value* from *list.* (Assume `equal?` is used to test equality.)

```
> (remove 3 '(1 3 2 3 3 4 3 1 2))
(1 2 4 1 2)

> (remove 3 '(1 2 4 1 2))
(1 2 4 1 2)
```

(**remove-duplicates** *list*)

Retunrns a list in which each element of appears only once. The order of elements in the resulting list is irrelevant. (Assume `equal?` is used to test equality.)

```
> (remove-duplicates '(1 3 2 3 3 4 3 1 2))
(1 3 2 4)
```

(**append** *list1 list2*)

Append the values of two lists

```
> (append '(1 2 3) '(list a b c))
(1 2 3 a b c)
```

(**reverse** *list*)

Return a list whose elements are in reverse order from the given list.

```
> (reverse '(1 2 3))
(3 2 1)

> (reverse '((1 2 3) 4 (5 6)))
((5 6) 4 (1 2 3)
```

Note: there are numerous ways to define `reverse`. Try to define it both recursively and iteratively. You may find the following helper function (which corresponds to the `postpend()` method we studied in Java) function for the recursive definition:

```
(define snoc
  (lambda (lst elt)
    (if (null? lst)
        (list elt)
        (cons (car lst) (snoc (cdr lst) elt)))))
```

(**zip** *list1 list2*)

Zip two corresponding elements of two lists together, as shown below. If the two lists differ in size, the resulting list should have the length of the shorter of the two inputs:

```
> (zip '(1 2 3) '(a b c))
((1 a) (2 b) (3 b))

> (zip '(1 2) '(a b c))
((1 a) (2 b))

> (zip '(1 2 3) '(a b))
((1 a) (2 b))
```

5.4. S-Expressions

A **symbolic expression**, or **s-expression** for short, is defined inductively as follows. An s-expression is either:

- a literal (i.e., number, boolean, symbol, character, string, or empty list)
- a list of s-expressions.

(Note: many Scheme texts, such as SICP, refer to a literal as an **atom**.) An s-expression can be viewed as a tree where each list corresponds to an (unlabelled) tree node, and each subexpression corresponds to a subtree.

There are many functions on trees/s-expressions. Here's a representative one:

(**flatten** sexp)
Return a list of the leaves of the tree *sexp* in an in-order traversal.

```
> (flatten '((1 2 3) 4 (5 (6 7)))
(1 2 3 4 5 6 7)

> (flatten 1)
(1)
```