CS251 Programming Languages **Handout # 13'**
Prof. Lyn Turbak **February 18, 2001**
Wellesley College

Simple Interpretation (Final Version)

1 Introduction

An *interpreter* is a program written in one language (the *implementation language*) that executes a program written another language (*the source language*). The implementation and source languages are typically different. For example, we might write an interpreter for Java in Scheme. We would call such an interpreter a "Java interpreter", naming it after the source language, not the implementation language. It is possible to write an interpreter for a language in itself; such an interpreter is known as a *meta-circular interpreter*. For example, chapter 4 of *SICP* presents a meta-circular interpreter for Scheme.

One of the best ways to gain insight into programming language design and implementation is read, write, and modify interpreters. We will spend a significant amount of time in this course studying interpreters. We begin with an intepreter for an extremely simple "toy" language, an integer expression language we'll call INTEX. To understand various programming language features, we will add them to INTEX to get more complex languages. Eventually, we will build up to intepreters for source languages similar to Scheme, ML, and Haskell.

2 Abstract Syntax for INTEX

An INTEX program specifies a function that takes any number of integer arguments and returns an integer result. Abstractly, an INTEX program is a tree constructed out of two different kinds of nodes:

- A **program** node with two subtrees:
  - A formal parameter list *formals*, which is a list of identifiers;
  - A *body* expression.
- An **expression** node is one of the following:
  - A **literal** specifying an integer constant, known as its *value*.
  - A **variable reference** specifying an identifier, known as its *name*.
  - A **binary application** specifying a binary operator (known as its *rator*) that is one of the five operators `+`, `-`, `*`, and `div`, and two operand expressions (known as *rand1* and *rand2*).

These nodes can be be depicted graphically and arranged intro trees. For example, Figure 1 depicts the tree denoting an INTEX averaging program. Such trees are known as **abstract syntax trees**, because they specify the abstract logical structure of a program without any hint of how the program might be written down in concrete syntax. (See Section 6 for a discussion of concrete syntax.)

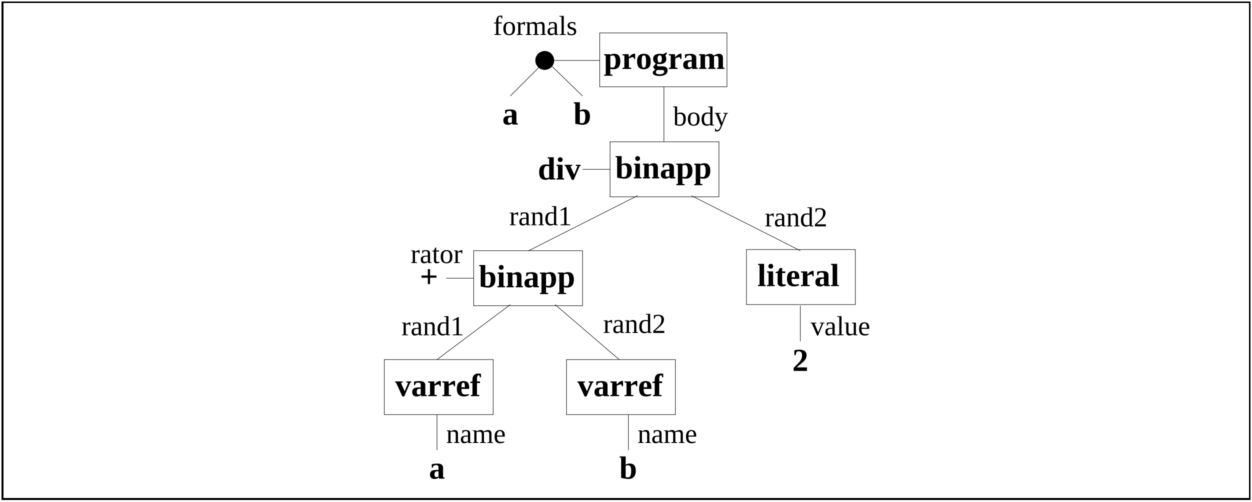


Figure 1: Abstract syntax tree for an INTEX averaging program.

INTEX program and expression trees can be viewed as abstract data types (ADTs) that admit many possible implementations. Figure 2 presents the contracts for Scheme functions implementing these ADTs. Using these functions, the INTEX tree in Figure 1 can be constructed in Scheme via the following expression:

```
(define avg
  (make-program (list 'a 'b)
    (make-binapp 'div
                 (make-binapp '+
                              (make-varref 'a)
                              (make-varref 'b))
                 (make-literal 2))))
```

Here, Scheme symbols are used to represent INTEX variable names and operator names, and Scheme lists are used to represent the formal parameter lists in INTEX.

3 Manipulating INTEX Programs

Since INTEX programs are just trees, they can be constructed, traversed, and decomposed via standard tree manipulation techniques. In particular, functions that take INTEX expressions as arguments are typically recursive functions that perform a case analysis on the type of node at the root of the given expression tree. We introduce this idiom in the context of two examples.

3.1 Example 1: Program Size

For example, the following `exp-size` function calculates the number of nodes in an INTEX expression tree:

Program Nodes

(make-program *formals body*)
Returns a program node with the formal parameter list *formals* and body expression *body*.

(program-formals *pgm*)
Returns the formal parameter list of *pgm*.

(program-body *pgm*)
Returns the body of *pgm*.

(program? *node*)
Returns `#t` if *node* is a program node, and `#f` otherwise.

Expressions Nodes

Integer Literals

(make-literal *int*)
Returns a literal node whose value is the integer *int*.

(literal-value *lit*)
Returns the value of the literal node *lit*.

(literal? *node*)
Returns `#t` if *node* is a literal node, and `#f` otherwise.

Variable References

(make-varref *sym*)
Returns a variable reference node whose name is the given symbol *sym*.

(varref-name *varref*)
Returns the name of the variable reference node *varref*.

(varref? *node*)
Returns `#t` if *node* is a variable references node, and `#f` otherwise.

Binary Applications

(make-binapp *rator rand1 rand2*)
Returns a binary application node whose rator is *rator* and whose operands are *rand1* and *rand2*.

(binapp-rator *binapp*)
Returns the operator of the binary application node *binapp*.

(binapp-rand1 *binapp*)
Returns the first operand expression of the binary application node *binapp*.

(binapp-rand2 *binapp*)
Returns the second operand expression of the binary application node *binapp*.

(binapp? *node*)
Returns `#t` if *node* is a binary application node, and `#f` otherwise.

Figure 2: Contract for Scheme functions implementing INTEX abstract syntax trees.

```
(define exp-size
  (lambda (exp)
    (if (literal? exp)
        1
          (if (varref? exp)
            1
            (if (binapp? exp)
                   (+ 1
                      (exp-size (binapp-rand1 exp))
                      (exp-size (binapp-rand2 exp)))
                (error "EXP-SIZE: Unrecognized expression --" exp))))))
```

The `error` invocation in the final line handles the case where `exp` is not a legal INTEX expression. (See Appendix B for details of the `error` construct.) This final clause will never be executed if `exp` is generated by the constructors in Figure 2. We include it as part of a defensive programming methodology for handling ill-formed inputs. (Such a clause is not necessary in languages like ML and Haskell, where the type system provides a guarantee that such errors cannot possibly happen.)

Using nested `if`s to perform the case analysis on node type has the disadvantage that the code wanders towards the right-hand side of the page. This can be avoided by using Scheme's `cond` construct, which is just syntactic sugar for a sequence of nested `if`s:

```
(define exp-size
  (lambda (exp)
    (cond ((literal? exp) 1)
          ((varref? exp) 1)
          ((binapp? exp)
           (+ 1
              (exp-size (binapp-rand1 exp))
              (exp-size (binapp-rand2 exp))))
          (else (error "EXP-SIZE: Unrecognized expression --" exp))
          )))
```

To find the size of an INTEX *program* (as opposed to an expression), we write a separate function:

```
(define program-size
  (lambda (pgm)
    (+ 1 (exp-size (program-body pgm)))))
```

We can test our functions out on the `avg` program constructed in Section 2.

```
(program-size avg)
```
6

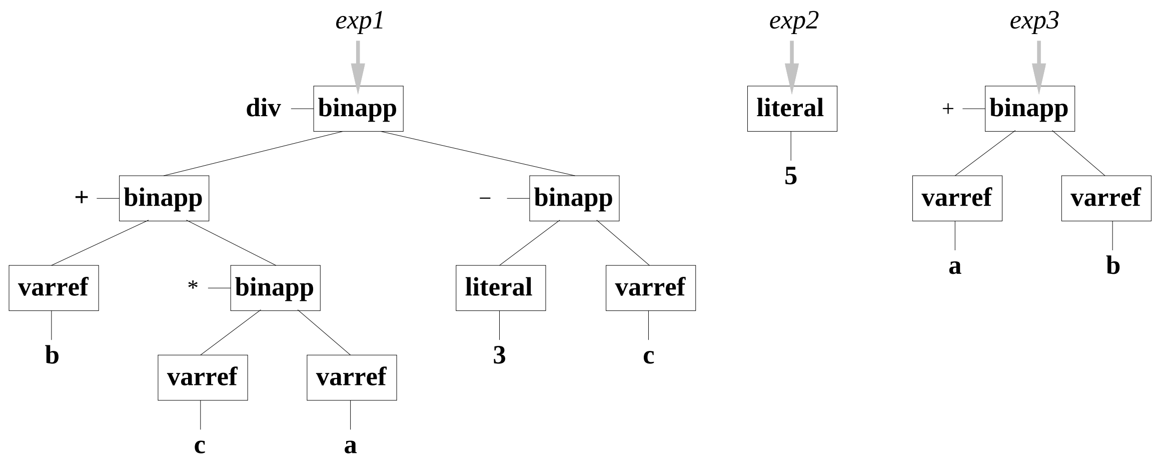
```
(exp-size (program-body avg))
```
5

```
(exp-size (binapp-rand1 (program-body avg)))
```
3

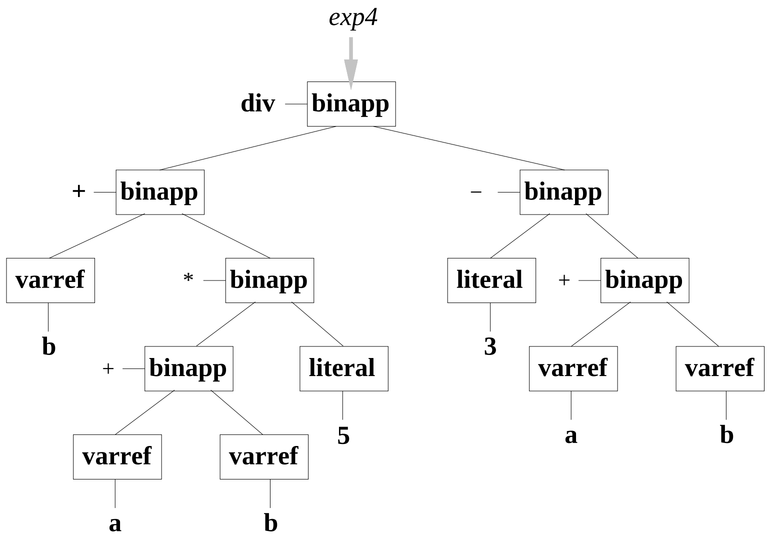
```
(exp-size (binapp-rand2 (program-body avg)))
```
1

3.2 Example 2: Substitution

As a second example, we consider substituting expression trees for variable reference nodes that appear in a given tree. For instance, consider the following three expression trees:



Suppose we want to substitute *exp2* for the variable reference node named `a` and *exp3* for the variable reference nodes named `c` in the tree *exp1*. The result of such a substitution is the following expression *exp4*:



We make the following obervations concerning substitution:

- The expression tree being substituted for a variable node can be any tree. Because trees are replaced by trees, the result of substitution is always a well-formed expression tree.
- The result of substitution is always a "new" tree. That is, trees are immutable, and the only way to "modify" a tree is to make a copy of it that differs in certain positions. For instance, in the above example, *exp1*, *exp2*, and *exp3* are exactly the same after the substitution is performed as they were before.
- Those variables that are not replaced by substitution (such as `b` in the example) are preserved in the resulting tree.
- If there are several occurrences of the variable being replaced by substitution, a copy of the replacement tree will made for each occurrence. For example, since *exp1* contains two occurrences of `c`, *exp4* contains two copies of *exp3*. (If there are no occurrences of the variable being replaced, no copies of the replacement tree will appear in the resulting tree.)
- The replacement tree may itself contain variable references, including ones being replaced by the substitution. However, any variables appearing in the replacement trees are *not* replaced by the substitution process. For instance, *exp3* contains a reference to `a`, but this is not replaced by *exp2*. This is the semantics of so-called **simultaneous substitution**, in which substitutions for several variables are performed in parallel. If we really want the `a` in *exp3* to be replaced by *exp2*, we can first substitute *exp3* for `c` in *exp1*, and then perform a second substitution of *exp2* for `a` in the result of the first substitution.

Our goal is to write a `subst` function that performs substitution on INTEX trees. The `subst` function must take a specification of which variables are being replaced, the replacement trees for each variable, and the tree in which the substitutions are being performed. There are many ways to specify the association between the replacement variables and corresponding replacements trees. For example, we could provide `subst` with a list of replacement variables and a list of replacement trees (ordered to correspond with the variables in the variable list). Alternatively, we could provide `subst` with a list of variable/replacement tree pairs.

Here, we will take a another approach: we will specify the association between variables and replacements trees via a data structure known as an **environment**. An environment is an immutable table that maps keys (in this case, variable names) to values (in this case, expression trees); it is sometimes called a **dictionary** or **symbol table**. It turns out that associating variable names to values is extremely common in programming language implementations, so we will be making heavy use of environments throughout this course. We might as well start using them here.

The complete contract for environments appears in Appendix A. For our purposes, we only need the following operations:

(**env-make** *names vals*)
Returns an environment containing bindings between the names in the list *names* and their corresponding values in the list *vals*.

(**env-lookup** *name env*)
Returns the value associated with *name* in environment *env*, or a distinguished *unbound token* if *name* is not bound in *env*.

(**unbound?** *thing*)
Returns true if *thing* is the distinguished unbound token, and false otherwise.

We will assume that the `subst` takes an environment specifying the desired substitutions (i.e., bindings between variable names and expression trees), and the expression in which the substitutions should be performed. For example, in the example above, *exp4* would be the result of the following invocation:

```
(subst (env-make '(a c) (list exp2 exp3))
       exp1)
```

Here is a definition of the `subst` function assuming this interface:

```
(define subst
  (lambda (env exp)
    (cond ((literal? exp) exp)
          ((varref? exp)
           (let ((probe (env-lookup (varref-name exp) env)))
             (if (unbound? probe) exp probe)))
          ((binapp? exp)
           (make-binapp (binapp-rator exp)
                        (subst env (binapp-rand1 exp))
                        (subst env (binapp-rand2 exp))))
          (else (error "SUBST: Unrecognized expression -- " exp))
          )))
```

The `subst` function perform substitution by recursively descending the tree with an unchanging substitution environment “in hand”. At a variable reference node, it uses the environment to either replace the node (if the name is bound in the environment) or leave the node unchanged (if the name is not bound in the environment). On the way out of the recursion, `subst` makes a copy of all the non-variable nodes; it returns the copy of the root node on which it was initially invoked. In terms of information flow, we can imagine that the environment flows downward from the top of the tree to the leaves, and that the new nodes flow upward toward the root, combined along the way to form the resulting tree.

To hide the construction of environments used by `subst`, it is helpful to define the following auxliary functions:

```
(define subst1
  (lambda (new old exp)
    (subst (env-make (list old) (list new)) exp)))

(define subst*
  (lambda (news olds exp)
    (subst (env-make olds news) exp)))
```

The invocation `(subst1` *new old exp*`)` substitutes the expression *new* for variable references named *old* in *exp*. The `subst*` function is similar, but specifies a list of replacement expressions and a list of variables. Note that the order of replacement expressions and variables in `subst1` and `subst*` is different from the order in which they are supplied to `env-make`.

4 A Substitution Model Interpreter for INTEX

We would like to be able to "run" or "evaluate" INTEX programs. Intuitively, this involves supplying actual integer arguments corresponding to the formal parameters, and then calculating the value of the program body assuming that each formal parameter stands for the corresponding supplied integer argument. We have discussed a **substitution model** for Scheme that formalizes these intuitions via a collection of rewriting rules. We can adapt these rules to INTEX and implement them in Scheme.

At the core of the substitution model is the `subst` function from the previous section, so we are already well on our way to implementing an INTEX evaluator. What's missing is a means of calculating the value of an expression tree consisting of only binary applications and integers. Let's assume that we have a function `(subst-eval` *exp*`)` that takes such an expression tree and returns an integer literal node (*not* an integer) corresponding to the value of calculating the operations in the tree. Then we can express the evaluation of INTEX programs via the following `subst-run` function:

```
(define subst-run
  (lambda (pgm ints)
    (literal-value
     (subst-eval (subst* (map make-literal ints)
                         (program-formals pgm)
                         (program-body pgm))))))
```

This function takes two arguments: an INTEX program *pgm*, and a list of integers *ints*. It returns the integer that is the result of evaluating the body of *pgm* in a context where the formal parameters in *pgm* stand for the corresponding integers in *ints*.

The `subst-run` function works in two passes over the tree. In the first pass, it substitutes the actual arguments for the references to the formal parameters appearing in the body of the program. Note that because the first argument to `subst*` must be a list of expression trees, it is necessary to map `make-literal` over the integer arguments. In the second pass, `subst-eval` calculates the value of the post-substitution expression tree. By the specification of `subst-eval` this value is an integer literal node, so `subst-run` must use `literal-value` to extract its value in order to return an integer.

To complete the definition of the evaluator, it is necessary to provide a definition of `subst-eval`:

```
(define subst-eval
  (lambda (exp)
    (cond ((literal? exp) exp)
          ((varref? exp)
           (throw 'unbound-variable (varref-name exp)))
          ((binapp? exp)
           (make-literal
            ((binop-to-node-function (binapp-rator exp))
             (subst-eval (binapp-rand1 exp))
             (subst-eval (binapp-rand2 exp)))))
          (else (error "SUBST-EVAL: Unknown expression type -- " exp))
          )))
```

This function recursively determines the values of the operands of a binary application, and then calculates the result of performing the specified operator on these values. The details of performing the arithmetic operations is hidden in `binop-to-node-function`. It is an error if `subst-eval` encounters a variable, since all formal parameters should have been already replaced by the substitution pass. Any remaining variables are so-called **unbound variables**. For reasons described in Appendix B, unbound variable errors are reported via the `throw` construct rather than the `error` construct.

All that remains to be explained is how arithmetic operations are performed. The following function maps a symbol denoting a binary operation into a Scheme function that performs that operation.

```
(define binop-to-function
  (lambda (binop)
    (cond ((eq? binop '+) +)
          ((eq? binop '-) -)
          ((eq? binop '*) *)
          ((eq? binop 'div)
           (lambda (a b)
             (if (= b 0)
                 (throw 'division-by-zero (list 'div a b))
                 (quotient a b))))
          ((eq? binop 'mod)
           (lambda (a b)
             (if (= b 0)
                 (throw 'division-by-zero (list 'mod a b))
                 (remainder a b))))
          (else (error "BINOP-TO-FUNCTION: unrecognized binop " binop))
          )))
```

The functions for `div` and `mod` trap the case where the second argument is 0, and use `throw` (rather than `error`) to report the error. Because the INTEX language only permits the manipulation of integer values, it is important that all INTEX operators return integers. Note that `quotient` and `remainder` are Scheme's functions for calculating integer quotients and remainders. For example, `(quotient 7 2)` evaluates to 3 and `(remainder 7 2)` evaluates to 1.

As it stands, `binop-to-function` function does not have the right interface for use in `subst-eval`, where an operator must take two integer literal nodes and return an integer literal result. The following `binop-to-node-function` performs the requisite coercions:

```
(define binop-to-node-function
  (lambda (binop)
    (lambda (lit1 lit2)
      (make-literal
        ((binop-to-function binop)
         (literal-value lit1)
         (literal-value lit2))))))
```

At this stage, the concern over whether `subst-eval` should return a literal node rather than an integer may seem rather pedantic. Certainly, `subst-eval` could be written more simply if it returned an integer rather than an integer literal node. However, there is an important reason to make this distinction. The essence of the substitution model is rewriting one expression into another. If one operand of a binary application is another binary application, then the inner binary application must rewrite to a literal node in order for the outer binary application to be well-formed.

As written above, the `subst-eval` function hides the sequential nature of the rewriting process, and makes it a little difficult to see this point. It is possible to write `subst-eval` as an iteration that rewrites one binary application at each step until no more remain; in this alternative approach, the reason why `subst-eval` should return a node becomes more apparent. Moreover, we will see that having **subst-eval** return an expression node allows us to more easily handle various extensions to INTEX.

Having defined **subst-run**, we can now use it to evaluate INTEX programs like **avg**:

```
(subst-run avg '(3 8))
5

(subst-run avg '(-10 10))
0
```

5 An Environment Model Interpreter for INTEX

The substitution model interpreter makes two passes over the body of the INTEX program: one to perform the substitution of actual arguments for formal parameters, and the second to evaluate the resulting expression tree. It is not necessary to perform these two processes in independent passes over the tree. The two passes can be fused into a single function that walks over the body of the program only once. To make this possible, the function needs to take *two* arguments: (1) the expression tree and (2) an environment specifying the substitution. Below is an **env-eval** function implementing this idea:

```
(define env-eval
  (lambda (exp env)
    (cond ((literal? exp)
           (literal-value exp))
          ((varref? exp)
           (let ((probe (env-lookup (varref-name exp) env)))
             (if (unbound? probe)
               (throw 'unbound-variable (varref-name exp))
               probe)))
          ((binapp? exp)
           ((binop-to-function (binapp-rator exp))
            (env-eval (binapp-rand1 exp) env)
            (env-eval (binapp-rand2 exp) env)))
          (else ("ENV-EVAL: Unknown expression type -- " exp))
          )))
```

The environment specifies the values of the formal parameter names that might be used within the expression. The **env-eval** function effectively "delays" the substitution involving the formal parameters until the point at which it requires the value of such a parameter. Because this approach uses an extra environment parameter to encode the delayed substitution, it is known as the **environment model** of evaluation.

The environment model evaluator determines the *meaning* of each expression with respect to an environment that specifies the meanings of names appearing within the expression. In INTEX, the meaning of each expression is an integer. This means that the entities returned by **env-eval** and bound to names in the environments used by `env-eval` are integers rather than literal nodes with integer values. This differs from `subst`, where the environments bound names to literal nodes, and also differs from `subst-eval`, where the result was an expression node, not an integer. In this respect, the environment model is simpler than the substitution model. However, we will later see other programming language features which can be more straightforwardly described in the substitution model than in the environment model.

Evaluating a program via the environment model is accomplished via the following function, which is the environment model analog of `subst-run`:

```
(define env-run
  (lambda (pgm ints)
    (env-eval (program-body pgm)
              (env-make (program-formals pgm) ints))))
```

Of course, we can now try out `env-run`:

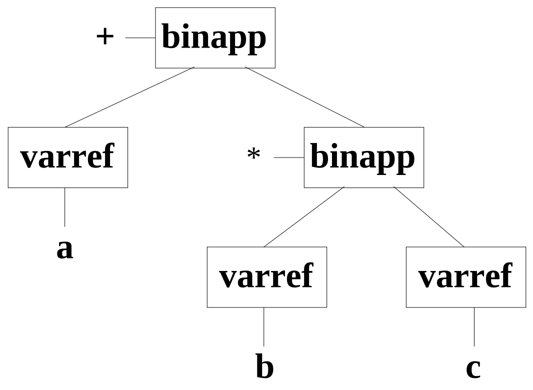
```
(env-run avg '(3 8))
5

(env-run avg '(-10 10))
0
```

6 Concrete Syntax for INTEX

We have seen that abstract syntax trees (ASTs) make it easy to write Scheme programs that manipulate INTEX programs. But it's rather awkward to use the AST functions from Section 2 to create INTEX program trees. Moreover, we can't very well expect that INTEX programmers are going to use AST functions written in Scheme to write INTEX programs!

To make programs easier to read and write, we can develop a **concrete syntax** that specifies how program trees can be written as sequences of characters. There are many design choices to make when definining concrete syntax. For example:

- What sorts of identifiers (i.e., variables names) are allowed in the language? Are they case sensitive? Can they only include letters and numbers, or can they include punctuation symbols like `_`, `-`, and `+`? For instance, is `a-b` a single identifier, or is it specifying the subtraction of variable `b` from variable `a`?
- How are applications of binary operators written? Suppose we want to represent the following INTEX expression tree:



There are many possible notations. For instance:

- In **infix notation**, the operator is placed between the operands, as in `a + (b * c)`. If we specify that `*` has a higher **precedence** than `+` (as is usually assumed in mathematical notation), we can dispense with the explicit parentheses and write `a + b * c`.
- In **prefix notation**, the operator is placed before the operands. We are already familiar with Scheme's version of prefix notation, in which the example would be written `(+ a (* b c))`. But there are other variants as well; e.g., `+(a,*(b,c))`.
- In **postfix notation**, the operator is placed after the operands, as in `b c * a +`. Postfix notation is unambiguous and can be written without any explicit parentheses.

• What should the top-level program structure be? How should formal parameters be declared and referenced? Again, there are many possibilities; here are a few:

- We could use C and Java's approach of using a distinguished `main` function to specify the entry point to a program:

  ```
  int main (int a, int b) {return (a + b) / 2;}
  ```

  In this case the `int` type annotations, `return` keyword, and semi-colon are "noise" in the sense that they don't have any meaning in INTEX and would be ignored. A syntax that mapped more straightforwardly onto INTEX would omit these:

  ```
  main (a, b) {(a + b) / 2}
  ```

- We could use a notation similar to that used to define mathematical functions:

  ```
  avg(a,b) = div(+(a,b),2)
  ```

  Here, the name `avg` has no semantic import (it is ignore by INTEX), but it is a helpful comment.
- Rather than giving variables names, we could instead refer to them by position, as is done in the TEX formatting language and various Unix shell script languages. A TEX-like notation for the averaging program might be:

  ```
  avg[2]{(#1 + #2)/2}
  ```

Here, the [2] indicates that the program takes two parameters, and #n references the nth parameter.

- We can represent the INTEX tree in s-expression notation, where each tree node is explicitly labelled by its type:

```
(program (a b)
  (binapp div
          (binapp +
                  (varref a)
                  (varref b))
          (literal 2)))
```

  This syntax makes the connection between this notation and INTEX trees very clear, but it is rather verbose. We can simplify it without ambiguity to a more concise s-expression notation that dispenses with the `binapp`, `varref`, and `literal` tags:

```
(program (a b)
  (div (+ a b) 2))
```

It turns out that the last approach (the "compressed" s-expression format) is particularly easy to implement and manipulate in Scheme, so we will adopt it as our concrete syntax for INTEX and its extensions. The ability to create trees by quoting s-expressions in Scheme makes it very convenient to manipulate INTEX programs and expressions. For instance:

```
(size-exp '(+ (* 2 a) (* 3 b)))
7

(env-run '(program (a b) (div (+ a b) 2)) '(3 8))
5

;; Perform a substitution in an expression.
(subst (env-make '(a c) '(3 5)) '(bind c (+ a b) (* c d)))
(bind c (+ 3 b) (* c d))
```

Note that we could have instead chosen any of the other approaches to concrete syntax mentioned above. But in general, this would require writing a program, known as a **parser**, that automatically constructs the INTEX program tree based on the sequence of characters in the concrete syntax. A discussion of parsing techniques is beyond the scope of this course; if you want to learn more about parsing, you should take CS301 (Compiler Design), in which a large chunk of the course is devoted to parsing technology.[1]

There are two drawbacks of using s-expression notation for the concrete syntax of the toy languages we will study.

[1]The beauty of s-expressions is that they are easily parsed and pretty-printed, and Scheme is already equipped with an s-expression parser and pretty-printer. This is why it is so convenient to use s-expressions for representing the syntax of our toy languages in Scheme; there's no need to write a parser or a pretty-printer!

1. The toy languages will end up looking a lot like Scheme. We must be careful to keep in mind that they are very different languages from Scheme even though they look like Scheme on the surface.

2. The fact that INTEX program trees are Scheme s-expressions may tempt you to break the barrier of the AST abstraction. For instance, if `avg` names the program tree for the averaging program, you may be tempted to write `(cadr avg)` to find the formal parameters rather than `(program-formals avg)`. You should *never* use "raw" Scheme list functions to manipulate nodes in a program tree. Instead, you should *always* use the functions specified in the AST contract. You should program defensively, assuming that (1) the implementation of the contract could change at any time and (2) you want your interpreters to work regardless of the implementation of the AST functions.

   This requires some discipline on your part. You should try to think in terms of the "types" of the entities you are manipulating. For instance, a literal node is conceptually a different kind of thing than the integer that is its value. In the "compressed" s-expression notation, these will unfortunately have exactly the same underlying representation, but you should still strive to distinguish cases where you are manipulating the node (e.g., when you are buiding an expression tree) vs. cases where you are manipulating the integer (e.g., when you are adding it to another integer). Later in the semester, we will program in languages (ML and Haskell) that have a much stronger notion of "type", and in which the literal node could never be confused with its integer value.

A Environments

An environment is an immutable table that maps names to values. Abstractly, it can be viewed as a collection of name/value bindings. Throughout this semester, we shall see that environments play a critical role in modeling how names are handled in programming languages.

The core interface to the environment abstract data type (ADT) is shown in Figure 3. There are numerous other functions for manipulating environments that are shown in Figure 4. All of these can easily be implemented in terms of the core functions above, but it is convenient to supply them as well.

B Error Handling

When implementing interpreters, we need to handle many different kinds of errors. For instance, the program being interpreted may not be a syntactically legal program. If it is syntactically legal, there are many other problems that can be encountered when it is run. The interpreter might attempt to evaluate a variable that has never been declared; or it might attempt to divide a number by zero; or it might attempt to apply a primitive operator to the wrong kinds of operands (e.g. adding 1 to true).

Scheme provides an `error` function that is useful for handling errors. It has the form `(error` *message irritant*`)`. The *message* argument is a string indicating the nature of the error, while *irritant* (which can be any Scheme value) indicates the source of the error. When Scheme evaluates an invocation of `error`, it suspends the current computation and prints out *message* and *irritant*. It

(env-empty)
Returns an environment with no bindings.

(env-bind *name val env*)
Returns an environment containing a binding between *name* and *val* in addition to the existing bindings of *env*. The new binding overrides any existing binding involving *name*.

(env-lookup *name env*)
Returns the value associated with *name* in environment *env*, or a distinguished *unbound token* if *name* is not bound in *env*.

unbound
A variable that names the distinguished unbound token.

(unbound? *thing*)
Returns true if *thing* is the distinguished unbound token, and false otherwise.

(env-names *env*)
Returns a list of the names that appear in the bindings of *env*. The order of the names is unspecified.

Figure 3: Core functions of the environment ADT.

(env-make *names vals*)
Returns an environment containing bindings between the names in the list *names* and their corresponding values in the list *vals*.

(bindings->env *alist*)
Returns an environment containing the bindings in the association list *alist*, a list of key/value duples.

(env-values *env*)
Returns a list of the values that appear in the bindings of *env*. The order of the values is unspecified. If a value is bound to n names, it will appear n times in the result.

(env->bindings *env*)
Returns an association list containing one name/value duple for each binding in *env*. The order of the duples is unspecified.

(env-keep *names env*)
Returns an environment containing only those bindings of *env* whose names are in the list *names*.

(env-remove *names env*)
Returns an environment containing only those bindings of *env* whose names are not in the list *names*.

(env-extend *names vals env*)
Returns an environment that, in addition to the existing bindings of *env*, contains bindings between the names in the list *names* and their corresponding values in the list *vals*. The new bindings override any existing bindings involving a name in *names*.

(env-merge *env1 env2*)
Returns an environment containing all the bindings in *env1* and *env2*. When a name is bound in both, the binding in *env1* takes precedence.

Figure 4: Other functions of the environment ADT.

also enters a debugging mode that allows the user to inspect the state of the computation when the error occurred.

There are situations where the aborting behavior of `error` is undesirable. Suppose we want to write a function that tests an interpreter on a test suite of source language programs, some of which have errors. We don't want the testing function to abort the first time it encounters a program with an error; rather, we'd like it to be able to observe the error and go on to test the next program.

The ability to recover gracefully from abnormal situations is usually provided by what is known as an **exception handling system**. In an exception handling system, it is possible (1) to indicate that an exceptional situation has occured by **throwing** an exception and (2) to **catch** and handle exceptions thrown by other parts of the program. For instance, Java programs can use the `throw` construct to "throw" an exception value that is "caught" by the nearest dynamically enclosing instance of `try/catch`. Any pending computation between the point of the `throw` and the point of the `try/catch` is aborted. (If you don't understand this, don't worry - we will study exception handling in more detail later in the semester.)

Such an exception handling mechanism is not a part of standard Scheme, but it is easy to implement one (in less than a page of code!). The mechanism uses `throw` and `catch` functions specified in Figure 5. As an example, consider the following function and sample invocations:

```
(define catch-test
  (lambda (x)
    (catch (lambda ()
             (* (+ (if (even? x)
                       (throw 'even x)
                       x)
                     1)
                (- (if (< x 0)
                       (throw 'negative x)
                       x)
                     1)))
           (lambda (tag value) (list tag value)))))

> (catch-test 3)
8 ; = (+ (+ 3 1) (- 3 1)

> (catch-test 4)
(even 4)

> (catch-test -3)
(negative -3)
```

When implementing interpreters in this course, we will use `throw` rather than Scheme's `error` construct to indicate that a non-syntactic error has occurred. This will make it possible to write functions that can test suites of source language programs that contain programs with errors in them. The testing function can catch any exception raised during the evaluation of a program, and proceed to test the next program. Note that you will only have to use `throw` in your programs; you should not have to write any instances of `catch`.

(throw *tag info***)**
Throw an exception with tag *tag* and information value *info* to the nearest dynamically enclosing catch, where it will be handled. All pending computation between the point of the `throw` and the point of the `catch` is aborted. It is assumed that the entire program is wrapped in a default exception handler that prints out *tag* and *info*.

(catch *thunk handler***)**
Assume that *thunk* is a zero-parameter procedure and *handler* is a two-argument procedure. Establish an exception handler handler that is in effect during the computation generated by applying *thunk* to zero arguments. If the computation does not throw any exceptions, then `catch` returns with the value of the computation. But if the computation throws an exception, `catch` returns the result of applying handler to the tag and info of the exception.

Figure 5: Exception handling contract.