The Kamin Interpreters in C++

Tim Budd

January 17, 2002

Abstract

This paper describes a series of interpreters for the languages used in the book "Programming Languages: An Interpreter-Based Approach" by Samuel Kamin (Addison-Wesley, 1989). Unlike the interpreters provided by Kamin, which are written in Pascal, these interpreters are written in C++. It is my belief that the use of inheritance in C++ better illustrates the unique features of each of the several languages. In the Pascal versions of the interpreters the differences between the various interpreters, although small, are scattered throughout the code. In the C++ versions differences are produced using only the mechanism of subclassing. This means that the vast majority of code remains the same, and differences can be much more precisely isolated.

The chapters in this report correspond to the chapters in the original text. Where motivational or background material is provided in that source it is generally omitted here. A major exception is in those places (chiefly chapters 3, 7 and 8) where I have selected a syntax slightly different from that provided by Kamin.

The use of an Object-Oriented language for the interpreters may seem a bit incongruous, since Object-Oriented programming is not discussed until Chapter 7. Nevertheless, I think the benefits of programming the interpreters in C++ outweighs this problem.

Chapter 1

The Basic Interpreter

The structure of our basic interpreter[1] differs somewhat from that described by Kamin. Our interpreter is structured around a small main program which manipulates three distinct types of data structures. The main program is shown in Figure 1.1, and will be discussed in more detail in the next section. Each of the three main data structures is represented by a C++ class, such is subclassed in various ways by the different interpreters. The three varieties of data structures are the following:

- **Readers**. Instances of this class prompt the user for input values, and break the input into a structure of unevaluated components. A single instance of either the class Reader or a subclass is created during the initialization process for each interpreter. The base reader class is subclassed in those interpreters which introduce new syntactic elements (such as quoted lists in Lisp or vectors in APL).
- **Environments**. An Environment is a data structure used to maintain a collection of symbol-value pairs, such as the global run-time environment or the values of arguments passed to a function. Values can be added to an environment, and the existing binding of a symbol to a value can be changed to a new value.
- **Expressions**. Expressions represent the heart of the system, and the differences between the various interpreters is largely found in the various different types of expressions they manipulate. Expressions know how to "evaluate themselves" where the meaning of that expression is determined by each type of expression. In addition expressions also know how to print their value, and that, too, differs for each type of expression.

In subsequent sections we will explore in more detail each of these data structures.

[1] It should be noted that our basic interpreter is not an interpreter for Basic.

```
main() {
   Expr entered;    // expression as entered by users

   // common initialization
   emptyList = new ListNode(0, 0);
   globalEnvironment = new Environment(emptyList, emptyList, 0);
   valueOps = new Environment(emptyList, emptyList, 0);
   commands = new Environment(emptyList, emptyList, valueOps);

   // language-specific initialization
   initialize();

   // the read-eval-print loop
   while (1) {
      entered = reader→promptAndRead();

      // see if expression is quit
      Symbol * sym = entered()→isSymbol();
      if (sym && (*sym == "quit"))
         break;

      // nothing else, must just be an expression
      entered.evalAndPrint(commands, globalEnvironment);
      }
}
```

Figure 1.1: The Read-Eval-Print Loop for the interpreters

1.1 The Main Program

Figure 1.1 shows the main program,[2] which defines the top level control for the interpreters. The same main program is used for each of the interpreters. Indeed, the vast majority of code remains constant throughout the interpreters.

The structure of the main program is very simple. To begin, a certain amount of initialization is necessary. There are four global variables found in all the interpreters. The variable emptyList contains a list with no elements. (We will return to a discussion of lists in Section 1.4.4). The three environments globalEnvironment, valueOps and commands represent the top-level context for the interpreters. The globalEnvironment contains those symbols that are accessible at the top level. The valueOps are those operations that can be performed at any level, but which are not symbols themselves that can be manipulated by the user. Finally commands are those functions that can be invoked only at the top level of execution. That is, commands cannot be executed within function

[2] I have omitted the "include" directives and certain global declarations from this figure. The complete code can be found elsewhere (where?).

definitions.

Following the common initialization the function initialize is called to provide interpreter-specific initialization. This chiefly consists of adding values to the three environments. This function is changed in each of the various interpreters.

The heart of the system is a single loop, which executes until the user types the directive quit.[3] The reader (which must be defined as part of the interpreter-specific initialization) requests a value from the user. After testing for the the quit directive, the entered expression is evaluated. We will defer an explaination of the evalAndPrint method until Section 1.4, merely noting here that it evaluates the expression the user has entered and prints the result. The read-eval-print cycle then continues.

1.2 Readers

Readers are implemented by instances of class Reader, shown in Figure 1.2. The only public function performed by this class is provided by the method promptAndRead, which prints the interpreter prompt, waits for input from the user, and then parses the input into a legal, but unevaluated, expression (usually a symbol, integer or list-expression). These actions are implemented by a variety of utility routines, which are declared as protected so that they may be made available to later subclasses.

The code that implements this data structure is relatively straight-forward, and most of it will not be presented here. The main method is the single public-accessible routine promptAndRead, which is shown in Figure 1.3. This method loops until the user enters an expression. The method fillInputBuffer places the instance pointer p at the first non-space character (also stripping out comments). Thus lines containing only spaces, newlines, or comments are handled quickly here, and cause no further action. Also, as have noted previously, an end-of-input indication is caught by the method fillInputBuffer, which then places the quit command in the input buffer. The method readExpression (Figure 1.4) is the parser used to break the input into an unevaluated expression. This method is declared virtual, and thus can be redefined in subclasses. The base method recognizes only integers, symbols, and lists. The routine to read a list recursively calls the method to read an expression.

1.3 Environments

As we have noted already, the Environment data structure is used to maintain symbol-value pairings. In addition to the global environments defined during initialization, environments are created for argument lists passed to functions, and in various other contexts by some of the later interpreters. Environments can be linked together, so that if a symbol is not found in one environment

[3] The reader data structure will trap end-of-input signals, and if detected acts as if the user had typed the quit directive.

```
class Reader {
public:
    Expression * promptAndRead();

protected:
    char buffer[1000];    // the input buffer
    char * p;             // current location in buffer

    // general functions
    void printPrimaryPrompt();
    void printSecondaryPrompt();
    void fillInputBuffer();
    int  isSeparator(int);
    void skipSpaces();
    void skipNewlines();

    // functions that can be overridden
    virtual Expression * readExpression();

    // specific types of expressions
    int readInteger();
    Symbol * readSymbol();
    ListNode * readList();
};
```

Figure 1.2: Class Description of the reader class

```
Expression * Reader::promptAndRead()
{
   // loop until the user types something
   do {
      printPrimaryPrompt();
      fillInputBuffer();
      } while (! *p);

   // now that we have something, break it apart
   Expression * val = readExpression();

   // make sure we are at and of line
   skipSpaces();
   if (*p) {
      error("unexpected characters at end of line:", p);
      }

   // return the expression
   return val;
}
```

Figure 1.3: The Method promptandRead from class Reader

another can be automatically searched. This facilitates lexical scoping, for example.

For reasons having to do with memory management, the Environment data structure, shown in Figure 1.5, is declared as a subclass of the class Expression. Unlike other expressions, however, environments are never directly manipulated by the user. Also for memory management reasons, there is a class Env declared which can maintain a pointer to an environment. The two methods defined in class Env set and return this value. Anytime a pointer is to be maintained for any period of time, such as the link field in an environment, it is held in a variable declared as Env rather than as a pointer directly. Finally the overridden virtual methods isEnvironment and free in class Environment are also related to memory management, and we will defer a discussion of these until the next section.

The three methods used to manipulate environments are lookup, add and set. The first attempts to find the value of the symbol given as argument, returning a null pointer if no value exists. The method add adds a new symbol-value pair to the front of the current environment. The method set is used to redefine an existing value. If the symbol is not found in the current environment and there is a valid link to another environment the linked environment is searched. If the link field is null (that is, there is no next environment), the symbol and valued are added to the current environment.

Environments are implemented using the List data structure, a form of Ex-

```
Expression * Reader::readExpression()
{
    // see if it's an integer
    if (isdigit(*p))
        return new IntegerExpression(readInteger());

    // might be a signed integer
    if ((*p == '-') && isdigit(*(p+1))) {
        p++;
        return new IntegerExpression(- readInteger());
        }

    // or it might be a list
    if (*p == '(') {
        p++;
        return readList();
        }

    // otherwise it must be a symbol
    return readSymbol();
}

ListNode * Reader::readList()
{
    // skipNewlines will issue secondary prompt
    // until a valid character is typed
    skipNewlines();

    // if end of list, return empty list
    if (*p == ')') {
        p++;
        return emptyList;
        }

    // now we have a non-empty character
    Expression * val = readExpression();
    return new ListNode(val, readList());
}
```

Figure 1.4: The method readExpression and readList

```
class Environment : public Expression {
private:
    List theNames;
    List theValues;
    Env theLink;

public:
    Environment(ListNode *, ListNode *, Environment *);

    // overridden methods
    virtual Environment * isEnvironment();
    virtual void free();

    // new methods
    Expression * lookup(Symbol *);
    void add(Symbol *, Expression *);
    void set(Symbol *, Expression *);
};

class Env : public Expr {
public:
    operator Environment * ();
    void operator = (Environment * r);
};
```

Figure 1.5: The Environment data structure

```
Expression * Environment::lookup(Symbol * sym)
{
   ListNode * nameit = theNames;
   ListNode * valueit = theValues;

   while (! nameit→isNil()) {
      if (*sym == nameit→head())
         return valueit→head();
      nameit = nameit→tail();
      valueit = valueit→tail();
      }

   // otherwise see if we can find it on somebody elses list
   Environment * link = theLink;
   if (link) return link→lookup(sym);

   // not found, return nil value
   return 0;
}
```

Figure 1.6: The method lookup in class Environment

pression we will describe in more detail in Section 1.4.4. Two parallel lists contain the symbol keys and their associated values. For the moment it is only necessary to characterize lists by four operations. A list is composed of list nodes (elements of class ListNode). Each node contains an expression (the head) and, recursively, another list. The special value emptyList, which we have already encountered, terminates every list. The operation head returns the first element of a list node. When provided with an argument, the operation head can be used to modify this first element. The operation tail returns the remainder of the list. Finally the operation isNil returns true if and only if the list is the empty list.

Figure 1.6 shows the method lookup, which is defined in terms of these four operations. The while loop cycles over the list of keys until the end (empty list) is reached. Each key is tested against the argument key, using the equality test provided by the class Symbol. Once a match is found the associated value is returned.

If the entire list of names is searched with no match found, if there is a link to another environment the lookup message is passed to that environment. If there is no link, a null value is returned.

The routine to add a new value to an environment (Figure 1.7) merely attaches a new name and value to the beginning of the respective lists. Note by attaching to be beginning of a list this will hide any existing binding of the name, although such a situation will not often occur. The method set searches for an existing binding, replacing it if found, and only adding the new element

```
void Environment::add(Symbol * s, Expression * v)
{
   theNames = new ListNode(s, (ListNode *) theNames);
   theValues = new ListNode(v, (ListNode *) theValues);
}

void Environment::set(Symbol * sym, Expression * value)
{
   ListNode * nameit = theNames;
   ListNode * valueit = theValues;

   while (! nameit->isNil()) {
      if (*sym == nameit->head()) {
         valueit->head(value);
         return;
         }
      nameit = nameit->tail();
      valueit = valueit->tail();
      }

   // see if we can find it on somebody elses list
   Environment * link = theLink;
   if (link) {
      link->set(sym, value);
      return;
      }

   // not found and we're the end of the line, just add
   add(sym, value);
}
```

Figure 1.7: Methods used to Insert into an environment

to the final environment if no binding can be located.

1.4 Expressions

The class Expression is a root for a class hierarchy that contains the majority of classes defined in these interpreters. Figure 1.8 shows a portion of this class hierarchy. We have already seen that environments are a form of expression, as are integers, symbols, lists and functions.

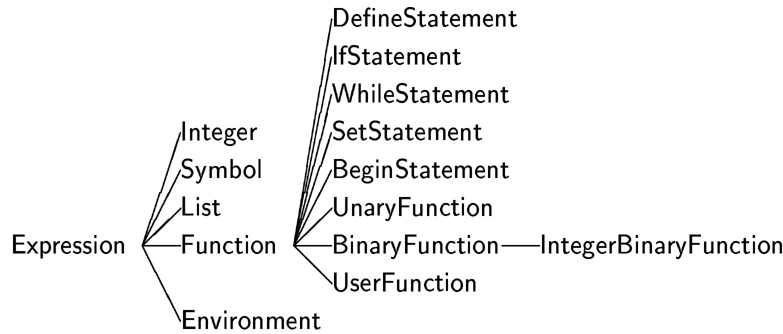

Figure 1.8: The Expression class Hierarchy in Chapter 1

1.4.1 The Abstract Class

The major purposes of the abstract class Expression (Figure 1.9) are to perform memory management functions, to permit conversions from one type to another in a safe manner, and to define protocol for evaluation and printing of expression values. The latter is easist to dismiss. The virtual methods eval and print provide for evaluation and printing of values. The eval method takes as argument a target expression to which the evaluated expression will be assigned, as well as two environments. The first environment contains the list of legal value-ops for the expression, while the second is the more general environment in which the expression is to be evaluated. The default method for eval merely assigns the current expression to the target. This suffices for objects, such as integers, which yield themselves no matter how many times they are evaluated. The default method print, on the other hand, prints an error message. Thus this method should always be overridden in subclasses.

Memory Management

For long running programs it is imperative that memory associated with unused expressions be recovered by the underlying operating system. This is accomplished in these interpreters through the mechanism of reference counts. Every expression contains a reference count field, which is initially set to zero by the constructor in class Expression. The integer value maintained in this field represents the number of pointers that reference the object. When this count becomes zero, no pointers refer to the object and the memory associated with it can be recovered.

The maintenance of reference counts if peformed by the class Expr (Figure 1.10). As with the class Env we have already encountered, the class Expr is a holder class, which maintains an expression pointer. A value can be inserted into an Expr either through construction or the assignment operator. A value can be retrieved either though the protected method val or, as a notational con-

```
class Expression {
private:
    friend class Expr;
    int referenceCount;
public:
    Expression();

    virtual void free();

    // basic object protocol
    virtual void eval(Expr &, Environment *, Environment *);
    virtual void print();

    // conversion tests
    virtual Expression * touch();
    virtual IntegerExpression * isInteger();
    virtual Symbol * isSymbol();
    virtual Function * isFunction();
    virtual ListNode * isList();
    virtual Environment * isEnvironment();
    virtual APLValue * isAPLValue();
    virtual Method * isMethod();
};
```

Figure 1.9: The Class Expression

```
class Expr {
private:
    Expression * value;

protected:
    Expression * val()
        { return value; }

public:
    Expr(Expression * = 0);

    Expression * operator ()()
        { return val(); }

    void operator = (Expression *);

    void evalAndPrint(Environment *, Environment *);
};
```

Figure 1.10: The class Expr

venience, through the parenthesis operator. The method evalAndPrint, as have noted already, merely passes the eval message on to the underlying expression and prints the resulting value.

Figure 1.4.1 gives the implementation of the constructor and assignment operator for class Expr. The constructor takes an optional pointer to an expression, which may be a null expression (the default). If the expression is non-null, the reference count for the expression is incremented. Similarly, the assignment operator first increments the reference count of the new expression. Then it decrements the reference count of the existing expression (if non-null), and if the reference count reaches zero, the memory is released, using the system function delete. Immediately prior to destruction, the virtual method free is invoked. Classes can override this method to provide any necessary class-specific maintenance. For example, the class Environment (Figure 1.5) assigns null values to the structures theNames, theValues and theLink, thereby possibly triggering the release of their storage as well.

Type Conversion

A common difficulty in a statically typed language such as C++ is the *container problem*. Elements placed into a general purpose data structure, such as a list, must have a known type. Generally this is accomplished by declaring such elements as a general type, such as Expression. But in reality such elements are usually instances of a more specific subclass, such as an integer or a symbol. When we remove these values from the list, we would like to be able to recover the original type.

```
Expr::Expr(Expression * val)
{
    value = val;
    if (val) value→referenceCount++;
}

void Expr::operator = (Expression * newvalue)
{
    // increment right hand side of assignment
    if (newvalue) {
        newvalue→referenceCount++;
        }

    // decrement left hand side of assignment
    if (value) {
        value→referenceCount−−;
        if (value→referenceCount = 0) {
            value→free();
            delete value;
            }
        }

    // then do the assignment
    value = newvalue;
}
```

```
Environment * Expression::isEnvironment()
{
   return 0;
}

Environment * Environment::isEnvironment()
{
   return this;
}

Expression * a = new Symbol("test");
Expression * b = new Environment(emptyList, emptyList, 0);

Environment * c = a→isEnvironment();    // will yield null
Environment * d = b→isEnvironment();    // will yield the environment

if (c)
   printf("c is an environment");    // won't happen
if (d)
   printf("d is an environment");    // will happen
```

Figure 1.11: Type safe object test and conversion

There are actually two steps in the solution of this problem. The first step is testing the type of an object, to see if it is of a certain form. The second step is to legally assign the object to a variable declared as the more specific class. In these interpreters the mechanism of virtual methods is used to combine these two functions. In the abstract class Expression a number of virtual functions are defined, such as isInteger and isEnvironment. These are declared as returning a pointer type. The default behavior, as provided by class Expression, is to return a null pointer. In an appropriate class, however, this method is overridden so as to return the current element. That is, the class associated with integers overrides isInteger, the class associated with symbols overrides isSymbol, and so on. Figure 1.11 shows the two definitions of isEnvironment, the first from class Expression and the second from class Environment. By testing whether the result of this method is non-null or not, one can not only test the type of an object but one can assign the value to a specific class pointer without compromising type safety. An example bit of code is provided in Figure 1.11 that illustrates the use of these functions.

The method touch presents a slightly different situation. It is defined in the abstract class to merely return the object to which the message is sent. That is, it is a null-operation. In Chapter **??**, when we introduce delayed evaluation, we will define a type of expression which is not evaluated until it is needed. This expression will override the touch method to force evaluation at that point.

```
class IntegerExpression : public Expression {
private:
   int value;
public:
   IntegerExpression(int v)
      { value = v; }

   virtual void print();
   virtual IntegerExpression * isInteger();

   int val()
      { return value; }
};
```

Figure 1.12: The class IntegerExpression

1.4.2 Integers

Internally within the interpreters integers are represented by the class IntegerExpression (Figure 1.12). The actual integer value is maintained as a private value set as part of the construction process. This value can be accessed via the method val. The only overridden methods are the print method, which prints the integer value, and the isInteger method, which yields the current object.

1.4.3 Symbols

Symbols are used to represent uninterpreted character strings, for example identifier names. Instances of class Symbol (Figure 1.13) maintain the text of their value in a private instance variable. This character pointer can be recovered via the method chars. Storage for this text is allocated as part of the construction process, and deleted by the virtual method free. The equality testing operators return true if the current symbol matches the text of the argument.

Figure 1.13 also shows the implementation of the method eval in the class Symbol. When a symbol is evaluated it is used as a key to index the current environment. If found the (possibly touched) associated value is assigned to the target. If it is not found an error message is generated. The routine error always yields a null expression.

1.4.4 Lists

We have already encountered the behavior of the List data structure (Figure 1.14) in the discussion of environments. As with expressions and environments, lists are represented by a pair of classes. The first, class ListNode, maintains the actual list data. The second, class List, is merely a pointer to a list node, and exists only to provide memory management operations.

```
class Symbol : public Expression {
private:
    char * text;

public:
    Symbol(char *);

    virtual void free();
    virtual void eval(Expr &, Environment *, Environment *);
    virtual void print();
    virtual Symbol * isSymbol();

    int operator == (Expression *);
    int operator == (char *);
    char * chars() { return text; }
};

void Symbol::eval(Expr & target, Environment * valueops, Environment *
rho)
{
    Expression * result = rho→lookup(this);
    if (result)
        result = result→touch();
    else
        result = error("evaluation of unknown symbol: ", text);
    target = result;
}
```

Figure 1.13: The class Symbol

```
class ListNode : public Expression {
protected:
    Expr h;        // the head field
    Expr t;        // the tail field

public:
    ListNode(Expression *, Expression *);

    // overridden methods
    virtual void free();
    virtual void eval(Expr &, Environment *, Environment *);
    virtual void print();
    virtual ListNode * isList();

    // list specific methods
    int isNil();
    int length();
    Expression * at(int);
    virtual Expression * head();
    void head(Expression * x);
    ListNode * tail();
};


class List : public Expr {
public:
    operator ListNode *();
    void operator = (ListNode * r);
};
```

Figure 1.14: The classes List and ListNode

Only one feature of the latter class deserves comment; rather than overloading the parenthesis operator the class List defines a conversion operator which permits instances of class List to be converted without comment to ListNodes. Thus in most cases a List can be used where a ListNode is expected, and the conversion will be implicitly defined. We have seen this already, without having noted the fact, in several places where the variable emptyList (an instance of class List) was used in situations where an instance of class ListNode was required.

The actual list data is maintained in the instance variables h and t, which we have already noted can be retrieved (and, in the case of the h, set) by the methods head and tail. The method length returns the length of a list, and the method at permits a list to be indexed as an array, starting with zero for the head position.

The majority of methods, such as length, at, print, are simple recursive routines, and will not be discussed. Only one method is sufficiently complex to deserve comment, and this is the procedure used to evaluate a list. A list is interpreted as a function call, and thus the evaluation of a list involves finding the indicated function and invoking it, passing as arguments the remainder of the list. These actions are performed by the method eval shown in Figure 1.15. An empty list always evaluates to itself. Otherwise the first argument to the list is examined. If it is a symbol, a test is performed to see if it is one of the value-ops. If it is not found on the value-op list the first element is evaluated, whether or not it is a symbol. Generally this will yield a function value. If so, the method apply, which we will discuss in the next section, is used to invoke the function. If the first argument did not evaluate to a function an error is indicated.

1.5 Functions

If expressions are the heart of the interpreter, then functions are the muscles that keep the heart working. All behavior, statements, valueops, as well as user-defined functions, are implemented as subclasses of class Function (Figure 1.16). As we noted in the last section, when a function (written as a list expression) is evaluated the method apply in invoked. This method takes as argument the target for the evaluation and a list of unevaluated arguments. The default behavior in class Function is to evaluate the arguments, using the simple recursive routine evalArgs, then invoke the method applyWithArgs.

Both the methods apply and applyWithArgs are declared as virtual, and can thus be overridden in subclasses. Those function that do not evaluate their arguments, such as the functions implementing the control structures of Chapter 1, override the apply method. Function that do evaluate their arguments, such as the majority of value-Ops, override the applyWithArgs method.

Two subclasses of Function deserve mention. The class UnaryFunction overrides apply to test that only one argument has been provided. Similarly the class BinaryFunction tests for exactly two arguments. The remaining major subclass of Function is the class UserFunction. We will defer a discussion of this until we examine the implementation of the define statement.

1.6 The Basic Evaluator

We are now in a position to finally describe the characteristics that are unique to the basic evaluator of chapter one. This interpreter recognizes one command (the define statement), several built-in statements (if, while, set, and begin), and a number of value-ops. All are implemented internally as functions. What syntactic category a symbol is associated with is determined by what environment it is placed on, and not by the structure of the function.

```
void ListNode::eval(Expr & target, Environment * valueops, Environment *
rho)
{

    // an empty list evaluates to nil
    if (isNil()) {
        target = this;
        return;
        }

    // if first argument is a symbol, see if it is a valueop
    Expression * firstarg = head();
    Expression * fun = 0;
    Symbol * name = firstarg→isSymbol();
    if (name)
        fun = valueops→lookup(name);

    // otherwise evaluate it in the given environment
    if (! fun)  {
        firstarg→eval(target, valueops, rho);
        fun = target();
        }

    // now see if it is a function
    Function * theFun = 0;
    if (fun) theFun = fun→isFunction();
    if (theFun) {
        theFun→apply(target, tail(), rho);
        }
    else {
        target = error("evaluation of unknown function");
        return;
        }
}
```

Figure 1.15: The method eval from class List

```
class Function : public Expression {
public:
    virtual Function * isFunction();

    virtual min.log

    void apply(Expr &, ListNode *, Environment *);
    virtual void applyWithArgs(Expr &, ListNode *, Environment *);
    virtual void print();

    // isClosure is recognized only by functions
    virtual int isClosure();
};

static ListNode * evalArgs(ListNode * args, Environment * rho)
{
    if (args→isNil())
        return args;
    Expr newhead;
    Expression * first = args→head();
    first→eval(newhead, valueOps, rho);
    return new ListNode(newhead(), evalArgs(args→tail(), rho));
    newhead = 0;
}

void Function::apply(Expr & target, ListNode * args, Environment * rho)
{
    List newargs = evalArgs(args, rho);
    applyWithArgs(target, newargs, rho);
    newargs = 0;
}
```

Figure 1.16: The class Function and the method apply

```
class DefineStatement : public Function {
public:
    virtual void apply(Expr &, ListNode *, Environment *);
};

void DefineStatement::apply(Expr & target, ListNode * args, Environment *
rho)
{
    if (args→length() ≠ 3) {
        target = error("define requires three arguments");
        return;
        }

    Symbol * name = args→at(0)→isSymbol();
    if (! name) {
        target = error("define missing name");
        return;
        }

    ListNode * argNames = args→at(1)→isList();
    if (! argNames) {
        target = error("define missing arg names list");
        return;
        }

    rho→set(name, new UserFunction(argNames, args→at(2), rho));

    // yield as value the name of the function
    target = name;
};
```

Figure 1.17: Implementation of the define statement

1.6.1 The define statement

The define statement is implemented as the single instance of the class DefineStatement (Figure 1.17), entered with the key "define" in the commands environment. The class overrides the virtual method apply, since it must access its arguments before they are evaluated. It tests that the arguments are exactly three in number, and that the first is a symbol and the second a list. If no errors are detected, an instance of the class UserFunction is created and and set in the current (always global) environment.

The class UserFunction created by the define statement is similarly a subclass of class Function (Figure 1.18). User functions maintain in instance variables the list of argument names, the body of the function, and the lexical context in which they are to execute. These values are set by the constructor when the

function is defined, and freed by the virtual method free when no longer needed.

User functions always work with evaluated arguments, and thus they override the method applyWithArgs. The implementation of this method is also shown in Figure 1.18. This method checks that the number of arguments supplied matches the number in the function definition, then creates a new environment to match the arguments and their values. The expression which represents the body of the function is then evaluated. By passing the new context as argument to the evaluation, symbolic references to the arguments will be matched with the appropriate values.

1.6.2 Built-In Statements

The built-in statements if, while, set and begin are each defined by functions entered in the valueOps environment. With the exception of begin, these must capture their arguments before they are evaluated and thus, like define, they override the method apply.

The If statement

The If statement (Figure 1.19) first insures it has three arguments. It then evaluates the first argument. Using the auxiliary function isTrue (which will vary over the different interpreters as our definition of "true" changes) the truth or falsity of the first expression is determined. Depending upon the outcome, either the second or third argument is evaluated to determine the result. In the Chapter 1 interpreter the value 0 is false, and all other values (integer or not) are considered to be true.

The while statement

The function that implements the while statement is shown in Figure 1.20. Although the while statement requires two arguments, it nevertheless cannot usefully be made a subclass of class BinaryFunction, since it must access its arguments before they are evaluated. The implementation of the while statements loops until the first argument evaluates to a true condition, using the same test for true method used by the if statement. The results returned by evaluating the body of the while statement are ignored, as the body is executed just for side effects.

The set statement

The implementation of the set statement is shown in Figure 1.21. The function insures the first argument is a symbol, evaluates the second argument, then sets the binding of the symbol to value in the current environment.

```
class UserFunction : public Function {
protected:
    List argNames;
    Expr body;
    Env context;
public:
    UserFunction(ListNode *, Expression *, Environment *);
    virtual void free();
    virtual void applyWithArgs(Expr &, ListNode *, Environment *);
    virtual int isClosure();
};

void UserFunction::applyWithArgs(Expr& target, ListNode* args,
Environment* rho)
{
    // number of args should match definition
    ListNode *an = argNames;
    if (an->length() != args->length()) {
        error("argument length mismatch");
        return;
        }

    // make new environment
    Env newrho;
    newrho = new Environment(an, args, context);

    // evaluate body in new environment
    Expression * bod = body();
    if (bod)
        bod->eval(target, valueOps, newrho);
    else
        target = 0;

    newrho = 0;    // force memory recovery
}
```

Figure 1.18: The class UserFunction and method of application

```
class IfStatement : public Function {
public:
    virtual void apply(Expr &, ListNode *, Environment *);
};

void IfStatement::apply(Expr & target, ListNode * args, Environment * rho)
{
    if (args→length() ≠ 3) {
        target = error("if statement requires three arguments");
        return;
        }

    Expr cond;
    args→at(0)→eval(cond, valueOps, rho);
    if (isTrue(cond()))
        args→at(1)→eval(target, valueOps, rho);
    else
        args→at(2)→eval(target, valueOps, rho);
    cond = 0;
}

int isTrue(Expression * cond)
{
    IntegerExpression *ival = cond→isInteger();
    if (ival && ival→val() == 0)
        return 0;
    return 1;
}
```

Figure 1.19: The implementation of the If statement

```
void WhileStatement::apply(Expr & target, ListNode * args, Environment *
rho)
{   Expr stmt;

    if (args→length() ≠ 2) {
        target = error("while statement requires two arguments");
        return;
        }

    // grab the two pieces of the statement
    Expression * condexp = args→at(0);
    Expression * stexp = args→at(1);

    // then start the execution loop
    condexp→eval(target, valueOps, rho);
    while (isTrue(target())) {
        // evaluate body
        stexp→eval(stmt, valueOps, rho);
        // but ignore it (and force memory reclamation)
        stmt = 0;
        // then reevaluate condition
        condexp→eval(target, valueOps, rho);
        }
}
```

Figure 1.20: The implementation of the While statement

```
void SetStatement::apply(Expr & target, ListNode * args, Environment * rho)
{
   if (args→length() ≠ 2) {
      target = error("set statement requires two arguments");
      return;
      }

   // get the two parts
   Symbol * sym = args→at(0)→isSymbol();
   if (! sym) {
      target = error("set commands requires symbol for first arg");
      return;
      }

   // set target to value of second argument
   args→at(1)→eval(target, valueOps, rho);

   // set it in the environment
   rho→set(sym, target());
}
```

Figure 1.21: The implementation of the set statement

The begin statement

Unlike the other statements, the begin statement does what to evaluate each of its arguments. Thus it overrides the method applyWithArgs, instead of the method apply. It merely assigns to the target variable the value of the last expression (Figure 1.22).

1.6.3 The value-Ops

The Value-ops are functions placed in the valueop global environment. They can be divided into two categories; there are those that take two integer arguments and produce an integer result (+, −, ∗, /, =, < and >) and those that take a single argument (print).

The implementation of the integer binary functions is simplified by the introduction of an intermediate class IntegerBinaryFunction, a subclass of BinaryFunction (Figure 1.23). The private state for each instance of this class holds a pointer to a function that takes two integer values and generates an integer result. The applyWithArgs method in this class decodes the two integer arguments, then invokes the stored function to produce the new integer value. To implement each of the seven binary integer functions (the relational functions generate 0 and 1 values for true and false, remember) it is only necessary define an appropriate function and pass it as argument to the constructor during initialization of the interpreter. This can be seen in Figure 1.24.

```
void BeginStatement::applyWithArgs(Expr& target, ListNode* args,
Environment* rho)
{
    int len = args→length();

    // yield as result the end of the list
    if (len < 1)
        target = error("begin needs at least one statement");
    else
        target = args→at(len - 1);
}
```

Figure 1.22: The implementation of the begin statement

The print function is implemented by a subclass of UnaryFunction that merely invokes the method print on the argument. All expressions will respond to this method.

1.7 Initializing the Run-Time Environment

Figure 1.24 shows the initialization routine for the interpreters of chapter one. In chapter one there are no global variables defined at the start of execution. There is one command, the statement define, and a number of value-ops.

```
class IntegerBinaryFunction : public BinaryFunction {
private:
    int (∗fun)(int, int);
public:
    IntegerBinaryFunction(int (∗afun)(int, int));
    virtual void applyWithArgs(Expr &, ListNode ∗, Environment ∗);
    virtual int value(int, int);
};

void IntegerBinaryFunction::applyWithArgs(Expr& target, ListNode  args,
        Environment∗ rho)
{
    Expression ∗ left = args→at(0);
    Expression ∗ right = args→at(1);
    if ((! left→isInteger()) || (! right→isInteger())) {
        target = error("arithmetic function with nonint args");
        return;
        }

    target =  new IntegerExpression(
        fun(left→isInteger()→val(), right→isInteger()→val()));
}

int PlusFunction(int a, int b) { return a + b; }
int MinusFunction(int a, int b) { return a − b; }
int TimesFunction(int a, int b) { return a ∗ b; }
int DivideFunction(int a, int b)
{
    if (b ≠ 0)
        return a / b;
    error("division by zero");
    return 0;
}
int IntEqualFunction(int a, int b) { return a == b; }
int LessThanFunction(int a, int b) { return a < b; }
int GreaterThanFunction(int a, int b) { return a > b; }
```

Figure 1.23: Implementation of the Arithmetic Functions

```
initialize()
{
    // create the reader/parser
    reader = new Reader;

    // initialize the commands environment
    Environment * cmds = commands;
    cmds→add(new Symbol("define"), new DefineStatement);

    // initialize the value-ops environment
    Environment * vo = valueOps;
    vo→add(new Symbol("if"), new IfStatement);
    vo→add(new Symbol("while"), new WhileStatement);
    vo→add(new Symbol("set"), new SetStatement);
    vo→add(new Symbol("begin"), new BeginStatement);
    vo→add(new Symbol("+"), new IntegerBinaryFunction(PlusFunction));
    vo→add(new Symbol("-"), new IntegerBinaryFunction(MinusFunction));
    vo→add(new Symbol("*"), new IntegerBinaryFunction(TimesFunction));
    vo→add(new Symbol("/"), new IntegerBinaryFunction(DivideFunction));
    vo→add(new Symbol("="), new IntegerBinaryFunction(IntEqualFunction));
    vo→add(new Symbol("<"), new IntegerBinaryFunction(LessThanFunction));
    vo→add(new Symbol(">"), new IntegerBinaryFunction(GreaterThanFunction));
    vo→add(new Symbol("print"), new PrintFunction);
}
```

Figure 1.24: Initialization of the Basic Evaluator