The Kamin Interpreters in C++

Tim Budd

November 13, 2018

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

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

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 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.

[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

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 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.

```
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

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 Expression 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 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.

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

```
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

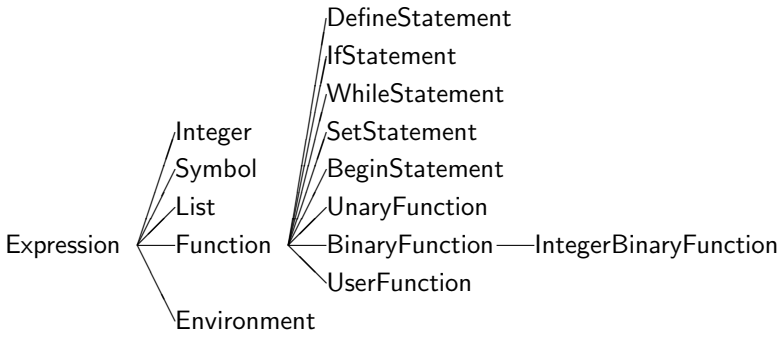

Figure 1.8: The Expression class Hierarchy in Chapter 1

```
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

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 convenience, 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.

```
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;
}
```

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.

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 5, 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.

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

```
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

```
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

```
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

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.

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

```
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

```
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

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.

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

```
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

```
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

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.

```
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

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.

```
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 de-

```
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

fine an appropriate function and pass it as argument to the constructor during initialization of the interpreter. This can be seen in Figure 1.24.

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

Chapter 2

The Lisp Interpreter

The interpreter for Lisp differs only slightly from that of Chapter one. The reader/parser is modified so as to recognize quoted constants, two new global variables (T and nil) are added, and a number of new value-ops are defined. In all other respects it is the same. Figure 2.1 shows the class hierarchy for the expression classes added in chapter 2.

2.1 The Lisp reader

The Lisp reader is created by subclassing from the base class Reader (Figure 2.2). The only change is to modify the method readExpression to check for leading quote marks. If no mark is found, execution is as in the default case. If a quote mark is found, the character pointer is advanced and the following expression is turned into a quoted constant. Note that no checking is performed on this expression. This permits symbols, even separators, to be treated as data. That is, '; is a quoted symbol, even though the semicolon itself is not a legal symbol.

To create quoted constants it is necessary to introduce a new type of expression. When an instance of class QuotedConst is evaluated, it simply returns its (unevaluated) data value.

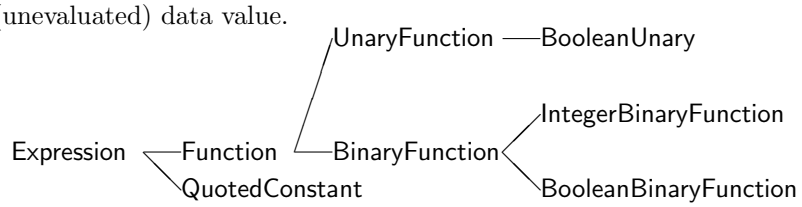

Figure 2.1: Classes added in Chapter Two

```
class QuotedConst : public Expression {
private:
   Expr theValue;
public:
   QuotedConst(Expression * val)
      { theValue = val; }

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

class LispReader : public Reader {
protected:
   virtual Expression * readExpression();
};

void QuotedConst::eval(Expr &target, Environment *, Environment *)
{
   target = theValue();
}

void QuotedConst::print()
{
   printf("'"); theValue()->print();
}

Expression * LispReader::readExpression()
{
   // if quoted constant, return it,
   if ((*p == '\'') || (*p == '`')) {
      p++;
      return new QuotedConst(readExpression());
      }
   // otherwise simply return what we had before
   return Reader::readExpression();
}
```

Figure 2.2: The Lisp reader/parser

2.2 Value-ops

In addition to adding a number of new value-ops, the Lisp interpreter modifies the meaning of a few of the Chapter 1 functions. For example the relational operators must now return the values T or nil, and not 1 and 0 values. Similarly the meaning of *true* and *false* used by the if and while statements is changed. Finally the equality testing function (=) must now recognize both symbols and integers.

2.2.1 Relationals

Figure 2.3 shows the revised definition of the equality testing function, which now must be prepared to handle symbols and well as integers.

Implementation of the boolean binary functions is simplified by the introduction of a class BooleanBinaryFunction (Figure 2.4). This class decodes the two integer arguments and invokes a further method to determine the boolean result. Based on this result either the value of the global symbol representing true or the symbol representing false is returned.

Finally Figure 2.4 shows the revised function used by if and while statements to determine the truth or falsity of their condition. Unlike in Chapter 1, where 0 and 1 were used to represent true and false, here nil is used as the only false value.

2.2.2 Car, Cdr and Cons

Car and cdr are implemented as simple unary functions (Figure 2.5), and cons is a simple binary function that creates a new ListNode out of its two arguments.[1]

2.2.3 Predicates

The implementation of the predicates number?, symbol?, list? and null? is simplified by the creation of a class BooleanUnary (Figure 2.6), subclassing UnaryFunction. As with the integer functions implemented in chapter 1, instances of BooleanUnary maintain as part of their state a function that takes an expression and returns an integer (that is, boolean) value. Thus for each predicate it is only necessary to write a function which takes the single argument and returns a true/false indication.

[1] A matter for debate is whether Cons should give an error if the second argument is not a list. Real Lisp doesn't care; but also uses a different format for printing such lists. Our interpreter prints such as lists exactly as if the second argument had been a list containing the element.

```
class EqualFunction : public BinaryFunction {
public:
   virtual void applyWithArgs(Expr&, ListNode *, Environment *);
};

void EqualFunction::applyWithArgs(Expr& target, ListNode * args,
       Environment *rho)
{
   Expression * one = args->at(0);
   Expression * two = args->at(1);

   // true if both numbers and same number
   IntegerExpression * ione = one->isInteger();
   IntegerExpression * itwo = two->isInteger();
   if (ione && itwo && (ione->val() == itwo->val())) {
      target = true();
      return;
      }

   // or both symbols and same symbol
   Symbol * sone = one->isSymbol();
   Symbol * stwo = two->isSymbol();
   if (sone && stwo && (*sone == stwo)) {
      target = true();
      return;
      }

   // or both lists and both nil
   ListNode * lone = one->isList();
   ListNode * ltwo = two->isList();
   if (lone && ltwo && lone->isNil() && ltwo->isNil()) {
      target = true();
      return;
      }

   // false otherwise
   target = false();
}
```

Figure 2.3: The revised Definition of the equality function

```
class BooleanBinaryFunction : public BinaryFunction {
private:
   int (*fun)(int, int);
public:
   BooleanBinaryFunction(int (*thefun)(int, int)) { fun = thefun; }
   virtual void applyWithArgs(Expr&, ListNode*, Environment*);
   virtual int value(int, int);
};

void BooleanBinaryFunction::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;
      }

   if (value(left->isInteger()->val(), right->isInteger()->val()))
      target = true();
   else
      target = false();
}

int LessThanFunction::value(int a, int b) { return a < b; }
int GreaterThanFunction::value(int a, int b) { return a > b; }

int isTrue(Expression * cond)
{
   // the only thing false is nil
   ListNode *nval = cond->isList();
   if (nval && nval->isNil())
      return 0;
   return 1;
}
```

Figure 2.4: Returning boolean results from relationals

```
void CarFunction(Expr & target, Expression * arg)
{
   ListNode * thelist = arg->isList();
   if (! thelist) {
      target = error("car applied to non list");
      return;
      }
   target = thelist->head()->touch();
}

void CdrFunction(Expr & target, Expression * arg)
{
   ListNode * thelist = arg->isList();
   if (! thelist) {
      target = error("car applied to non list");
      return;
      }
   target = thelist->tail()->touch();
}

void ConsFunction(Expr & target, Expression * left, Expression * right)
{
   target = new ListNode(left, right);
}
```

Figure 2.5: Implementation of Car, Cdr and Cons

```
class BooleanUnary : public UnaryFunction {
private:
    int (*fun)(Expression *);
public:
    BooleanUnary(int (*thefun)(Expression *);
    virtual void applyWithArgs(Expr& target, ListNode* args, Environment*);
};

void BooleanUnary::applyWithArgs(Expr & target, ListNode * args, Environment
*)
{
    if (fun(args→head()))
        target = true();
    else
        target = false();
}

int NumberpFunction(Expression * arg)
{
    return 0 != arg→isInteger();
}

int SymbolpFunction(Expression * arg)
{
    return 0 != arg→isSymbol();
}

int ListpFunction(Expression * arg)
{
    ListNode * x = arg→isList();
    // list? doesn't return true on nil
    if (x && x→isNil()) return 0;
    if (x) return 1;
    return 0;
}

int NullpFunction(Expression * arg)
{
    ListNode * x = arg→isList();
    return x && x→isNil();
}
```

Figure 2.6: The class Boolean Unary

2.3 Initialization of the Lisp Interpreter

Figure 2.7 shows the initialization method for the Lisp interpreter.

```
initialize()
{

    // create the reader/parser
    reader = new LispReader;

    // initialize the global environment
    Symbol * truesym = new Symbol("T");
    true = truesym;
    false = emptyList();
    Environment * genv = globalEnvironment;
    // make T evaluate to T always
    genv→add(truesym, truesym);
    genv→add(new Symbol("nil"), emptyList());

    // 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 BinaryFunction(EqualFunction));
    vo→add(new Symbol("<"), new BooleanBinaryFunction(LessThanFunction));
    vo→add(new Symbol(">"), new
BooleanBinaryFunction(GreaterThanFunction));
    vo→add(new Symbol("cons"), new BinaryFunction(ConsFunction));
    vo→add(new Symbol("car"), new UnaryFunction(CarFunction));
    vo→add(new Symbol("cdr"), new UnaryFunction(CdrFunction));
    vo→add(new Symbol("number?"), new BooleanUnary(NumberpFunction));
    vo→add(new Symbol("symbol?"), new BooleanUnary(SymbolpFunction));
    vo→add(new Symbol("list?"), new BooleanUnary(ListpFunction));
    vo→add(new Symbol("null?"), new BooleanUnary(NullpFunction));
    vo→add(new Symbol("print"), new UnaryFunction(PrintFunction));
}
```

Figure 2.7: Initialization of the Lisp interpreter

Chapter 3

The APL Interpreter

My version of the APL interpreter differs somewhat from that provided by Kamin:

- My version will recognize arbitrary rank (dimension) arrays, not simply scalar, vector and two dimensional arrays. (Although currently it is only able to print those three types).
- The C++ version of the interpreter recognizes vector constants without the necessity for quoting them, as in (resize (3 4) (indx 12)).
- I have eliminated the if and while statements, thus forcing programmers into a more "APL" style of thought.
- My version of catenation works now for values of arbitrary dimensionality. (Transpose and print are the only two functions that limit the dimensionality of their arguments).

Despite the APL interpreter being larger than any other interpreter, I think that the addition of a few more functions could give the student an even better feel for the language, as well as providing a smooth transition to functional programming. Specifically, I think reduction should be defined as a functional, and inner and outer products added as operations. I have not done this as yet, however.

Figure 3.1 shows the class hierarchy for the classes introducted in this chapter.

3.1 APL Values

The APL interpreter manipulates APL values, which are defined by the data type APLValue (Figure 3.2). An APL value represents a integer rectilinear array.

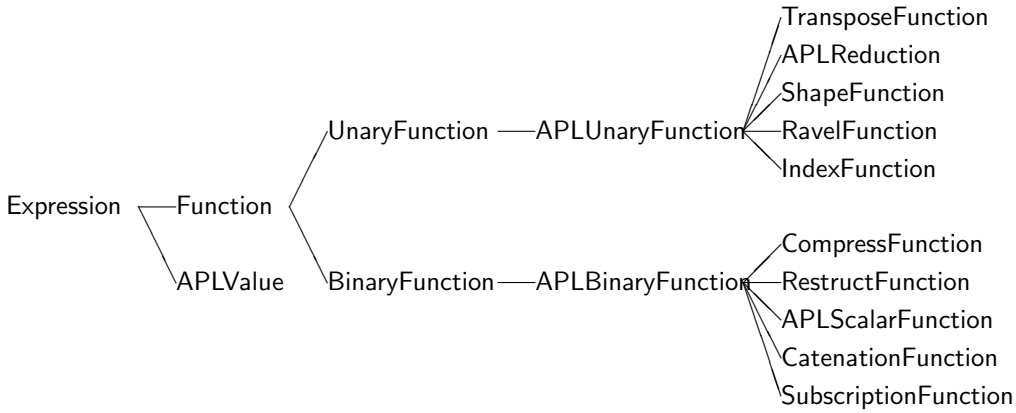

Figure 3.1: The APL interpreter class hierarchy

Internally, such a value is represented by a list that maintains the shape (extent along each dimension) and a vector of integer values. The length of the shape list provides the rank (dimensionality) of the data value. The product of the values in the shape indicates the number of elements in the array, except in the case of scalar values, which have an empty shape array.

APL values are stored in what is called *ravel-order*. This is what in some other languages is called row-major order.

The methods defined for APL values can be used to determine the number of elements contained in the structure (size), obtain the shape of the value (shape), obtain the shape at any given dimension (shapeAt), obtain the value at any given ravel-order position (at), and finally change the value at any position (atPut).

3.2 The APL Reader

The APL reader is modified so that individual scalar values and vectors of integers are recognized as APL values. The class definition for APLreader is shown in Figure 3.3, and the code for the two auxiliary functions in the next figure.

3.3 APL Functions

The implementation of the APL functions is simplified by the addition of two auxiliary classes, APLUnary and APLBinary. In addition to checking that the right number of arguments are provided to a function application, these check to insure that the arguments are APL values[1] and invoke yet another virtual

[1]A largely gratuitous move, since the user has no way of creating anything other than an APL value. Still, it doesn't do any harm to be careful.

```
class APLValue : public Expression {
private:
   List shapedata;
   int * data;
public:
   APLValue(ListNode *, int);

   // the overridden methods
   virtual APLValue * isAPLValue();
   virtual void free();
   virtual void print();

   // methods unique to apl values
   int size();
   ListNode * shape();
   int shapeAt(int);
   int at(int pos);
   void atPut(int pos, int val);
};
```

Figure 3.2: The Representation for APL Values

function applyOp, to perform the actual calculation.

3.3.1 Scalar Functions

By far the largest class of APL functions are the so-called *scalar functions*. These are the conventional arithmetic and logical functions, such as addition and multiplication, extended in the natural way to arrays. The only complication in the implementation of these values concerns what is called *scalar extension*. That is, a scalar value can be used as either the left or right argument to a scalar function, and it is treated as if it were an entire array of the correct dimensionality to match the other argument. Since scalar extension can occur with either the left or right argument, the code for scalar functions divides naturally into three cases.

Scalar functions are implemented using a single class by making use, as we have done before, of an instance variable that contains a pointer to a integer function that generates an integer result. The class APLscalarFunction and the method applyOp are shown in Figure 3.5. Note that the same functions used in the previous interpreters can be used in the construction of the APL scalar functions.

```
class APLreader : public LispReader {
protected:
    virtual Expression * readExpression();
private:
    APLValue * readAPLscalar(int);
    APLValue * readAPLvector(int);
};

Expression * APLreader::readExpression()
{
    // see if it is a scalar value
    if ((*p == '-') && isdigit(*(p+1))) {
        p++;
        return readAPLscalar( - readInteger());
        }

    if (isdigit(*p))
        return readAPLscalar(readInteger());

    // see if it is a vector constant
    if (*p == '(') {
        p++;
        skipNewlines();
        if (isdigit(*p))
            return readAPLvector(0);
        return readList();
        }

    // else default
    return LispReader::readExpression();
}
```

Figure 3.3: The APL reader

```
APLValue * APLreader::readAPLscalar(int d)
{
    // read a scalar value, but make it an APL value
    APLValue * newval = new APLValue(emptyList, 1);
    newval→atPut(0, d);
    return newval;
}

APLValue * APLreader::readAPLvector(int size)
{
    skipNewlines();

    // if at end of list, make new vector
    if (*p == ')') {
        p++;
        return new APLValue(
            new ListNode(new IntegerExpression(size), emptyList()),
            size);
        }

    // else we better have a digit, save it and get the rest
    int sign = 1;
    if (*p == '-') { sign = -1; p++; }
    if (! isdigit(*p))
        error("ill formed apl vector constant");
    int val = sign * readInteger();
    APLValue * newval = readAPLvector(size + 1);
    newval→atPut(size, val);
    return newval;
}
```

Figure 3.4: The APL reader functions

```
void APLScalarFunction::applyOp(Expr& target, APLValue* left, APLValue*
right)
{
   if (left→size() == 1) {    // scalar extension of left
      int extent = right→size();
      APLValue * newval = new APLValue(right→shape(), extent);
      int lvalue = left→at(0);
      while (−−extent >= 0)
         newval→atPut(extent, fun(lvalue, right→at(extent)));
      target = newval;
      }
   else if (right→size() == 1) {    // scalar extension of right
      int extent = left→size();
      APLValue * newval = new APLValue(left→shape(), extent);
      int rvalue = right→at(0);
      while (−−extent >= 0)
         newval→atPut(extent, fun(left→at(extent), rvalue));
      target = newval;
      }
   else {              // conforming arrays
      int extent = left→size();
      if (extent != right→size()) {
         target = error("conformance error on scalar function");
         return;
         }
      for (int i = left→shape()→length(); −−i >= 0; )
         if (left→shapeAt(i) != right→shapeAt(i)) {
            target =
            error("conformance error on scalar function");
            return;
            }

      APLValue * newval = new APLValue(left→shape(), extent);
      while (−−extent >= 0)
         newval→atPut(extent,
            fun(left→at(extent), right→at(extent)));
      target = newval;
      }
}
```

Figure 3.5: APL Scalar Functions

3.3.2 Reduction

For each scalar function there is an associated reduction function.[2] Reduction in these interpreters always occurs along the last dimension. Thus to compute the size of a new value is suffices to remove the last dimension value. This also simplifies the generation of the new values, since the argument array can be processed in units as long as the final dimension. As with the scalar functions, there is one class defined for all the reductions, with each instance of this class maintaining the particular scalar function being used for the reduction operations. Figure 3.6 shows the code used in computing the APL reduction function.

3.3.3 Compression

Compression, like reduction, operates on the last dimension of a higher order array, changing its extent to that of the number of one elements in the left-argument vector. The length of the left argument vector must match the extent of the last dimension of the right argument. The compression function (Figure 3.7) first computes the number of one elements in the left argument, then iterates over the right argument generating the new values.

3.3.4 Shape and Reshape

The shape function merely copies the size on its argument into a new APL value. The reshape function (restruct) generates a new value with a size given by the left argument, which must be a vector, using elements from the right argument, recycling over the ravel ordering of the right argument multiple times if necessary. The implementation of these functions is shown in Figure 3.8.

3.3.5 Ravel and Index

The ravel function (Figure 3.9) merely takes an argument of arbitrary dimensionality and returns the values as a vector. The index function (called iota in real APL) takes a scalar value and returns a vector of numbers from 1 to the argument value.

3.3.6 Catenation

The catenation function joins two arrays along their last dimension. They must match in all other dimensions. To build the new result first a row from the first array is copies into the final array, then a row from the second array, then another row from the first, followed by another row from the second, and so on until all rows from each argument have been used.

[2]The statement is true of real APL. The Kamin interpreters do not implement reductions with relational operators, which are, however, not particularly useful.

```
static int lastSize(ListNode * sz)
{
    int i = sz->length();
    if (i > 0) {
        IntegerExpression * ie = sz->at(i-1)->isInteger();
        if (ie)
            return ie->val();
        }
    return 1;
}

static ListNode * removeLast(ListNode * sz)
{
    ListNode * newsz = emptyList;
    int i = sz->length()-1;
    while (--i >= 0)
        newsz = new ListNode(sz->at(i), newsz);
    return newsz;
}

void APLReduction::applyOp(Expr & target, APLValue * arg)
{
    // compute the size of the new expression
    int rowextent = lastSize(arg->shape());
    int extent = arg->size() / rowextent;
    APLValue * newval = new APLValue(removeLast(arg->shape()), extent);

    while (--extent >= 0) {
        int start = (extent + 1) * rowextent - 1;
        int newint = arg->at(start);
        for (int i = rowextent - 2; i >= 0; i--)
            newint = fun(arg->at(--start), newint);
        newval->atPut(extent, newint);
        }

    target = newval;
}
```

Figure 3.6: Implementation of the APL reduction function

```
static ListNode * replaceLast(ListNode * sz, int i)
{
    ListNode *nz = new ListNode(new IntegerExpression(i), emptyList());
    for (i = sz→length() - 1; --i >= 0; )
        nz = new ListNode(sz→at(i), nz);
    return nz;
}

void CompressionFunction::applyOp(Expr& target, APLValue* left, APLValue*
right)
{
    if (left→shape()→length() >= 2) {
        target = error("compression requires vector left arg");
        return;
        }
    int lsize = left→size();    // works for both scalar and vec
    int rsize = lastSize(right→shape());
    if (lsize != rsize) {
        target = error("compression conformability error");
        return;
        }
    // compute the number of non-zero values
    int i, nsize;
    nsize = 0;
    for (i = 0; i < lsize; i++)
        if (left→at(i)) nsize++;

    // now compute the new size
    int rextent = right→size();
    int extent = (rextent / lsize) * nsize;

    APLValue * newval = new APLValue(replaceLast(right→shape(), nsize),
              extent);

    // now fill in the values
    int index = 0;
    for (i = 0; i <= rextent; i++)
        if (left→at(i % lsize))
            newval→atPut(index++, right→at(i));
    target = newval;
}
```

Figure 3.7: The Compression function

```
void ShapeFunction::applyOp(Expr & target, APLValue * arg)
{
    int extent = arg→shape()→length();
    ListNode * newshape = new ListNode(new IntegerExpression(extent),
            emptyList());
    APLValue * newval = new APLValue(newshape, extent);
    while (--extent >= 0) {
        IntegerExpression * ie = arg→shape()→at(extent)→isInteger();
        if (ie)
            newval→atPut(extent, ie→val());
        else
            target = error("impossible case in Shapefunction");
        }
    target = newval;
};

void RestructFunction::applyOp(Expr & target, APLValue * left, APLValue *
right)
{
    int llen = left→shape()→length();
    if (llen >= 2) {
        target = error("restruct requires vector left arg");
        return;
        }
    llen = left→size();    // works for either scalar or vector
    int extent = 1;
    ListNode * newShape = emptyList;
    while (--llen >= 0) {
        newShape = new ListNode(new IntegerExpression(left→at(llen)),
            newShape);
        extent *= left→at(llen);
        }
    APLValue * newval = new APLValue(newShape, extent);
    int rsize = right→size();
    while (--extent >= 0)
        newval→atPut(extent, right→at(extent % rsize));
    target = newval;
}
```

Figure 3.8: The shape and reshape functions

```
void RavelFunction::applyOp(Expr & target, APLValue * arg)
{
    int extent = arg->size();
    APLValue * newval = new APLValue(extent);
    while (--extent >= 0)
        newval->atPut(extent, arg->at(extent));
    target = newval;
}

void IndexFunction::applyOp(Expr & target, APLValue * arg)
{
    if (arg->size() != 1) {
        target = error("index function requires scalar argument");
        return;
        }
    int extent = arg->at(0);
    APLValue * newval = new APLValue(extent);
    while (--extent >= 0)
        newval->atPut(extent, extent + 1);
    target = newval;
}
```

Figure 3.9: Ravel and Index

```
void CatenationFunction::applyOp(Expr& target, APLValue* left, APLValue*
right)
{
    ListNode * lshape = left→shape();
    ListNode * rshape = right→shape();
    int llen = lshape→length();
    int rlen = rshape→length();
    if (llen <= 0 || (llen != rlen)) {
        target = error("catenation conformability error");
        return;
        }

    // get the size of the last row in each structure
    int lrow, rrow;
    IntegerExpression * ie = lshape→at(llen−1)→isInteger();
    if (ie)
        lrow = ie→val();
    else
        lrow = 1;
    ie = rshape→at(rlen−1)→isInteger();
    if (ie)
        rrow = ie→val();
    else
        rrow = 1;

    // build up the new size
    int extent = lrow + rrow;
    ListNode * newShape = new ListNode(
        new IntegerExpression(extent), emptyList());
    llen = llen − 1;
    while (−−llen >= 0) {
        newShape = new ListNode(lshape→at(llen), newShape);
        ie = lshape→at(llen)→isInteger();
        if (ie)
            extent *= ie→val();
        }

    APLValue * newval = new APLValue(newShape, extent);

    // now build the new values
    int i, index, lindex, rindex;
    index = lindex = rindex = 0;
    while (index < extent) {
        for (i = 0; i < lrow; i++)
            newval→atPut(index++, left→at(lindex++));
        for (i = 0; i < rrow; i++)
            newval→atPut(index++, right→at(rindex++));
        }

    target = newval;
}
```

Figure 3.10: Implementation of the Catenation function

```
void TransposeFunction::applyOp(Expr& target, APLValue * arg)
{
    // transpose of vectors or scalars does nothings
    if (arg→shape()→length() != 2) {
        target = arg;
        return;
        }

    // get the two extents
    int lim1 = arg→shapeAt(0);
    int lim2 = arg→shapeAt(1);

    // build new shapes
    ListNode * newShape =
        new ListNode(arg→shape()→at(1),
        new ListNode(arg→shape()→at(0), emptyList()));
    APLValue * newval = new APLValue(newShape, lim1 * lim2);

    // now compute the values
    for (int i = 0; i < lim2; i++)
        for (int j = 0; j < lim2; j++)
            newval→atPut(i * lim1 + j,
                arg→at(j * lim2 + i));

    target = newval;
}
```

Figure 3.11: The Transpose Function

3.3.7 Transpose

While real APL defines transpose for arbitrary dimension arrays, the transpose presented here works only for arrays of dimension two or less. For vector and scalars the transpose does nothing. Thus the only code required (Figure 3.11) is to take the transpose of a two dimensional array.

3.3.8 Subscription

The Pascal interpreter provided by Kamin applies subscription to the first dimension of a multidimension value. In order to be consistent with the other functions, my version does subscription along the last dimension. Neither is exactly the same as the real APL version. The subscription code is shown in Figure 3.12.

```
void SubscriptFunction::applyOp(Expr& target, APLValue *left, APLValue *right)
{
    if (right→shape()→length() >= 2) {
        target = error("subscript requires vector second arg");
        return;
        }
    int rsize = right→size();
    int lsize = lastSize(left→shape());
    int extent = (left→size() / lsize) * rsize;

    APLValue * newval = new APLValue(replaceLast(left→shape(), rsize),
        extent);

    for (int i = 0; i < extent; i++)
        newval→atPut(i, left→at(
            (i / rsize) * lsize + (right→at(i % rsize)−1)));
    target = newval;
}
```

Figure 3.12: The Subscription function

3.4 Initialization of the APL interpreter

The initialization code for the APL interpreter is shown in Figure 3.13.

```
initialize()
{

    // initialize global variables
    reader = new APLreader;

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

    // initialize the value ops environment
    Environment * vo = valueOps;
    vo→add(new Symbol("set"), new SetStatement);
    vo→add(new Symbol("+"), new APLScalarFunction(PlusFunction));
    vo→add(new Symbol("-"), new APLScalarFunction(MinusFunction));
    vo→add(new Symbol("*"), new APLScalarFunction(TimesFunction));
    vo→add(new Symbol("/"), new APLScalarFunction(DivideFunction));
    vo→add(new Symbol("max"), new APLScalarFunction(scalarMax));
    vo→add(new Symbol("or"), new APLScalarFunction(scalarOr));
    vo→add(new Symbol("and"), new APLScalarFunction(scalarAnd));
    vo→add(new Symbol("="), new APLScalarFunction(scalarEq));
    vo→add(new Symbol("<"), new APLScalarFunction(LessThanFunction));
    vo→add(new Symbol(">"), new APLScalarFunction(GreaterThanFunction));
    vo→add(new Symbol("+/"), new APLReduction(PlusFunction));
    vo→add(new Symbol("-/"), new APLReduction(MinusFunction));
    vo→add(new Symbol("*/"), new APLReduction(TimesFunction));
    vo→add(new Symbol("//"), new APLReduction(DivideFunction));
    vo→add(new Symbol("max/"), new APLReduction(scalarMax));
    vo→add(new Symbol("or/"), new APLReduction(scalarOr));
    vo→add(new Symbol("and/"), new APLReduction(scalarAnd));
    vo→add(new Symbol("compress"), new CompressionFunction);
    vo→add(new Symbol("shape"), new ShapeFunction);
    vo→add(new Symbol("ravel"), new RavelFunction);
    vo→add(new Symbol("restruct"), new RestructFunction);
    vo→add(new Symbol("cat"), new CatenationFunction);
    vo→add(new Symbol("indx"), new IndexFunction);
    vo→add(new Symbol("trans"), new TransposeFunction);
    vo→add(new Symbol("[]"), new SubscriptFunction);
    vo→add(new Symbol("print"), new UnaryFunction(PrintFunction));
}
```

Figure 3.13: APL interpreter initialization

Chapter 4

The Scheme Interpreter

After all the code required to generate the APL interpreter of Chapter 3, the Scheme interpreter is simplicity in itself. Of course, this has more to do with the similarity of Scheme to the basic Lisp interpreter of Chapter 2 than with any differences between APL and Scheme.

To implement Scheme it is only necessary to provide an implementation of the lambda function. This is accomplished by the class Lambda, shown in Figure 4.1. The actual implementation of lambda uses the same class UserFunction we have seen in previous chapters.

Initialization of the Scheme interpreter differs slightly from the code used to initialize the Lisp interpreter (Figure 4.2). The define command is no longer recognized, having been replaced by the set/lambda pair. The built-in arithmetic functions are now considred to be global symbols, and not value-ops. Indeed, there are no comands or value-ops in this language.

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

void LambdaFunction::apply(Expr & target, ListNode * args, Environment * rho)
{
   if (args→length() != 2) {
      target = error("lambda requires two arguments");
      return;
      }

   ListNode * argNames = args→head()→isList();
   if (! argNames) {
      target = error("lambda requires list of argument names");
      return;
      }

   target = new UserFunction(argNames, args→at(1), rho);
}
```

Figure 4.1: The class Lambda

```
initialize()
{

    // initialize global variables
    reader = new LispReader;

    // initialize the value of true
    Symbol * truesym = new Symbol("T");
    true = truesym;
    false = emptyList();

    // initialize the command environment
    // there are no command or value-ops as such in scheme

    // initialize the global environment
    Environment * ge = globalEnvironment;
    ge→add(new Symbol("if"), new IfStatement);
    ge→add(new Symbol("while"), new WhileStatement);
    ge→add(new Symbol("set"), new SetStatement);
    ge→add(new Symbol("begin"), new BeginStatement);
    ge→add(new Symbol("+"), new IntegerBinaryFunction(PlusFunction));
    ge→add(new Symbol("-"), new IntegerBinaryFunction(MinusFunction));
    ge→add(new Symbol("*"), new IntegerBinaryFunction(TimesFunction));
    ge→add(new Symbol("/"), new IntegerBinaryFunction(DivideFunction));
    ge→add(new Symbol("="), new BinaryFunction(EqualFunction));
    ge→add(new Symbol("<"), new BooleanBinaryFunction(LessThanFunction));
    ge→add(new Symbol(">"), new
BooleanBinaryFunction(GreaterThanFunction));
    ge→add(new Symbol("cons"), new BinaryFunction(ConsFunction));
    ge→add(new Symbol("car"), new UnaryFunction(CarFunction));
    ge→add(new Symbol("cdr"), new UnaryFunction(CdrFunction));
    ge→add(new Symbol("number?"), new BooleanUnary(NumberpFunction));
    ge→add(new Symbol("symbol?"), new BooleanUnary(SymbolpFunction));
    ge→add(new Symbol("list?"), new BooleanUnary(ListpFunction));
    ge→add(new Symbol("null?"), new BooleanUnary(NullpFunction));
    ge→add(new Symbol("primop?"), new BooleanUnary(PrimoppFunction));
    ge→add(new Symbol("closure?"), new BooleanUnary(ClosurepFunction));
    ge→add(new Symbol("print"), new UnaryFunction(PrintFunction));
    ge→add(new Symbol("lambda"), new LambdaFunction);
    ge→add(truesym, truesym);
    ge→add(new Symbol("nil"), emptyList());
}
```

Figure 4.2: Initialization of the Scheme Interpreter

Chapter 5

The SASL interpreter

The SASL interpreter is largely constructed by removing features from the Scheme interpreter, such as while loops and so on, and changing the implementation of the cons function to add delayed evaluation. Figure 5.1 shows the class hierarchy for the classes added in this chapter.

5.0.1 Thunks

Delayed evaluation is provided by adding a new expression type, called the *thunk*. Figure 5.2 shows the data structure used to represent this type of value. Every thunk maintains a boolean value indicating whether the thunk has been evaluated yet, an expression (representing either the unevaluated or evaluated expression, depending upon the state of the boolean flag), and a context in which the expression is to be evaluated. Thunks print either as three dots, if they have not yet been evaluated, or as the printed representation of their value, if they have.

Here we finally see an overridden definition for the method *touch*. You will recall that this method was defined in Chapter 1, and that all other expressions merely return their value as the result of this expression. Thunks, on the other hand, will evaluate themselves if touched, and then return their new evaluated result. With the addition of this feature many of the definitions we have presented in earlier chapters, such as the definitions of car and cdr, hold equally well when given thunks as arguments.

Since thunks can represent lists, symbols, integers and so on, the predicate methods isSymbol and the like must be redefined as well. If the thunk represents an evaluated value, these simply return the result of testing that value (Figure 5.3).

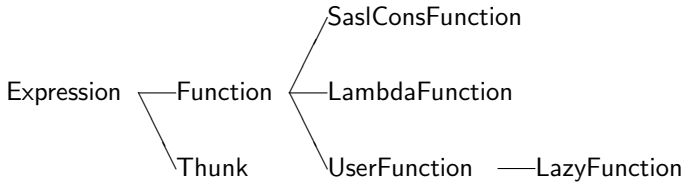

Figure 5.1: Class Hierarchy for expressions in the SASL interpreter

5.1 Lazy Cons

The SASL cons function differs from the Scheme version in producing a list node containing a pair of thunks, rather than a pair of values (Figure 5.4). Class SaslConsFunction must now be a subclass of Function and not BinaryFunction, because it must grab its arguments before they are evaluated. Thus it must itself check to see that only two arguments are passed to the function.

5.2 Lazy User Functions

User defined functions must be provided with lazy evaluation semantics as well. This is accomplished by defining a new class LazyFunction (Figure 5.5). Lazy functions act just like user functions from previous chapters, only they do not evaluate their arguments. Thus the function body is evaluated by the method apply, rather than passing the evaluated arguments on to the method applyWithArgs. The lambda function from the previous chapter is modified to produce an instance of LazyFunction, rather than UserFunction.

```
class Thunk : public Expression {
private:
    int evaluated;
    Expr value;
    Env context;
public:
    Thunk(Expression *, Environment *);

    virtual void free();
    virtual void print();
    virtual Expression * touch();
    virtual void eval(Expr &, Environment *, Environment *);

    virtual IntegerExpression * isInteger();
    virtual Symbol * isSymbol();
    virtual Function * isFunction();
    virtual ListNode * isList();
};

void Thunk::print()
{
    if (evaluated)
        value()->print();
    else
        printf("...");
}

Expression * Thunk::touch()
{
    // if we haven't already evaluated, do it now
    if (! evaluated) {
        evaluated = 1;
        Expression * start = value();
        if (start)
            start->eval(value, valueOps, context);
        }
    Expression * val = value();
    if (val)
        return val->touch();
    return val;
}
```

Figure 5.2: Definition of Thunks

```
void Thunk::eval(Expr & target, Environment * valusops, Environment * rho)
{
    touch();
    value()→eval(target, valusops, rho);
}

ListNode * Thunk::isList()
{
    // if its evaluated try it out
    if (evaluated) return value()→isList();

    // else it's not
    return 0;
}

Symbol * Thunk::isSymbol()
{
    if (evaluated) return value()→isSymbol();
    return 0;
}

Function * Thunk::isFunction()
{
    if (evaluated) return value()→isFunction();
    return 0;
}

IntegerExpression * Thunk::isInteger()
{
    if (evaluated) return value()→isInteger();
    return 0;
}
```

Figure 5.3: Thunk predicates

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

void SaslConsFunction::apply(Expr & target, ListNode * args, Environment * rho)
{
   // check length
   if (args→length() != 2) {
      target = error("cons requires two arguments");
      return;
      }

   // make thunks for car and cdr
   target = new ListNode(new Thunk(args→at(0), rho),
      new Thunk(args→at(1), rho));
}
```

Figure 5.4: The Sasl Lazy Cons function

```
class LazyFunction : public UserFunction {
public:
    LazyFunction(ListNode * n, Expression * b, Environment * c)
        : UserFunction(n, b, c) {}
    virtual void apply(Expr &, ListNode *, Environment *);
};

//    convert arguments into thunks
static ListNode * makeThunks(ListNode * args, Environment * rho)
{
    if ((! args) || (args→isNil()))
        return emptyList;
    Expression * newcar = new Thunk(args→head(), rho);
    return new ListNode(newcar, makeThunks(args→tail(), rho));
}

void LazyFunction::apply(Expr & target, ListNode * args, Environment * rho)
{
    // number of args should match definition
    ListNode * anames = argNames;
    if (anames→length() != args→length()) {
        error("argument length mismatch");
        return;
        }

    // convert arguments into thunks
    ListNode * newargs = makeThunks(args, rho);

    // make new environment
    Env newrho = new Environment(anames, newargs, context);

    // evaluate body in new environment
    if (body())
        body()→eval(target, valueOps, newrho);
    else
        target = 0;

    newrho = 0;
}
```

Figure 5.5: The implementation of lazy functions

Chapter 6

The CLU interpreter

The CLU interpreter is created by introducing a new datatype, the cluster, and three new types of functions. Constructors create new instances of a cluster, selectors access a portion of a cluster state, and modifiers change a portion of a cluster state. Figure 6.1 shows the class hierarchy for the classes added in this chapter.

6.1 Clusters

A cluster simply encapsulates a series of names and values, hiding them from normal examination. The most natural way to do this is for a cluster to maintain an environment (Figure 6.2). The predicate isCluster returns this environment value.

To create a cluster requires a constructor function. The constructor is provided with a list of names of the elements in the internal representation of the cluster, and simply insures that the arguments it is provided with match in number of the names it maintains.

6.2 Selectors and Modifiers

To access or modify the elements of a constructor requires functions called selectors or modifiers. Each of these maintain as their state the name of the field they are responsible for. When invoked with a constructor, the access or change their given field.

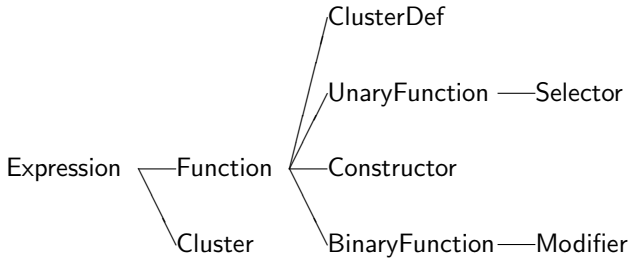

Figure 6.1: Class Hierarchy for the CLU interpreter

6.3 Defining clusters

It thus remains only to give the (rather lengthy) definition of the function that generates constructor information from the textual description. (We do not say generates clusters themselves, for that is the responsibility of the constructor functions). This function is shown in Figure 6.4. It rips apart a cluster definition and does the right things (need a better description here, but I don't have time to write it now). (Need to point out that cluster functions have an internal and an external name, and these are put of different environments). (I suppose an alternative would have been to introduce a new datatype for two part names, which when evaluated would look up their second part in the cluster provided by their first part).

```
class Cluster : public Expression {
private:
    Env data;
public:
    Cluster(ListNode * names, ListNode * values)
        { data = new Environment(names, values, 0); }
    virtual void free()
        { data = 0; }
    virtual void print()
        { printf("<userval>"); }
    virtual Environment * isCluster()
        { return data; }
};

class Constructor : public Function {
private:
    List names;
public:
    Constructor(ListNode * n);
    virtual void free();
    virtual void applyWithArgs(Expr &, ListNode *, Environment *);
};

void Constructor::applyWithArgs(Expr &target, ListNode *args, Environment
*rho)
{
    ListNode * nmes = names;
    if (args→length() != nmes→length()) {
        target = error("wrong number of args passed to constructor");
        return;
        }
    target = new Cluster(nmes, args);
}
```

Figure 6.2: The definition of a cluster value

```
class Selector : public UnaryFunction {
private:
    Expr fieldName;
public:
    Selector(Symbol * name);
    virtual void free();
    virtual void applyWithArgs(Expr &, ListNode *, Environment *);
};

void Selector::applyWithArgs(Expr & target, ListNode * args, Environment * rho)
{
    Environment * x = args→head()→isCluster();
    if (! x) {
        target = error("selector given non-cluster");
        return;
        }
    Symbol *s = fieldName()→isSymbol();
    if (!s)
        error("impossible case in selector, no symbol");
    target = x→lookup(s);
    if (! target())
        error("selector cannot find symbol:", s→chars());
}

class Modifier : public BinaryFunction {
private:
    Expr fieldName;
public:
    Modifier(Symbol * name);
    virtual void free();
    virtual void applyWithArgs(Expr &, ListNode *, Environment *);
};

void Modifier::applyWithArgs(Expr & target, ListNode * args, Environment * rho)
{
    Environment * x = args→head()→isCluster();
    if (! x) {
        target = error("selector given non-cluster");
        return;
        }

    // set the result to the value
    target = args→at(1);
    x→set(fieldName()→isSymbol(), target());
}
```

Figure 6.3: Selectors and Modifiers for clusters

```
void ClusterDef::apply(Expr & target, ListNode * args, Environment * rho)
{
    Expr setprefix = new Symbol("set-");

    // must have at least name, rep and one def
    if (args→length() < 3) {
        target = error("cluster ill formed");
        return;
        }

    // get name
    Symbol * name = args→head()→isSymbol();
    args = args→tail();
    if (! name) {
        target = error("cluster missing name");
        return;
        }

    // now make the environment in which cluster will execute
    Environment * inEnv = new Environment(emptyList, emptyList, rho);

    // next part should be representation
    ListNode * rep = args→head()→isList();
    args = args→tail();
    if (! rep) {
        target = error("cluster missing rep");
        return;
        }
    Symbol *s = rep→at(0)→isSymbol();
    if (! (s && (*s == "rep"))) {
        target = error("cluster missing rep");
        return;
        }
    rep = rep→tail();

    // make the name into a constructor with the representation
    inEnv→add(name, new Constructor(rep));
```

Figure 6.4: The cluster recognition function

```
    // now run dow the rep list, making accessor functions
    while (! rep→isNil()) {
        s = rep→head()→isSymbol();
        if (! s) {
            target = error("ill formed rep in cluster");
            return;
            }
        inEnv→add(s, new Selector(s));
        catset(inEnv, setprefix()→isSymbol(), "",
            s, new Modifier(s));
        rep = rep→tail();
        }

    // remainder should be define commands
    while (! args→isNil()) {
        ListNode * body = args→head()→isList();
        if (! body) {
            target = error("ill formed cluster");
            return;
            }
        s = body→at(0)→isSymbol();
        if (! (s && (*s == "define"))) {
            target = error("missing define in cluster");
            return;
            }
        s = body→at(1)→isSymbol();
        if (! s) {
            target = error("missing name in define");
            return;
            }

        // evaluate body to define new function
        Expr temp;
        body→eval(temp, commands, inEnv);
        // make outside form
        catset(rho, name, "$", s, inEnv→lookup(s));
        temp = 0;

        // get next function
        args = args→tail();
        }

    // what do we return?
    target = 0;
    setprefix = 0;
}
```



```
static void catset(Environment * rho, Symbol * left, char * mid,
        Symbol * right, Expression * val)
{    char buffer[120];

    // catenate the two symbols
    strcpy(buffer, left→chars());
    strcat(buffer, mid);
    strcat(buffer, right→chars());

    // now put the new value into rho
    rho→add(new Symbol(buffer), val);
}
```

Figure 6.5: The cluster recognition function (continued)

Chapter 7

The Smalltalk interpreter

As with chapter 3, with the Smalltalk interpreter I have also made a number of changes. These include the following:

- I have changed the syntax for message passing. The first argument in a message passing expression is an object, which is defined (for implementation purposes) as a type of function. The second argument must be the message selector, a symbol. This change is not only produces a syntax that is slightly more Smalltalk-like, but it more closely reinforces the critical object-oriented idea that the interpretation of a message depends upon the receiver for that message.
- Integers are objects, and respond to messages. The most obvious effect of this is to restore infix syntax for arithmetic operations, since (3 + 4) is interpreted (Smalltalk-like) as the message "+" being passed to the object 3 with argument 4.
- The initial environment is very spare. There are only the two classes Object and Integer, which respond to the messages subclass, method and new, and integer instances that respond to arithmetic messages.
- The if command is a message sent to integers (0 for false and non-zero for true). This is also more Smalltalk-like. The following expression sets z to the minimum of x and y.

  ((x < y) if (set z x) (set z y))

- The only non-message statements are the assignment statement set and the begin statement. (Note - there is no loop. I couldn't think of a good way to do this within the syntax given using message passing (no blocks!) but I don't think this will be too great a problem; recursion can be used in most cases where looping is used currently).

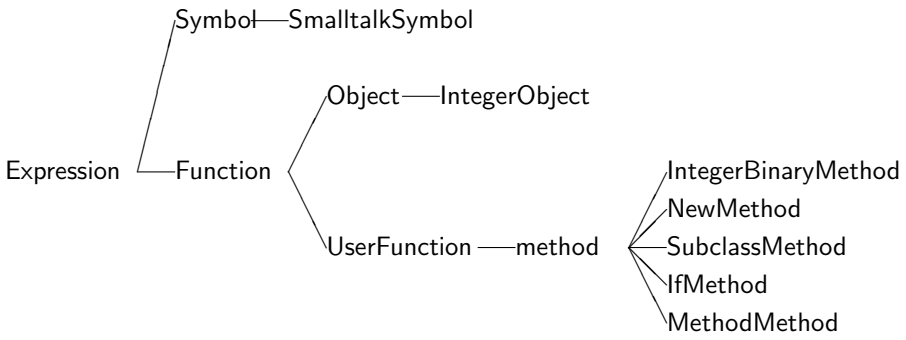

Figure 7.1: Expression class hierarchy for Smalltalk interpreter

A class hierarchy for the classes added in this chapter is shown in Figure 7.1.

7.1 Objects and Methods

An object is an encapsulation of behavior and state. That is, an object maintains, like a cluster, certain state information accessible only within the object. Similarly objects maintain a collection of functions, called *methods*, that can be invoked only via message passing. Internally, both these are represented by environments (Figure 7.2). The methods environment contains a collection of functions, and the data environment contains a collection of internal variables. Objects are declared as a subclass of function so that normal function syntax can be used for message passing. That is, a message is written as

(object message arguments)

Methods are similar to conventional functions (and are thus subclasses of UserFunction) in that they have an argument list and body. Unlike conventional functions they have a receiver (which must always be an object) and the environment in which the method was created, as well as the environment in which the method is invoked. Thus methods define a new message doMethod that takes these additional arguments.

A subtle point to note is that the creation environment in normal functions is captured when the function is defined. For objects this environment cannot be defined when the methods are created, but must wait until a new instance is created. Our implementation waits even longer, and passes it as part of the message passing protocol.

The mechanism of message passing is defined by the function apply in class Object (Figure 7.3). Messages require a symbol for the first argument, which

```
class Object : public Function {
private:
   Env methods;
   Env data;
   friend class SubclassMethod;
public:
   Object(Environment * m, Environment * d);

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

   // methods used by classes to create new instances
   // note these are invoked only on classes, not simple instances
   ListNode * getNames();
   Environment * getMethods();
};

class Method : public UserFunction {
public:
   Method(ListNode *anames, Expression * bod) ;

   virtual void doMethod(Expr&, Object*, ListNode*,
      Environment*, Environment*);

   virtual Method * isMethod();
};
```

Figure 7.2: Classes for Object and Method

must match a method for the object. This method is then invoked. Similarly Figure 7.3 shows the execution of normal methods (that is, those methods other than the ones provided by the system). The execution context is set for the method, and the receiver is added as an implicit first argument, called self in every method. The method is then invoked as if it were a conventional function.

7.2 Classes

Classes are simply objects. As such, they respond to certain messages. In our Smalltalk interpreter there are initially two classes, Object and Integer. The class Object is a superclass of Integer, and is typically the superclass of most user defined classes as well. There are initially three messages that classes respond to:

- subclass. This message is used to create new classes, as subclasses of existing classes. Any arguments provided are treated as the names of instance variables (local state) to be generated when instances of the new classes are created. The new class is returned as an object, and is usually immediately assigned to a global variable. The syntax for new classes is thus similar to the following:

  ```
  (set Foo (Object subclass x y z))
  ```

  which creates a new class with three instance variables, and assigns this class to the variable Foo. Subclasses can also access instance variables defined in classes.

  It is legal to subclass from class Integer, although the results are not useful for any purpose.

- new. This message, which takes no arguments, is used to create a new instance of the receiver class. The new instance is returned as the result of the method, as in the following:

  ```
  (set newfoo (Foo new))
  ```

  Although the class Integer responds to the message new, no useful value is returned. (Real Smalltalk has something called *metaclasses* that can be used to prevent certain classes from responding to all messages. Our Smalltalk doesn't).

- method. This message is used to define a new method for a class. Following the keyword method the syantx is the same as a normal function definition. Within a method the pseudo-variable self can be used to represent the receiver for the method.

```
void Object::apply(Expr & target, ListNode * args, Environment * rho)
{
    // need at least a message
    if (args→length() < 1) {
        target = error("ill formed message expression");
        return;
        }

    Symbol * message = args→head()→isSymbol();
    if (! message) {
        target = error("object needs message");
        return;
        }

    // now see if message is a method
    Environment * meths = methods;
    Expression * methexpr = meths→lookup(message);
    Method * meth = 0;
    if (methexpr) meth = methexpr→isMethod();
    if (! meth) {
        target = error("unrecognized method name: ", message→chars());
        return;
        }

    // now just execute the method (take off message from arg list)
    meth→doMethod(target, this, args→tail(), data, rho);
}

void Method::doMethod(Expr& target, Object* self, ListNode* args,
    Environment *ctx, Environment *rho)
{
    // change the execution context
    context = ctx;

    // put self on the front of the argument list
    List newargs = new ListNode(self, args);

    // and execute the function
    apply(target, newargs, rho);

    // clean up arg list
    newargs = 0;
}
```

Figure 7.3: Implementation of Message Passing

```
(Integer method square () (self * self))
```

Classes are represented in the same format as other objects. They act as if they held two instance variables; names, which contains a list of instance variable names for the class, and methods, which contains the table of method definitions for the class. Note that these are held in the data area for the class. (A picture might help here...).

The implementation of the method subclass is shown in Figure 7.4. The instance variables for the parent class is obtained, and the new instance variables for the class added to them. Inheritance is implemented by creating a new empty method table, but having it point to the method table for the parent class. Thus a search of the method table for the newly created class will automatically search the parent class if no overriding method is found. These two values are inserted as data values in the new class object. The methods a class responds to will be exactly the same as those of the parent class (thus all classes respond to the same messages).

The implementation of the method new, shown in Figure 7.5, gets the list of instance variables associated with the class. A new environment is then created that assigns an empty value to each variable. Using the method table stored in the data area for the class object a new object is then created.

The method used to respond to the method command is shown in Figure 7.6. This is very similar to the function used to break apart the define command in Chapter 1. The only significant difference includes the addition of the receiver self as an implicit first parameter in the argument list, and the fact that the function is placed in a method table, rather than in the global environment.

7.3 Symbols and Integers

Symbols in Smalltalk have no property other than they evaluate to themselves, and are guaranteed unique. They are easily implemented by subclassing the existing class Symbol (Figure 7.7), and modifying the reader/parser to recognize the tokens. (Unlike symbols in real Smalltalk, our symbols are not objects and will not respond to any messages).

Integers are also redefined as objects, and a built-in method IntegerBinaryMethod, similarly to IntegerBinaryFunction, is created to simplify the arithmetic methods.

Control flow is implemented as a message to integers. (In real Smalltalk control flow is implemented as messages, but to different objects). If the receiver is zero the first argument to the if method is returned, otherwise the second argument is returned.

```
void SubclassMethod::doMethod(Expr& target, Object* self, ListNode *args,
    Environment *ctx, Environment *rho)
{

    // the argument list is added to the list of variables
    ListNode * vars = self→getNames();
    while (! args→isNil()) {
        vars = new ListNode(args→head(), vars);
        args = args→tail();
        }

    // the method table is empty, but points to inherited method table
    Environment * newmeth = new Environment(emptyList, emptyList,
            self→getMethods());

    // make the new data area
    Environment * newEnv = new Environment(emptyList, emptyList, rho);
    newEnv→add(new Symbol("names"), vars);
    newEnv→add(new Symbol("methods"), newmeth);

    // now make the new object
    Environment * meths = self→methods;
    target = new Object(meths, newEnv);
}

ListNode * Object::getNames()
{
    Environment * datavals = data;
    Expression * x = datavals→lookup(new Symbol("names"));
    if ((! x) || (! x→isList())) {
        error("impossible case in Object::getNames");
        return 0;
        }
    return x→isList();
}
```

Figure 7.4: Implementation of the subclass method

```
void NewMethod::doMethod(Expr& target, Object* self, ListNode *args,
    Environment *ctx, Environment *rho)
{
    // get the list of instance names
    ListNode * names = self→getNames();

    // cdr down the list, making a list of values (initially zero)
    ListNode * values = emptyList;
    for (ListNode *p = names; ! p→isNil(); p = p→tail())
        values = new ListNode(new IntegerExpression(0), values);

    // make the new environment for the names
    Environment * newenv = new Environment(names, values, rho);

    // make the new object
    target = new Object(self→getMethods(), newenv);
}
```

Figure 7.5: The method new

7.4 Smalltalk reader

The Smalltalk reader subclasses the reader class so as to recognize integers and symbols (Figure 7.9).

7.5 The big bang

To initialize the interpreter we must create the objects Object and Integer. (Need more explanation here, but I'll just give the code for now).

```
void MethodMethod::doMethod(Expr& target, Object* self, ListNode *args,
    Environment *ctx, Environment *rho)
{
    if (args→length() != 3) {
        target = error("method definition requires three arguments");
        return;
        }
    Symbol * name = args→at(0)→isSymbol();
    if (! name) {
        target = error("method definition missing name");
        return;
        }

    ListNode * argNames = args→at(1)→isList();
    if (! argNames) {
        target = error("method definition missing arg names");
        return;
        }
    // put self on front of arg names
    argNames = new ListNode(new Symbol("self"), argNames);

    // get the method table for the given class
    Environment * methTable = self→getMethods();

    // put method in place
    methTable→add(name, new Method(argNames, args→at(2)));

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

Figure 7.6: The method method

```
class SmalltalkSymbol : public Symbol {
public:
    virtual void eval(Expr & target, Environment *, Environment *)
        { target = this; }
};

static Env IntegerMethods;

class IntegerObject : public Object {
private:
    Expr value;
public:
    IntegerObject(int v) : Object(IntegerMethods, 0)
        { value = new IntegerExpression(v); }

    virtual void print()
        { if (value()) value()->print(); }

    virtual void free()
        { value = 0; }

    virtual IntegerExpression * isInteger()
        { if (value()) return value()->isInteger(); return 0; }
};
```

Figure 7.7: Symbols and Integers in Smalltalk

```
void IfMethod::doMethod(Expr & target, Object * self,
    ListNode * args, Environment * ctx, Environment * rho)
{
    if (args→length() != 2) {
        target = error("wrong number of args for if");
        return;
        }
    IntegerExpression * cond = self→isInteger();
    if (! cond) {
        target = error("impossible!", "no cond in if");
        return;
        }
    if (cond→val())
        args→at(0)→eval(target, valueOps, rho);
    else
        args→at(1)→eval(target, valueOps, rho);
}
```

Figure 7.8: Implementation of the if method

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

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

    // or it might be a symbol
    if (*p == '#') {
        char token[80], *q;

        for (q = token; ! isSeparator(*p); )
            *q++ = *p++;
        *q = '\0';
        return new SmalltalkSymbol(token);
        }

    // anything else, do as before
    return Reader::readExpression();
}
```

Figure 7.9: The Smalltalk reader

```
initialize()
{
    // initialize global variables
    reader = new SmalltalkReader;

    // the only commands are the assignment command  and begin
    Environment * vo = valueOps;
    vo→add(new Symbol("set"), new SetStatement);
    vo→add(new Symbol("begin"), new BeginStatement);

    // initialize the global environment
    Environment * ge = globalEnvironment;

    // first create the object "Object"
    Environment* objMethods = new Environment(emptyList, emptyList, 0);
    Environment* objClassMethods = new Environment(emptyList, emptyList,
            objMethods);
    objClassMethods→add(new Symbol("new"), new NewMethod);
    objClassMethods→add(new Symbol("subclass"), new SubclassMethod);
    objClassMethods→add(new Symbol("method"), new MethodMethod);
    Environment * objData = new Environment(emptyList, emptyList, 0);
    objData→add(new Symbol("names"), emptyList());
    objData→add(new Symbol("methods"), objMethods);
    ge→add(new Symbol("Object"),
          new Object(objClassMethods, objData));

    // now make the integer methods
    IntegerMethods = new Environment(emptyList, emptyList, objMethods);
    Environment * im = IntegerMethods;
    // the integer methods are just as before
    im→add(new Symbol("+"), new IntegerBinaryMethod(PlusFunction));
    im→add(new Symbol("-"), new IntegerBinaryMethod(MinusFunction));
    im→add(new Symbol("*"), new IntegerBinaryMethod(TimesFunction));
    im→add(new Symbol("/"), new IntegerBinaryMethod(DivideFunction));
    im→add(new Symbol("="), new IntegerBinaryMethod(IntEqualFunction));
    im→add(new Symbol("<"), new IntegerBinaryMethod(LessThanFunction));
    im→add(new Symbol(">"), new IntegerBinaryMethod(GreaterThanFunction));
    im→add(new Symbol("if"), new IfMethod);
    ge→add(new Symbol("Integer"),
          new Object(objClassMethods, objData));
}
```

Figure 7.10: Initializing the Smalltalk interpreter

Chapter 8

The Prolog interpreter

As with chapters 3 and 7, I have in this chapter taken great liberties with the syntax used by Kamin in his interpreter. However, unlike chapters 3 and 7, where my intent was to make the interpreters closer in spirit to the original language, my intent here is to simplify the interpreter. Specifically, I wanted to build on the base interpreter, just as we have done for all other languages. I am able to do this by adopting *continuations* as the fundamental basis for my implementation, and by basing the code on slightly different primitives.

The language used by this interpreter has the following characteristics:

- As in real prolog, the only basic objects are symbols. I've even tossed out integers, just to simplify things. Symbols have no meaning other than their uniqueness. Those symbols beginning with lower case letters are atomic, while those beginning with upper case letters are variables.
- There are two basic statement types, the define statement we have seen all through the interpreters, and a new statement called query. The later is used to form questions.
- The bodies of functions or queries can be composed of four types of relations:
  - (print x) which if x is defined prints the value of x and is successful, and if x is not defined is not successful.
  - (:=: x y) which attempts to unify x and y, which can be either variables or symbols. The order of arguments is unimportant.
  - (and rel_1 rel_2 ...) which can take any number of relational arguments and is successful if all the relations are successful. Relations are tried in order.

– (or rel_1 rel_2 ...) which can take any number of relational arguments and is successful if one one of the relations is successful. They are tried in order.

For example, suppose sam is the father of alice, and alice is the mother of sally. We might encode this in a parent database as follows:

```
→ (define parent (X Y)
   (or
      (and (:=: X alice) (:=: Y sally))
      (and (:=: X sam) (:=: Y alice))
   ))
```

The query statement can then be used to ask queries of the database. For example, we can find out who is the parent of alice as follows:

```
→ (query (and (parent X alice) (print X)))
sam
ok
```

Or we can find the child of alice with the following:

```
→ (query (and (parent alice X) (print X)))
sally
ok
```

If we ask a question that does not have an answer, the response not-ok is printed.

```
→ (query (and (parent fred X) (print X)))
not ok
```

Prolog style rules can be introduced using the same form we have been using for functions.

```
→ (define grandparent (X Y)
   (and (parent X Z) (parent Z Y)))
→ (query (and (grandparent A B) (print A) (print B)))
sam
sally
ok
```

There is no built-in way to force a relation to cycle through all alternatives. However, this is easily accomplished by making a relation that will always fail, for example trying to unify apples with oranges:

```
→ (define fail () (:=: apples oranges))
```

We can then use this to print out all the parents in our database. Notice that not-ok is printed, since we eventually fail.

```
→ (query (and (parent X Y) (print X) (fail)))
sam
alice
not ok
```

Note - although it might appear the use of and's and or's is more powerful than writing rules in horn clauses, in fact they are identical; although horn

clauses will often require the introduction of unnecessary names. I myself find this formulation more natural, although I'm not exactly unbiased.

Unification of two unknown symbols works as expected. If any symbol subsequently becomes defined, the other is defined as well.

```
→ (define same (X Y) (:=: X Y))
→ (query (and (same A B) (:=: A sally) (print B)))
sally
ok
```

I will divide the discussion of the implementation into three parts. These are unification, symbol management, and backtracking.

8.1 Unification

Unification is the basis for logic programming. Using unification, unbound variables can be bound together. As we saw in the last example, this is more than simple assignment. If two unknown variables are unified together and subsequently one is bound, the other should be bound also. Unification also differs from assignment in that it can be "undone" during the process of backtracking.

Unification is most easily implemented by introducing a level of indirection. Prolog values will be represented by a new type of expression, called PrologValue (Figure 8.1). Instances of this class maintain a data value, which is either undefined (that is, null), a symbol, or another prolog value. The prolog reader is modified so as to return a prolog value were formerly a symbol was returned. (Also the reader will no longer recognize integers, which are not used in our simplified interpreter).

A prolog value that contains a symbol is used to represent the prolog symbol of the same name. A prolog value that contains an empty data value represents a currently unbound value. Finally a prolog value that points to another prolog value represents the unification of the first value with the second. Whenever we need the value of a prolog symbol, we first run down the chain of indirections to get to the bottom of the sequence. (This is done automatically by the overridden method isSymbol, which will yield the symbol value behind arbitrary levels of indirection if a prolog value represents a symbol.)

The unification algorithm is shown in Figure 8.2. For reasons we will return to when we discuss backtracking, the algorithm takes three arguments. The first is a reference to a pointer to a prolog value. If the unification process changes the value of either the the two other arguments, the pointer in the first argument is set to the altered value.

The unification process divides naturally into three parts. If either argument is undefined, it is changed so as to point to the other arguments. This is true regardless of the state of the other argument. This is how two undefined variables can be unified - the first is set to point to the second. If the second is subsequently changed, the first will still indirectly point to the new value.

```
class PrologValue : public Expression {
private:
   Expr data;

public:
   PrologValue(Expression * d) { data = d; }

   virtual void free() { data = 0; }
   virtual void print();
   virtual void eval(Expr &, Environment *, Environment *);
   virtual Symbol * isSymbol();
   virtual PrologValue * isPrologValue() { return this; }

   int isUndefined()          { return data() == 0; }
   void setUndefined()          { data = 0; }
   PrologValue * indirectPtr();
   void setIndirect(PrologValue *v)    { data = v; }
};
```

Figure 8.1: The class declaration for prolog values

Suppose it is, however, the first that is subsequently changed? In that case the next portion of the unification algorithm is entered. If both arguments are defined and either one is an indirection, then we simply try to unify the next level down in the pointer chains. (Note: Lots of pictures would make this clearer, but I don't have time right now..). If neither argument is undefined nor an indirection, they both must be symbols. In that case, unification is successful if and only if they have the same textual representation.

8.2 Symbol Management

The only significant problem here is that symbolic constants must evaluate to themselves and that symbolic variables can be introduced without declaration. We see the latter in sequences such as:

```
→ (define grandparent (X Y)
   (and (parent X Z) (parent Z Y)))
→ (query (and (grandparent A B) (print A) (print B)))
sam
sally
ok
```

Here the variable Z suddenly appears without prior use. The solution to both of these problems is found in the code used to respond to the eval request for a Prolog value. This code is shown in Figure 8.3. The virtual method isSymbol

```
static int unify(PrologValue *& c, PrologValue * a, PrologValue * b)
{

    // if either one is undefined, set it to the other
    if (a→isUndefined()) {
        c = a;
        a→setIndirect(b);
        return 1;
        }
    if (b→isUndefined()) {
        c = b;
        b→setIndirect(a);
        return 1;
        }

    // if either one are indirect, run down chain
    PrologValue * indirval;
    indirval = a→indirectPtr();
    if (indirval)
        return unify(c, indirval, b);
    indirval = b→indirectPtr();
    if (indirval)
        return unify(c, a, indirval);

    // both must now be symbolic, work if the same
    c = 0;
    Symbol * as = a→isSymbol();
    Symbol * bs = b→isSymbol();
    if ((! as) || (! bs))
        error("impossible", "unification of non-symbols");
    else if (strcmp(as→chars(), bs→chars()) == 0)
        return 1;
    return 0;
}
```

Figure 8.2: The Unification process

runs down any indirection links, returning the symbol data value if the last value in a chain of indirections represents a symbolic constant. If a symbol is found, we first look to see if the symbol is bound in the current environment. If so we simply return its binding[1]. If not, if the symbol begins with a lower case letter it evaluates to itself, and so we simply return it. If it is not a symbolic constant, than it is a new symbolic variable, and we add a binding to the current environment to indicate that the value is so-far undefined. Thus new symbols are added to the current environment as they are encountered, instead of generating error messages as they did in previous interpreters.

8.3 Backtracking

The seem to be two general approaches to implementing logic programming languages. The technique used by most modern prolog systems is called the WAM, or Warren Abstract Machine. The WAM performs backtracking by not popping the activation frame stack when a procedure is terminated, and saving enough information to restart the procedure in a record called the "choice point". Since in our interpreters calling a function is performed by recursively calling evaluation routines inside the interpreter, the activation stack for the users program is held in part in the activation stack for the interpreter itself. Thus it is difficult for us to manipulate the activation record stack directly. The alternative technique, which is actually historically older, is to build up an unevaluated expression that represents what it is you want to do next before you ever start execution. This is called a continuation, and we were introduced to this idea in the chapter on Scheme. When we are faced with a choice, we can then try one alternative and the continuation, and if that doesn't work try the next.

In general continuations are simply arbitrary expressions representing "what to do next". In our case they will always return a boolean value, indicating whether they are to be considered successful or not. We will sometimes refer to the continuation as the "future", since it represents the calculation we want to perform in the future.

In order to illustrate how backtracking can be implemented using continuation, let us consider the following invocation of our family database:

```
(query (and (grandparent sam A) (print A)))
```

There are two important points to note. The first is that the general approach will be a two step process, construct the future that represents the calculation we want to do, then do it. The second point is that the details are exceedingly messy; you should be eternally grateful that it is the computer that is performing this task, and not you.

[1] Checking for bindings before checking the first letter allows rules to begin with lower case letters, which seems to be more natural to most programmers.

```
Symbol * PrologValue::isSymbol()
{
    PrologValue * iptr = indirectPtr();
    if (iptr)
        return iptr→isSymbol();
    if (! isUndefined())
        return data()→isSymbol();
    return 0;
}

void PrologValue::eval(Expr&target, Environment*valueOps, Environment*rho)
{
    Symbol * s = isSymbol();
    if (s) {
        char * p = s→chars();
        Expression * r = rho→lookup(s);
        if (r) {
            target = r;
            return;
            }
        // symbol is not known
        // if lower case, eval to itself
        if ((*p >= 'a') && (*p <= 'z')) {
            // symbols eval to themselves
            target = this;
            return;
            }
        // else make a new symbol
        target = new PrologValue(0);
        rho→add(s, target());
        return;
        }
    target = this;
    return;
}
```

Figure 8.3: Evaluation of a prolog symbol

To begin, the continuation that represents what it is we want to do after evaluating the query is the null continuation, an expression that merely returns true. In order to try to keep track of the multiple levels of evaluation, let us write this as follows:

```
(and (grandparent sam A) (print A)) [ true ]
```

This says that we want to evaluate the and relation, and then do the calculation given by the bracketed expression.

Consider now the meaning of and. The and expression should evaluate the first relation, and if successful evaluate the second, and finally if that is successful evaluate the future given to the original expression. What then is the "future" of the first relation? It is simply the second relation and the original future. That is, the calculation we want to perform if the first relation is successful is simply the following:

```
(print A) [ true ]
```

We can wrap this in a bracket in order to make a continuation in our form out of it. Using *this* as the future for the first relation gives us the following:

```
(grandparent sam A) [ (print A) [ true ] ]
```

We are in effect turning the calculation inside out. We have replaced the and conjunction with a list of expressions to evaluate in the future.

The invocation of the grandparent relation causes the expression to be replaced by the function definition, with the arguments suitably bound to the parameters. That is, the effect is the same as:[2]

```
(and (parent sam Z) (parent Z A)) [ (print A) [ true ] ]
```

We have already analyzed the meaning of the and relation. The future we want to provide for the first relation is the expression yielded by:

```
(parent Z A) [ (print A) [ true ] ]
```

As before, we can expand the invocation of the parent relation by replacing it by its definition, making suitable transformations of the argument values.

```
(or
   (and (:=: Z alice) (:=: A sally))
   (and (:=: Z sam) (:=: A alice)) )
      [ (print A) [ true ] ]
```

The or relation should try each alternative in turn, passing it as the future the continuation passed to the or. If any is successful we should return success, otherwise the or should fail. Thus we can distribute the future to each clause of the or, and rewrite it as follows:[3]

```
if (and (:=: Z alice) (:=: A sally))
      [ (print A) [ true ] ]
```

[2]We will use textual replacement of the parameters by the arguments in our example, although in practice the effect is achieved via a level of indirection provided by environments, as in all the interpreters we have studied.

[3]The fact that we are replacing or by a conditional may seem odd, but the more important point is that we have moved the evaluation of the future down to each of the arguments to the or expression.

```
then return true
else if (and (:=: Z sam) (:=: A alice)) )
       [ (print A) [ true ] ]
then return true
else return false
```

If we perform the already-defined transformations on the and relations we obtain the following:

```
if (:=: Z alice) [ (:=: A sally) [ (print A) [ true ] ] ]
then return true
else if (:=: Z sam) [ (:=: A alice) [ (print A) [ true ] ] ]
then return true
else return false
```

Recall that this was all performed just to construct the continuation for the first clause in an earlier expression. Thus the expression we are now working on is as follows:

```
(parent sam Z) [
   if (:=: Z alice) [ (:=: A sally) [ (print A) [ true ] ] ]
   then return true
   else if (:=: Z sam) [ (:=: A alice) [ (print A) [ true ] ] ]
   then return true
   else return false ]
```

We before, we can expand the call on parent by its definition:[4]

```
(or
   (and (:=: sam alice) (:=: Z sally))
   (and (:=: sam sam) (:=: Z alice)))
[ if (:=: Z alice) [ (:=: A sally) [ (print A) [ true ] ] ]
   then return true
   else if (:=: Z sam) [ (:=: A alice) [ (print A) [ true ] ] ]
   then return true
   else return false ]
```

Once again distributing the future along each argument of the or expression yields:

```
if (and (:=: sam alice) (:=: Z sally))
   [ if (:=: Z alice) [ (:=: A sally) [ (print A) [ true ] ] ]
   then return true
   else if (:=: Z sam) [ (:=: A alice) [ (print A) [ true ] ] ]
   then return true
   else return false ]
then return true
else if (and (:=: sam sam) (:=: Z alice))
   [ if (:=: Z alice) [ (:=: A sally) [ (print A) [ true ] ] ]
   then return true
```

[4] I warned you about the messy details!

```
    else if (:=: Z sam) [ (:=: A alice) [ (print A) [ true ] ] ]
    then return true
    else return false ]
else return false
```

Performing yet one more time the transformations on the and relations yields:

```
if (:=: sam alice) [ (:=: Z sally)
    [ if (:=: Z alice) [ (:=: A sally) [ (print A) [ true ] ] ]
    then return true
    else if (:=: Z sam) [ (:=: A alice) [ (print A) [ true ] ] ]
    then return true
    else return false ] ]
then return true
else if (:=: sam sam) [ (:=: Z alice)
    [ if (:=: Z alice) [ (:=: A sally) [ (print A) [ true ] ] ]
    then return true
    else if (:=: Z sam) [ (:=: A alice) [ (print A) [ true ] ] ]
    then return true
    else return false ]
then return true
else return false
```

This is the final continuation that is constructed by the query expression. The most important feature of this expression is that it can be evaluated in a forward fashion, without backtracking. Having generated it, the next step is execution. Contrast this with the description we provided earlier. First an attempt is made to unify the symbols sam and alice. This fails, and thus the continuation for the first conditional is ignored. Next an attempt is made to unify the symbols sam and sam. This is successful, and thus we evaluate the continuation to the next expression. The continuation unifies Z and alice, binding the left-hand variable to the right-hand symbol. The continuation for that expression then trys to unify Z and alice, which is successful. Thus variable A is bound to sally, and is printed.[5]

Having described the general approach our interpreter will follow, we will now go on to provide the specific details.

Our continuations are built around a new datatype, which we will call the Continuation. A continuation should be thought of as an unevaluated boolean expression. The continuation performs some action, which may or may not succeed. The success of the action is indicated by the boolean value returned. The class Continuation is shown in Figure 8.4. The routine used to invoke a relation is the virtual method withContinuation, which takes as argument the future for the continuation.

[5] Observant readers will have noted that some of the conditionals could have been evaluated during the construction of the continuation. This is true, and is an important optimization in real systems.

```
class Continuation : public Expression {
public:
    virtual int withContinuation(Continuation *);
    virtual void print() { printf("<future>"); }
    virtual Continuation * isContinuation() { return this; }
};

static Continuation * nothing;    // the null continuation

int Continuation::withContinuation(Continuation * future)
{
    // default is to always work
    return 1;
}
```

Figure 8.4: The class Continuation

Initially there is nothing we want to do in the future. So the initial relation simply ignores its future, does nothing and always succeeds. In fact, in our implementation we maintain a global variable called nothing to hold this relation. You can think of this variable as maintaining the relation [true].

The simplest relation is the one correspond to the command to print. When a print relation is created, the value it will eventually print is saved as part of the relation. If the argument passed to the print relation is, following any indirection, a symbolic value than it is printed out, and the future passed to the relation is invoked. If the argument was not a symbol, or if the future calculation was unsuccessful, then the relation indicates its failure by returning a zero value. The code to accomplish this is shown in Figure 8.5.

Next let us consider the unification relation. As with printing, the two expressions representing the elements to be unified are saved when the unification operator is encountered during the construction of the future. When we invoke this relation the two arguments are unified, using the algorithm we have previously described. If this unification is successful the relation attempts to evaluate the future continuation. Only if both of these are successful does the relation return one. If either the unification fails or the future fails then the binding created by the unify procedure is undone and failure is reported. (Figure 8.6).

Next consider the or relation (Figure 8.7). This relation takes some number of argument relations. It tries each in turn, followed by the future it has been provided with. If any succeeds then it returns a true value, otherwise if all fail it returns a failure indication.

It is in the or relation that backtracking occurs, although it is difficult to tell from the code shown here. Recall that the unification algorithm undoes the effect of any assignment if the continuation passed to it cannot be performed.

```
class PrintContinuation : public Continuation {
private:
   Expr val;

public:
   PrintContinuation(Expression * x) { val = x; }
   virtual void free() { val = 0; }
   virtual int withContinuation(Continuation *);
};

int PrintContinuation::withContinuation(Continuation * future)
{
   // see if we are a symbol, if so print it out
   Symbol * s = val()→isSymbol();
   if (s) {
      printf("%s\n", s→chars());
      return future→withContinuation(nothing);
      }
   return 0;
}
```

Figure 8.5: The print relation

Thus the future that is passed to the or relation may be invoked several times before we finally find a sequence of assignments that works.

The and relation is perhaps the most interesting. To understand this let us first take the case of only two relations, which we will call *rel1* and *rel2*. Let f represent the continuation we wish to evaluate if the and relation is successful. What then is the future we should pass to the first relation? If the first relation is successful, we want to evaluate the second relation and then the continuation. Thus the future for the first relation is the composition of the future for the second relation and the original continuation. This can be written as rel2(f), but we must make it into a continuation, we we create a new datatype called a CompositionContinuation. Interestingly, this composition relation ignores *its* continuation, and is merely executed for its side effect. This is the future we want to pass to the first relation. We can generalize this to any number of arguments. For example the *and* of three arguments should return the value produced by rel1([rel2([rel3([f])])]), and so on.

The composition step is performed by the datatype ComposeContinuation, shown in Figure 8.8. As in our description, when a composition relation is evaluated it ignores the future it is provided with and merely returns the first relation provided with the second relation as its future. Having defined this, the and relation is a simple recursive invocation.

```
class UnifyContinuation : public Continuation {
private:
    Expr left;
    Expr right;
public:
    UnifyContinuation(Expression * a, Expression * b)
        { left = a; right = b; }
    virtual void free()
        { left = 0; right = 0; }
    virtual int withContinuation(Continuation *);
};

int UnifyContinuation::withContinuation(Continuation * future)
{
    PrologValue * a = left()→isPrologValue();
    PrologValue * b = right()→isPrologValue();

    // the following shouldn't ever happen, but check anyway
    if ((!a) || (!b)) {
        error("impossible", "missing prolog values in unification");
        return 0;
        }

    // now try unification
    PrologValue * c = 0;
    if (unify(c, a, b) && future→withContinuation(nothing))
        return 1;

    // didn't work, undo assignment and fail
    if (c)
        c→setUndefined();
    return 0;
}
```

Figure 8.6: The unification relation

```
class OrContinuation : public Continuation {
private:
    List relArgs;
public:
    OrContinuation(ListNode * args) { relArgs = args; }
    virtual void free() { relArgs = 0; }
    virtual int withContinuation(Continuation *);
};

int OrContinuation::withContinuation(Continuation * future)
{
    ListNode * args;
    // try each alternative in turn
    for (args = relArgs; ! args→isNil(); args = args→tail()) {
        Continuation * r = args→head()→isContinuation();
        if (! r) {
            error("or argument is non-relation");
            return 0;
            }
        if (r→withContinuation(future)) return 1;
        }
    // nothing worked
    return 0;
}
```

Figure 8.7: The or relation

```
class ComposeContinuation : public Continuation {
private:
    Expr left;
    Expr right;
public:
    ComposeContinuation(Expression * a, Expression * b)
        { left = a; right = b; }
    virtual void free()
        { left = 0; right = 0; }
    virtual int withContinuation(Continuation *);
};

int ComposeContinuation::withContinuation(Continuation * future)
{
    Continuation * a = left()→isContinuation();
    Continuation * b = right()→isContinuation();
    if ((! a) || (! b)) {
        error("compose with non relations??");
        return 0;
        }
    return a→withContinuation(b);
}

class AndContinuation : public Continuation {
private:
    List relArgs;
public:
    AndContinuation(ListNode * args)
        { relArgs = args; }
    virtual void free()
        { relArgs = 0; }
    virtual int withContinuation(Continuation *);
};

int AndContinuation::withContinuation(Continuation * future)
{
    ListNode * args;
    args = relArgs;
    Continuation * newrel = future;
    for (int i = args→length()−1; i >= 0; i−−)
        newrel = new ComposeContinuation(args→at(i), newrel);

    Expr p = newrel;    // for gc purposes
    int result = newrel→withContinuation(nothing);
    p = 0;
    return result;
}
```


Figure 8.8: The and relation

```
class UnifyOperation : public BinaryFunction {
public:
   virtual void applyWithArgs(Expr & target, ListNode * args,
      Environment *)
      { target = new UnifyContinuation(args→at(0), args→at(1)); }

};

class PrintOperation : public UnaryFunction {
public:
   virtual void applyWithArgs(Expr & target, ListNode * args,
      Environment *)
      { target = new PrintContinuation(args→at(0)); }
};

class AndOperation : public Function {
public:
   virtual void applyWithArgs(Expr & target, ListNode * args,
      Environment *)
      { target = new AndContinuation(args); }
};

class OrOperation : public Function {
public:
   virtual void applyWithArgs(Expr & target, ListNode * args,
      Environment *)
      { target = new OrContinuation(args); }
};
```

Figure 8.9: Building the Relations

You may have noticed that the class Continuation is not a subclass of class Function, and yet we have been discussing continuations as if they were functions. This is easily explained. Recall that evaluating a relation in our approach is a two-step process. First the relation is constructed, and in the second step the future is brought to life. The functional parts of each of the four relation-building operations are concerned only with the first part of this task. These are all trivial functions, shown in Figure 8.9.

The query statement is responsible for the construction and execution of the continuation corresponding to its argument. The function implementing the query statement is shown in Figure 8.10. A new environment is created prior to evaluating the arguments so that bindings created for new variables do not get entered into the global environment. Then the continuation is constructed,

```
void QueryStatement::apply(Expr&target, ListNode*args, Environment*rho)
{
    if (args→length() != 1) {
        target = error("wrong number of args to query");
        return;
        }

    // we make a new environment to isolate any new variables defined
    Env newrho = new Environment(emptyList, emptyList, rho);

    args→at(0)→eval(target, valueOps, newrho);

    Continuation * f = 0;
    if (target())
        f = target()→isContinuation();
    if (! f) {
        target = error("query given non-relation");
        return;
        }
    if (f→withContinuation(nothing))
        target = new Symbol("ok");
    else
        target = new Symbol("not ok");

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

Figure 8.10: Implementation of the query statement

simply by evaluating the argument. If this process is successful, the continuation is then executed, and if the continuation is successful the symbol ok is yielded as the result (and thus printed by the read-eval-print loop). If the continuation is not successful the symbol not-ok is generated.

The initialization function for the prolog interpreter (Figure 8.11) is one of the shortest we have seen. It is only necessary to create the two commands define and query, and the four relational-building operations.

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

    // make the empty relation
    nothing = new Continuation;

    // make the operators that are legal inside of relations
    Environment * rops = valueOps;
    rops→add(new Symbol("print"), new PrintOperation);
    rops→add(new Symbol(":=:"), new UnifyOperation);
    rops→add(new Symbol("and"), new AndOperation);
    rops→add(new Symbol("or"), new OrOperation);

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

Figure 8.11: Initialization of the Prolog interpreter

Possible Future Changes

The following list represents a few of the ideas that occurred to me as I was developing these interpreters for how things might be done differently. These are presented in no particular order. (Nor as any particularly grave criticism of the Kamin interpreters - I still think the book as a whole is very good).

- The C++ versions of the interpreters have an annoying habit of dumping core when an error occurs. Need to track this down and fix it.
- I would remove the while statement from the chapter 2 lisp interpreter. Students who do not have previous experience with Lisp often have a difficult time learning to program in a recursive fashion. For them the while statement is a crutch, and without it they would be forced to use the more Lisp-like features of the language.
- I would add functionals (called operators in APL) to chapter 3. Specifically I would make reduction take the function as an argument, and add inner and outer product. This would allow an easier transition to functional programming in the next section.
- I might be tempted to add a chapter before chapter 3 on Setl. This is another example of a language using large values, and allows a new and different problem domain to be discussed (namely logic).
- It would be nice to add call/cc to the scheme interpreter, but I don't exactly see how to do this right now. This is not quite as critical now that the Prolog interpreter uses continuations for its execution.
- I would remove the keyword "rep" from the CLU syntax, as it is unnecessary and its elimination simplifies the implementation.