Tree Abstractions for Programs

W. M. Waite

August 20, 1998

Abstract

Programs are often represented within a compiler by abstract trees. These abstractions take a standard form that can be specified by a context-free grammar. Standard techniques are available to implement the specifications by writing directly in a programming language, and there are tools that will generate the implementation automatically from the specifications.

This paper presents two implementations of the abstract tree for a simple straight-line programming language defined by Appel in his text *Modern Compiler Construction in Java*. The paper was generated from an Eli specification. Both of the implementations can also be generated from that specification.

Contents

1 Introduction

A compiler is a program that accepts text in some *source language* and produces equivalent text in some *target language*. Its behavior can be roughly described as follows:

1. Determine the structure of the source text and verify that it satisfies the rules of the source language.
2. Transform the structure of the source text to the structure of an equivalent target text.
3. Output the target text in a suitable form.

The structures of the source and target texts are often represented within the compiler by trees, because many properties of a construct depend on properties of that construct's components. Section 1.3 of the book *Modern Compiler Construction in Java* by Andrew W. Appel (Cambridge University Press, 1998) uses a simple, straight-line language to illustrate the relationship between an input text and the abstracting tree. Unfortunately, there are a number of inconsistencies in the example that obscure the nature of the relationship.

Appel's Figure 1.4 depicts the identifier strings and numbers as tree nodes. In his Grammar 1.3, however, these entities are represented by terminal symbols set in a special typeface. The word "print" is also set in the special typeface, but it does not appear in Figure 1.4. Finally, the operator characters (like "+") seem to play the same role in the grammar as "print", but their typeface differs from that of "print".

In Appel's Program 1.5, classes based on `Stm` and `Exp` are used to represent all of the interior nodes of Figure 1.4. Figure 1.4's leaves are all instances of other classes not defined in Program 1.5. Thus the classes defined in Program 1.5 correspond to *some* of the symbols and rules in Grammar 1.3, but not all: No class is defined for the symbol `Binop`, and hence there can be no classes to represent the rules having `Binop` on the left-hand side.

The consequence of these inconsistencies is that Appel's Grammar 1.3 does not constitute a reliable specification of the abstraction. From a software engineering point of view, that's a disaster. What it means is that the implementor of the specification is allowed to vary the code according to unstated criteria, and anyone attempting to maintain that code must examine it to understand it.

At the end of Chapter 1, Appel sets a programming exercise: implement a simple program analyzer and interpreter for the straight-line programming language. Since he does not want to worry about parsing the language at this point, he builds a tree for the following program by writing the necessary data constructors explicitly:

```
a := 5 + 3; b := ( print(a, a - 1), 10 * a ); print(b)
```

Each print statement results in a single line containing the values of the operand expressions, separated by spaces. Thus the output of the program should be:

```
8 7
80
```

This example is straightforward because no print statement has a nested print statement. The following example is more challenging for an interpreter:

```
a := 5 + 3; b := ( print(a, ( print(a / 4), a - 1 )), 10 * a ); print(b)
```

The output of this program should be:

```
2
8 7
80
```

Section 2 of this paper solves the programming exercise in C++, using the second program as a test. The C++ code was obtained by applying the following consistent implementation procedure to Appel's Grammar 1.3:

1. Define a class for all tree nodes.
2. Define a class for each nonterminal symbol of the grammar. Each nonterminal symbol class is a subclass of the tree node class.
3. Define a class for each rule. Each rule class is a subclass of the class for its left-hand-side nonterminal.
4. Each rule class has one field for each non-literal symbol on the right-hand side. The type of a field for a nonterminal symbol is a pointer to an object of that symbol's class. The type of a field for a terminal symbol is determined by the declaration of that symbol.

Literal symbols appearing on the right-hand side of a grammar rule carry no information, because they are fixed once the rule is known. They are used only in rules corresponding to phrases of some input text, to show the relationship between the textual delimiters and the components of the construct.

Once one has a notation for specifications and a consistent implementation procedure for that notation, a tool can be written to verify the specifications and apply the implementation procedure mechanically. This approach guarantees that the specification is complete and consistent, and that the implementation actually realizes it. Moreover, maintenance is shifted to a smaller, more readable description of the problem.

Eli provides tools that verify grammars describing trees and implement them as C data structures. Computations can be attached to a grammar specification, and the tools will generate code to carry out those computations on the data structure implementing the tree.

Section 2's solution to Appel's programming exercise is specified using these techniques in Section 3 of this paper. Because Section 2 and Section 3 solve the same problem in the same way, they provide a direct comparison between systematic hand coding and generation. Both solutions are executable, so that one can guarantee that no details have been omitted in either case.

Appel sketches a more ad-hoc solution to this problem based on the same general implementation technique, using Java as the implementation language. The inconsistencies mentioned earlier mean that his solution could differ slightly from the one presented here. Whether those differences would constitute enough of an improvement to offset the dangers of unsystematic hand coding is difficult to predict.

This paper was written using the *literate programming* style, in which code fragments are represented in the text by *macros*. Macros are numbered in order, and the macro(s) by which each macro is invoked is specified at the point of definition. Some of the macros are listed as being "attached to a product file" or "attached to a non-product file". The names of those macros are the names of the files that will be generated from the document. (The product/non-product distinction has to do with the way these files are treated by Eli.)

FunnelWeb was used to process this document.

2 A C++ Solution

The book *Design Patterns* (Addison-Wesley, Reading MA, 1995) is a collection of simple and elegant solutions to common problems in object-oriented software design. Designers gain leverage by re-using these solutions whenever they recognize the corresponding problems.

As shown in Section 2.1, representation of an abstract syntax tree is easily implemented by applying the "Composite" design pattern. Section 2.2 implements an interpreter by applying the "Visitor" pattern. Initialization and execution of the interpreter are discussed in Section 2.3. Finally, Section 2.4 collects the components and constructs the set of files making up the complete program.

2.1 Abstract Syntax Tree

The consistent implementation procedure for trees is a special case of the "Composite" design pattern. "Composite" is applicable to situations where complex structures are built from components, and these structures may themselves be components of other structures. It describes how to use recursive composition to avoid forcing clients to treat each component object specially.

A class that represents *all* components is the key to this design pattern. That class is provided by the "tree node" class created by step (1) of the consistent implementation procedure:

Class representing all tree nodes[1]:

```
class Node {
  public:
    virtual ~Node() {}
    Empty Accept method[9];
};
```

This macro is invoked in definition 24.

The purpose and implementation of the `Accept` method are discussed in Section 2.2.1.

Step (2) of the consistent implementation procedure requires one class for each nonterminal symbol of the grammar. Each nonterminal symbol class is a subclass of the tree node class, and all of them have exactly the same structure. The only thing that varies among the classes is the name:

Class definition for nonterminal[2](◇1):

```
class ◇1 : public Node {
  public:
    virtual ~◇1() {}
    Empty Accept method[9];
};
```

This macro is invoked in definition 3.

Here the parameter is the nonterminal symbol. Symbol classes must also accept the visitors that do the interpretation, as discussed in Section 2.2.1.

Classes representing nonterminal symbols[3]:

Class definition for nonterminal[2]('`Stm`')
Class definition for nonterminal[2]('`Exp`')
Class definition for nonterminal[2]('`ExpList`')
Class definition for nonterminal[2]('`Binop`')

This macro is invoked in definition 24.

The last two steps of the consistent implementation procedure are concerned with rule classes. Each rule class is a subclass of the class for its left-hand side symbol, and must have fields reflecting the rule's right-hand side symbols.

One of the inconsistencies in Appel's Grammar 1.3 lay in the specification of the right-hand side symbols. The following class definitions assume that `print`, `+`, `-`, `*` and `/` are all literal terminal symbols, while `id` and `num` are literal terminal symbols.

The `Binop` rule classes are the simplest because those rules have no right-hand-side symbols. All of them have identical structure, differing only in the rule name:

Class definition for Binop rule[4](◇1):

```
class ◇1 : public Binop {
  public:
    ◇1 () {}
    virtual ~◇1 () {}
    Prototype for the Accept method[8];
};
```

This macro is invoked in definition 6.

(The `Accept` method is explained in Section 2.2.1.)

The class definition for the assignment statement rule is more interesting, illustrating the way in which right-hand-side symbols of different types are represented:

Class definition for rule AssignStm[5]:

```
class AssignStm : public Stm {
  public:
    AssignStm (string arg1, Exp* arg2) { child1 = arg1; child2 = arg2; }
    virtual ~AssignStm () {}
    string Child1() { return child1; }
    Exp* Child2() { return child2; }
    Prototype for the Accept method[8];
  private:
    string child1; Exp* child2;
};
```

This macro is invoked in definition 6.

The fields implementing the right-hand-side symbols are private, and each is assigned a value by the class constructor. An access function is provided for each field so that routines outside the class can traverse the tree. There is little incentive to choose mnemonic names for the fields, since the primary documentation of the program is the grammar. The convention used here is positional, with the field names and access function names identical except for the case of the first letter.

Here is the complete set of rule classes:

Classes representing rules[6]:

```
class CompoundStm : public Stm {
  public:
    CompoundStm (Stm* arg1, Stm* arg2) { child1 = arg1; child2 = arg2; }
    virtual ~CompoundStm () {}
    Stm* Child1() { return child1; }
    Stm* Child2() { return child2; }
    Prototype for the Accept method[8];
  private:
    Stm* child1, *child2;
};
```

Class definition for rule AssignStm[5]

```
class PrintStm : public Stm {
  public:
    PrintStm (ExpList* arg1) { child1 = arg1; }
    virtual ~PrintStm () {}
    ExpList* Child1() { return child1; }
    Prototype for the Accept method[8];
  private:
    ExpList* child1;
};

class IdExp : public Exp {
  public:
    IdExp (string arg1) { child1 = arg1; }
    virtual ~IdExp () {}
    string Child1() { return child1; }
    Prototype for the Accept method[8];
  private:
    string child1;
};

class NumExp : public Exp {
  public:
    NumExp (int arg1) { child1 = arg1; }
    virtual ~NumExp () {}
    int Child1() { return child1; }
    Prototype for the Accept method[8];
  private:
    int child1;
};

class OpExp : public Exp {
  public:
    OpExp (Exp* arg1, Binop *arg2, Exp* arg3)
      { child1 = arg1; child2 = arg2; child3 = arg3; }
    virtual ~OpExp () {}
    Exp* Child1() { return child1; }
    Binop* Child2() { return child2; }
    Exp* Child3() { return child3; }
    Prototype for the Accept method[8];
  private:
    Exp* child1, *child3; Binop *child2;
};

class EseqExp : public Exp {
  public:
    EseqExp (Stm* arg1, Exp* arg2) { child1 = arg1; child2 = arg2; }
    virtual ~EseqExp () {}
    Stm* Child1() { return child1; }
    Exp* Child2() { return child2; }
    Prototype for the Accept method[8];
  private:
```

```
    Stm* child1; Exp* child2;
};

class PairExpList : public ExpList {
  public:
    PairExpList (Exp* arg1, ExpList* arg2) { child1 = arg1; child2 = arg2; }
    virtual ~PairExpList () {}
    Exp* Child1() { return child1; }
    ExpList* Child2() { return child2; }
    Prototype for the Accept method[8];
  private:
    Exp* child1; ExpList* child2;
};

class LastExpList : public ExpList {
  public:
    LastExpList (Exp* arg1) { child1 = arg1; }
    virtual ~LastExpList () {}
    Exp* Child1() { return child1; }
    Prototype for the Accept method[8];
  private:
    Exp* child1;
};
```

Class definition for Binop rule[4](‘`Plus`’)
Class definition for Binop rule[4](‘`Minus`’)
Class definition for Binop rule[4](‘`Times`’)
Class definition for Binop rule[4](‘`Div`’)

This macro is invoked in definition 24.

2.2 Interpret the Program

Interpretation is only one of the possible computations that might be carried out over an abstract syntax tree. These computations are all more or less independent of one another, and thus good software engineering practice requires them to be implemented as modules that are separate from the tree module and separate from each other.

The “Visitor” design pattern is applicable under these conditions. “Visitor” requires an abstract class that couples arbitrary computation classes to the tree. Each computation is then defined as a subclass of that abstract class. Section 2.2.1 shows how these relationships are established.

Simple expression evaluation is covered in Section 2.2.2, and Section 2.2.3 adds the implementations of variables and assignment. Print statements are the subject of Section 2.2.4, and the remaining rules are wrapped up in Section 2.2.5.

2.2.1 Visitors

`Visitor` is the superclass of all computation classes. Although it has no class relationship with any of the classes of the abstract tree, it and all of its subclasses are specific to that tree. Each abstract tree implementation must have its own visitor, because the methods of the visitor are in 1-to-1 correspondence to the rule classes of the tree:

Declare the abstract visitor class[7]:

```
class Visitor {
  public:
    virtual ~Visitor() {}
    virtual void VisitCompoundStm(CompoundStm*) = 0;
    virtual void VisitAssignStm(AssignStm*)     = 0;
    virtual void VisitPrintStm(PrintStm*)       = 0;
    virtual void VisitIdExp(IdExp*)             = 0;
    virtual void VisitNumExp(NumExp*)           = 0;
    virtual void VisitOpExp(OpExp*)             = 0;
    virtual void VisitEseqExp(EseqExp*)         = 0;
    virtual void VisitPairExpList(PairExpList*) = 0;
    virtual void VisitLastExpList(LastExpList*) = 0;
    virtual void VisitPlus(Plus*)               = 0;
    virtual void VisitMinus(Minus*)             = 0;
    virtual void VisitTimes(Times*)             = 0;
    virtual void VisitDiv(Div*)                 = 0;
};
```

This macro is invoked in definition 25.

Every tree class must be modified to accept visitor invocations by providing an `Accept` method. These methods all have the same prototype:

Prototype for the Accept method[8]:

```
virtual void Accept (Visitor*)
```

This macro is invoked in definitions 4, 5, 6, and 9.

Only the rule classes define bodies for their `Accept` methods. The general tree node class and all of the symbol classes define empty `Accept` methods:

Empty Accept method[9]:

Prototype for the Accept method[8] `= 0`

This macro is invoked in definitions 1 and 2.

`Accept` methods for the rule classes follow identical patterns:

Accept method definitions for rule classes[10]:

```
void CompoundStm        :: Accept (Visitor* v) { v->VisitCompoundStm(this); }
void AssignStm          :: Accept (Visitor* v) { v->VisitAssignStm(this); }
void PrintStm           :: Accept (Visitor* v) { v->VisitPrintStm(this); }
void IdExp              :: Accept (Visitor* v) { v->VisitIdExp(this); }
void NumExp             :: Accept (Visitor* v) { v->VisitNumExp(this); }
void OpExp              :: Accept (Visitor* v) { v->VisitOpExp(this); }
void EseqExp            :: Accept (Visitor* v) { v->VisitEseqExp(this); }
void PairExpList        :: Accept (Visitor* v) { v->VisitPairExpList(this); }
void LastExpList        :: Accept (Visitor* v) { v->VisitLastExpList(this); }
```

```
void Plus               :: Accept (Visitor* v) { v->VisitPlus(this); }
void Minus              :: Accept (Visitor* v) { v->VisitMinus(this); }
void Times              :: Accept (Visitor* v) { v->VisitTimes(this); }
void Div                :: Accept (Visitor* v) { v->VisitDiv(this); }
```

This macro is invoked in definition 26.

Suppose that a routine needs to perform a specific computation at a specific node of the tree, to which it has a pointer. The routine does not know what kind of tree node that pointer points to.

1. The routine invokes the node's `Accept` method, passing a pointer to the visitor subclass for the computation.
2. Every tree node is an instance of a rule class. The `Accept` method for that rule class is therefore the one invoked. That method invokes the corresponding method of the visitor subclass for the computation, passing a pointer to the tree node.
3. The method of the visitor subclass for the computation carries out the specific computation, accessing information from the node as required.

The declaration of the interpreter subclass of `Visitor` is almost identical to the definition of `Visitor` itself:

Declare the concrete interpreter subclass[11]:

```
class Interpreter : public Visitor {
  public:
    virtual void VisitCompoundStm(CompoundStm*);
    virtual void VisitAssignStm(AssignStm*);
    virtual void VisitPrintStm(PrintStm*);
    virtual void VisitIdExp(IdExp*);
    virtual void VisitNumExp(NumExp*);
    virtual void VisitOpExp(OpExp*);
    virtual void VisitEseqExp(EseqExp*);
    virtual void VisitPairExpList(PairExpList*);
    virtual void VisitLastExpList(LastExpList*);
    virtual void VisitPlus(Plus*);
    virtual void VisitMinus(Minus*);
    virtual void VisitTimes(Times*);
    virtual void VisitDiv(Div*);
  private:
    Declare the expression evaluation stack[12]
    Declare the print line stack[18]
    Declare the interpreter memory[15]
};
```

This macro is invoked in definition 27.

All of the method declarations are the same, but `Interpreter` must use private fields to pass information among its methods. These fields are explained in subsequent subsections.

2.2.2 Expression evaluation

Expressions are evaluated by the methods of the `Interpreter` class. Each evaluates the expression represented by a single rule node of the tree. All of these methods have the same prototype, which specifies a single parameter and no result. But expression evaluation produces a result — the value of the expression. Moreover, that result of evaluating an expression represented by a rule node must be used as an operand when evaluating the expression represented by it parent.

Consider an expression represented by an `OpExp` node. One way to evaluate such an expression would be to first invoke a routine to evaluate its left child, then a routine to invoke its right child. Finally, pass the resulting values to a routine that carries out the operation.

To implement this approach, all that is necessary is to provide mechanisms for passing arguments to a routine and returning results from that routine. Many current machines use a stack for these purposes: A caller pushes arguments onto the stack, the called routine removes those arguments and pushes its results. This strategy does not require that any explicit information be passed between invoker and invokee, so it can be used in the interpreter.

"Stack" is a so-called *container adaptor* in the C++ standard template library. That means that it changes one container into another, and by default the container to be adapted is a double-ended queue. There is no reason to change that default for this application.

Since the values of the expressions in the simple serial programming language are all integers, the stack elements should be integers:

Declare the expression evaluation stack[12]:

```
stack<int > ExpValues;
```

This macro is invoked in definition 11.

Here's how the interpreter evaluates expressions containing only constants and operators:

Interpret the NumExp and OpExp rules[13]:

```
void Interpreter::VisitNumExp(NumExp* node)
{ ExpValues.push(node->Child1());
}

void Interpreter::VisitOpExp(OpExp* node)
{ (node->Child1())->Accept(this);
  (node->Child3())->Accept(this);
  (node->Child2())->Accept(this);
}
```

This macro is invoked in definition 28.

Notice that `VisitOpExp` doesn't actually do any computation of its own. It simply visits its children in an appropriate order. The actual operation is performed by one of the subclasses of `Binop`, all of which have the same form:

Interpret any rule whose left-hand-side symbol is Binop[14](◇2):

```
void Interpreter::Visit◇1(◇1* node)
{ int right, left;
```

```
  right = ExpValues.top(); ExpValues.pop();
  left = ExpValues.top(); ExpValues.pop();
  ExpValues.push(left ◇2 right);
}
```

This macro is invoked in definition 28.

Here the first parameter is the name of a rule node (e.g. `Plus`) and the second is the corresponding C++ operator (e.g. `+`).

2.2.3 Variables and memory

Variables are represented by identifiers in the simple serial programming language. They can hold integer values, so the memory must be implemented by an *associative container* in which identifiers can be used as keys to access integers.

In the C++ standard template library, a *map* is the associative container that stores a value with each key. A map declaration needs a comparison function as well as the types of the key and the associated value:

Declare the interpreter memory[15]:

```
map<string, int, less<string> > Table;
```

This macro is invoked in definition 11.

(This declaration follows Appel in naming the interpreter's memory `Table`.)

Given `Table`, interpretation of the `AssignStm` and `IdExp` nodes is straightforward:

Interpret the AssignStm and IdExp rules[16]:

```
void Interpreter::VisitAssignStm(AssignStm* node)
{ (node->Child2())->Accept(this);
  Table[node->Child1()] = ExpValues.top(); ExpValues.pop();
}

void Interpreter::VisitIdExp(IdExp* node)
{ ExpValues.push(Table[node->Child1()]);
}
```

This macro is invoked in definition 28.

2.2.4 Print statements

On page 8 of *Modern Compiler Construction in Java*, Appel states that:

> $print(e_1, e_2, \ldots, e_n)$ displays the values of all the expressions, evaluated left to right, separated by spaces, terminated by a newline.

A problem arises when e_i contains a print statement. (An expression can be an `EseqExp`, which contains statements.) In that case, the semantics of the print statement seem to require that the output from the print statement inside e_i must occur on a previous line, and not be embedded in the line due to the current print statement.

Since the expressions are evaluated left to right, their values must be saved until the end of the print statement and then output as a single line of text. The obvious data structure to use for saving the values is a vector of integers. When the end of the print statement is reached, an iterator can be used to retrieve the values in the order in which they were inserted:

Interpret the PrintStm rule[17]:

```
void Interpreter::VisitPrintStm(PrintStm* node)
{ vector<int> line;
  vector<int>::iterator i;

  Compute all of the values on the print line[19]

  for (i = line.begin(); i != line.end(); i++) cout << *i << ' ';
  cout << '\n';
}
```

This macro is invoked in definition 28.

The expressions whose values are to be printed are not directly available at the `PrintStm` node. They appear only as children of `PairExpList` and `LastExpList` nodes. Thus the interpreter must use a stack to pass a pointer to `line` down through the tree to the `VisitPairExpList` and `VisitLastExpList` method invocations:

Declare the print line stack[18]:

```
stack<vector<int>* > PrintLine;
```

This macro is invoked in definition 11.

Note that if there is a print statement nested in the child of a print statement, the interpreter for that print statement will create and push a new vector before proceeding to evaluate its own child. Moreover, the complete line for the nested statement will be output before any of the elements of the line for the outer statement are printed.

Compute all of the values on the print line[19]:

```
PrintLine.push(&line); (node->Child1())->Accept(this); PrintLine.pop();
```

This macro is invoked in definition 17.

Each expression value in the print statement's expression list is simply attached to the current print line:

Interpret the PairExpList and LastExpList rules[20]:

```
void Interpreter::VisitPairExpList(PairExpList* node)
{ (node->Child1())->Accept(this);
  (PrintLine.top())->push_back(ExpValues.top()); ExpValues.pop();
  (node->Child2())->Accept(this);
}

void Interpreter::VisitLastExpList(LastExpList* node)
{ (node->Child1())->Accept(this);
  (PrintLine.top())->push_back(ExpValues.top()); ExpValues.pop();
}
```

This macro is invoked in definition 28.

2.2.5 Rules without computation

`VisitOpExp` didn't do any computation of its own, but simply sequenced the computations of its children. Both `VisitCompoundStm` and `VisitEseqExp` also have this property:

Interpret VisitCompoundStm and VisitEseqExp rules[21]:

```
void Interpreter::VisitCompoundStm(CompoundStm* node)
{ (node->Child1())->Accept(this);
  (node->Child2())->Accept(this);
}

void Interpreter::VisitEseqExp(EseqExp* node)
{ (node->Child1())->Accept(this);
  (node->Child2())->Accept(this);
}
```

This macro is invoked in definition 28.

Although these nodes seem uninteresting, they are vital as implementations of the left-to-right evaluation semantics of the language.

2.3 Initialization and Execution

Here is a transliteration of Appel's initialization code (*Modern Compiler Construction in Java*, page 12), modified as discussed in Section 1, for the expression

```
a := 5 + 3; b := ( print(a, ( print(a / 4), a - 1 )), 10 * a ); print(b)
```

Use explicit constructors to build a test tree[22]:

```
Stm* prog =
  new CompoundStm(
    new AssignStm("a", new OpExp(new NumExp(5), new Plus(), new NumExp(3))),
    new CompoundStm(
      new AssignStm(
        "b",
        new EseqExp(
          new PrintStm(
            new PairExpList(
              new IdExp("a"),
              new LastExpList(
                new EseqExp(
                  new PrintStm(
                    new LastExpList(
                      new OpExp(new IdExp("a"), new Div(), new NumExp(4))
                    )
                  ),
                new OpExp(new IdExp("a"), new Minus(), new NumExp(1))
                )
              )
            )
```

```
            ),
            new OpExp(new NumExp(10), new Times(), new IdExp("a"))
          )
        ),
        new PrintStm(new LastExpList(new IdExp("b")))
      )
    );
```

This macro is invoked in definition 29.

Once the tree has been constructed, it can be interpreted by instantiating an interpreter and invoking the `Accept` method of the tree root:

Interpret the test tree[23]:

```
Interpreter evaluate;

prog->Accept(&evaluate);
```

This macro is invoked in definition 29.

Termination of the `Accept` invocation indicates termination of the program being interpreted.

2.4 Program Files

The program files reflect the decomposition of the implementation into modules. Each module may have an interface specification and a body. Ancillary files to control the manufacture of the executable program and to provide instructions are also included.

This section gathers together the components of the C++ implementation of the straight-line program interpreter, combining them into files of the appropriate types according to the modular decomposition.

2.4.1 abstree.h

File `abstree.h` provides the interface for the module implementing the abstract syntax tree.

abstree.h[24]:

```
#ifndef ABSTREE_H
#define ABSTREE_H

#include <string>

using namespace std;

class Visitor;

```

Class representing all tree nodes[1]
Classes representing nonterminal symbols[3]
Classes representing rules[6]

```
#endif
```

This macro is attached to a product file.

The symbol `ABSTREE_H` is used to protect against successive inclusions of the material from a header file: `ABSTREE_H` will be defined at the first inclusion, and that definition will cause later inclusions to be omitted.

2.4.2 visitor.h

File `visitor.h` defines the visitor class for the abstract syntax tree.

visitor.h[25]:

```
#ifndef VISITOR_H
#define VISITOR_H

#include "abstree.h"

using namespace std;

```

Declare the abstract visitor class[7]

```
#endif
```

This macro is attached to a product file.

2.4.3 abstree.cc

File `abstree.cc` implements the `Accept` operations that take a visitor as an argument.

abstree.cc[26]:

```
#include "abstree.h"
#include "visitor.h"

```

Accept method definitions for rule classes[10]

This macro is attached to a product file.

2.4.4 interpreter.h

File `interpreter.h` is the definition of the interpreter class.

interpreter.h[27]:

```
#ifndef INTERPRETER_H
#define INTERPRETER_H

#include <string>
#include <vector>
#include <stack>
#include <map>
#include "visitor.h"

using namespace std;

```

Declare the concrete interpreter subclass[11]

```
#endif
```

This macro is attached to a product file.

2.4.5 interpreter.cc

File `interpreter.cc` implements the methods of the interpreter class.

interpreter.cc[28]:

```
#include <iostream>
#include "abstree.h"
#include "interpreter.h"

Interpret VisitCompoundStm and VisitEseqExp rules[21]
Interpret the AssignStm and IdExp rules[16]
Interpret the PrintStm rule[17]
Interpret the NumExp and OpExp rules[13]
Interpret the PairExpList and LastExpList rules[20]
Interpret any rule whose left-hand-side symbol is Binop[14]('Plus','+')
Interpret any rule whose left-hand-side symbol is Binop[14]('Minus','-')
Interpret any rule whose left-hand-side symbol is Binop[14]('Times','*')
Interpret any rule whose left-hand-side symbol is Binop[14]('Div','/')
```

This macro is attached to a product file.

2.4.6 try.cc

File `try.cc` builds the tree and invokes the interpreter to evaluate it.

try.cc[29]:

```
#include "abstree.h"
#include "interpreter.h"

int
main()
{ Use explicit constructors to build a test tree[22]
  Interpret the test tree[23]
}
```

This macro is attached to a product file.

2.4.7 Makefile

File `Makefile` controls the process of manufacturing the program from the source files.

Makefile[30]:

```
EXE     = trycpp

SRCS    = try.cc abstree.cc interpreter.cc

HDRS    = abstree.h visitor.h interpreter.h

CXX     = g++
CXXFLAGS        =
OBJS    = $(addsuffix .o,$(basename $(SRCS)))
DEPS    = $(addsuffix .dep,$(basename $(SRCS)))
```

```
all:    trycpp
        ./trycpp

$(EXE):         $(OBJS)
        $(CXX) -o $(EXE) $(OBJS)

clean:
        rm -f $(OBJS)

clobber:        clean
        rm -f $(EXE) try.dep

try.dep: $(SRCS) $(HDRS)
        $(CXX) $(CXXFLAGS) -MM $(SRCS) >try.dep

include try.dep
```

This macro is attached to a product file.

2.4.8 README.cpp

A README file gives instructions about how to use the contents of a directory to reach some particular goal.

README.cpp[31]:

```
This directory contains a set of C++ files that implement a solution to
the programming exercise in Chapter 1 of Appel's book "Modern Compiler
Construction in Java" (http://www.cs.princeton.edu/~appel/modern/java/).
The implementation was tested with gcc-2.7.2.

To obtain the solution, run the following command in this directory:

        make

The result should be an executable file "trycpp" and the following three
lines of output:

2
8 7
80
```

This macro is attached to a product file.

3 A Generated Solution

This section describes the complete set of specifications used to generate a solution to Appel's interpreter problem. It is a single FunnelWeb file from which the Eli system can generate either the executable solution or the set of specification files.

Section 3.1 specifies the abstract syntax tree for the language without reference to any particular computation. The material there is simply transliterated from Appel's book. Computations to carry out the

interpretation are attached to appropriate nodes in Section 3.2. An abstract data type is defined in Section 3.3 to map identifiers to the integer values assigned to them. Finally, Section 3.4 describes the initialization code and Section 3.5 produces the specification files.

3.1 Abstract syntax tree

LIDO is a specification language for abstract syntax trees. Here is a LIDO specification corresponding to Appel's Grammar 1.3:

Abstract syntax tree[32]:

```
RULE CompoundStm: Stm     ::= Stm ';' Stm             END;
RULE AssignStm:   Stm     ::= id ':=' Exp             END;
RULE PrintStm:    Stm     ::= 'print' '(' ExpList ')' END;
RULE IdExp:       Exp     ::= id                      END;
RULE NumExp:      Exp     ::= num                     END;
RULE OpExp:       Exp     ::= Exp Binop Exp           END;
RULE EseqExp:     Exp     ::= '(' Stm ',' Exp ')'     END;
RULE PairExpList: ExpList ::= Exp ',' ExpList         END;
RULE LastExpList: ExpList ::= Exp                     END;
RULE Plus:        Binop   ::= '+'                     END;
RULE Minus:       Binop   ::= '-'                     END;
RULE Times:       Binop   ::= '*'                     END;
RULE Div:         Binop   ::= '/'                     END;
```

This macro is invoked in definition 49.

Each rule is bracketed by the delimiters `RULE` and `END;`, the rule's name is given as a label rather than a parenthesized suffix, and Appel's `->` is rendered as `::=`.

Grammar 1.3 doesn't define the type of the objects used to represent non-literal terminal symbols (here `id` and `num`), nor does it provide any handle for the root of the tree. Both of these deficiencies must be repaired:

Complete Appel's tree specification[33]:

```
RULE Axiom: Program ::= Stm END;

TERM id: CharPtr;
TERM num: int;
```

This macro is invoked in definition 49.

LIDO requires that every type be represented by an identifier. Thus it is not possible to use the type `char*` directly in a LIDO specification. `CharPtr` is used here to represent `char*`, and must be declared:

*Representation of of char**[34]:

```
typedef char* CharPtr;
```

This macro is invoked in definition 52.

3.2 Interpreting the Program

There are two kinds of dependence exhibited by computations over this tree: dependence of an operator upon its operands, and dependence due to side effects on some shared object. Computation of expression values (Section 3.2.1) illustrates operator/operand dependence. In Appel's language, the only two shared objects manipulated by programs are the memory and the output stream. Section 3.2.2 shows how the dependences arising from those objects is taken into account.

3.2.1 Expression evaluation

One of the properties of an expression is its value. Every expression has a value, and this section deals only with expressions whose values are determined by local information. A binary operator is regarded as a function taking two values called `Left` and `Right` and delivering the result. All values are integers.

Expression evaluation[35]:

```
ATTR Value, Left, Right: int;

RULE: Exp ::= num
COMPUTE
  Exp.Value=num;
END;

RULE: Exp ::= Exp Binop Exp
COMPUTE
  Exp[1].Value=Binop.Value;
  Binop.Left=Exp[2].Value;
  Binop.Right=Exp[3].Value;
END;

RULE: Exp ::= '(' Stm ',' Exp ')'
COMPUTE
  Exp[1].Value=Exp[2].Value;
END;
```

This macro is defined in definitions 35 and 37.
This macro is invoked in definition 49.

The dependences among computations in these rules are direct data dependences. For example, the value of a dyadic expression depends on the result of the binary operator because there is a direct assignment of one to the other.

All of the dyadic operations follow a general pattern, which can be described once. Again, the dependence of the result value on the left and right operand values is clear because it is the result of a function having them as arguments:

Implementation of a binary operator[36](⋄2):

```
RULE: Binop ::= ⋄1
COMPUTE
  Binop.Value=⋄2(Binop.Left,Binop.Right);
END;
```

This macro is invoked in definition 37.

The first parameter is the operator indication, and the second is the built-in LIDO operation implementing the corresponding operation on integer values:

Expression evaluation[37]:

Implementation of a binary operator[36](‘’+’’,‘ADD’)
Implementation of a binary operator[36](‘’-’’,‘SUB’)
Implementation of a binary operator[36](‘’*’’,‘MUL’)
Implementation of a binary operator[36](‘’/’’,‘DIV’)

This macro is defined in definitions 35 and 37.
This macro is invoked in definition 49.

3.2.2 Left-to-right evaluation

The left-to-right evaluation required by the informal semantics of Appel's language need not be strictly followed, provided that the results are those that would be obtained with left-to-right evaluation. Expressions can therefore be evaluated in any order, provided that all of the memory accesses are properly sequenced.

Memory operations depend upon one another because they manipulate a shared data structure. The LIDO `CHAIN` implements a depth-first, left-to-right dependence relation, so this construct can be used to enforce the sequence of memory operations.

The `MemoryDep` chain corresponds to the memory of the interpreter. It does not carry a value from one computation to another, as do the `Value`, `Left` and `Right` attributes. Therefore it should be declared of type `VOID`:

Left-to-right evaluation[38]:

```
CHAIN MemoryDep: VOID;

RULE: Program ::= Stm
COMPUTE
  CHAINSTART Stm.MemoryDep="done";
END;

RULE: Stm ::= id ':=' Exp
COMPUTE
  Stm.MemoryDep=Store(id,Exp.Value) <- Exp.MemoryDep;
END;

RULE: Exp ::= id
COMPUTE
  Exp.Value=Fetch(id) <- Exp.MemoryDep;
  Exp.MemoryDep=Exp.Value;
END;
```

This macro is invoked in definition 49.

Note that each computation is “inserted” into the chain – the computation depends on the chain (via `<-`), and the subsequent chain depends on the computation (via `=`). Thus the computations are ordered along

a depth-first, left-to-right traversal of the tree. That tree traversal corresponds to a text-order traversal if none of the text's phrases were re-ordered when the tree was built.

A `VOID` chain has no physical existence in the final implementation, and therefore it occupies no space and requires no computation time. It's only purpose is to make dependence explicit so that the computations can be arranged in the proper order.

`Store` and `Fetch` provide the actual memory access. They are exported by an abstract data type whose implementation is given in Section 3.3. This illustrates the normal division of labor for such specifications: LIDO is concerned with the global dependence among computations that are defined by abstract data types.

Dependence among print statements is a consequence of the output device, not the memory. Thus that dependence can be enforced using another chain corresponding to the output device:

Print statement sequencing[39]:

```
CHAIN OutputDep: VOID;

RULE: Program ::= Stm
COMPUTE
  CHAINSTART Stm.OutputDep="done";
END;

RULE: Stm ::= 'print' '(' ExpList ')'
COMPUTE
  Stm.OutputDep=
    PTGOut(
      PTGLine(
        CONSTITUENTS Exp.Value SHIELD Exp
        WITH (PTGNode, PTGSeq, PTGValue, PTGNull))) <- ExpList.OutputDep;
END;
```

This macro is invoked in definition 49.

The actual construction of the line to be printed is treated as a problem of structured output: A `Value` is an integer, two `Value`s can be assembled into a `Seq` by separating them with a space, and a `Value` or `Seq` resulting from the argument(s) of a print statement constitutes a `Line` that should be terminated by a newline character. These statements can be formalized in the specification language of a pattern-based text generator

Output patterns[40]:

```
Value: $ int
Seq:   $ { " " } $
Line:  $ "\n"
```

This macro is invoked in definition 50.

For each pattern, the generator will create a function whose name is `PTG` followed by the name of the pattern. That function has a number of arguments based on the `$` markers in the pattern, and returns an object of type `PTGNode`. A built-in function `PTGOut` writes the value specified by its argument to the standard output stream.

`CONSTITUENTS` is a *remote attribution* that reaches down the tree to gather values. In this example it seeks all `Value` attributes of `Exp` nodes that are not `SHIELD`ed by other `Exp` nodes. It will apply `PTGValue` to each such attribute to obtain a value of type `PTGNode`. It will then use `PTGSeq` to combine adjacent values. If there is a subtree containing no `Exp` nodes, `CONSTITUENTS` will use `PTGNull` to generate a value of type `PTGNode`. (`PTGNull` is a built-in function that creates a `PTGNode` representing the empty string.)

3.3 Implementing the Memory

In Section 3.2.2, two operations were used to access memory:

Operations exported by the memory abstract data type[41]:

```
extern void Store(CharPtr, int);
extern int  Fetch(CharPtr);
```

This macro is invoked in definition 52.

This section defines an abstract data type that implements those operations by using library facilities and a bit of C code.

The Eli library provides facilities to recognize identical identifiers, map identifiers to entities, and associate properties with entities. A memory for the interpreter can be easily implemented using such facilities, although they offer considerably more power than this simple application requires. The first step is to make the identifier recognition and mapping operations part of the specification:

Include appropriate library modules[42]:

```
$/Scan/idn.specs
$/Name/envmod.specs
```

This macro is invoked in definition 55.

Since any arbitrary set of properties can be associated with an entity, those desired for this application must be specified. A special property definition language is used for this purpose:

Declare the "Value" property with integer values[43]:

```
Value: int;
```

This macro is invoked in definition 51.

Given these facilities, the C implementation of the abstract data type is quite simple:

Implementation of the memory module[44]:

```
Environment env;

void
Store(char* id, int val)
{ int sym, class = 1;

  mkidn(id, strlen(id), &sym, &class);
  ResetValue(DefineIdn(env, sym), val);
}

int
Fetch(char* id)
{ int sym, class = 1;

  mkidn(id, strlen(id), &sym, &class);
  return GetValue(DefineIdn(env, sym),0);
}
```

This macro is invoked in definition 53.

The variable `env` must be initialized before the `Store` and `Fetch` operations can be used:

Initialize the memory module[45]:

```
env = NewEnv();
```

This macro is invoked in definition 48.

3.4 Initialization and Execution

Tree construction and execution of the interpreter are closely bound to the generator that processed the specifications for the tree and the computations. That generator exports `NODEPTR` as the type of the tree node. `LIGA_ATTREVAL` is the operation that should be applied to the root to carry out the specified computations, but the generator does not export its prototype:

Make the prototype of the evaluator available[46]:

```
extern void LIGA_ATTREVAL(NODEPTR);
```

This macro is invoked in definition 53.

The generator also constructs a function for each rule. Its name is the rule name prefixed with `Mk`, and it has one more parameter than the number of right-hand side symbols (excluding literal terminals). The additional parameter is the first, and specifies the text coordinates (line and column) associated with the node. Here is an example of a rule and the prototype of the function generated from it:

```
RULE AssignStm:  Stm ::= id ':=' Exp END;
NODEPTR MkAssignStm(CoordPtr, CharPtr, NODEPTR)
```

`CoordPtr` is a pointer to the data type representing a coordinate Note that the type of a parameter corresponding to a nonliteral terminal symbol is the type declared as the value of that symbol, while the type of a parameter corresponding to a nonterminal symbol is always `NODEPTR`.

Finally, the generator exports an initialization function `InitTree` taking no parameters and delivering no result. `InitTree` must be invoked before invoking any of the node construction functions.

Given this information, construction of the tree is tedious but straightforward:

Code to construct the tree[47]:

```
NODEPTR tree;

InitTree();

tree =
  MkAxiom(NoPosition,
    MkCompoundStm(NoPosition,
      MkAssignStm(NoPosition,
        "a",
        MkOpExp(NoPosition,
          MkNumExp(NoPosition,5),
          MkPlus(NoPosition),
```

```
        MkNumExp(NoPosition,3)
      )
    ),
    MkCompoundStm(NoPosition,
      MkAssignStm(NoPosition,
        "b",
        MkEseqExp(NoPosition,
          MkPrintStm(NoPosition,
            MkPairExpList(NoPosition,
              MkIdExp(NoPosition,"a"),
              MkLastExpList(NoPosition,
                MkEseqExp(NoPosition,
                  MkPrintStm(NoPosition,
                    MkLastExpList(NoPosition,
                      MkOpExp(NoPosition,
                        MkIdExp(NoPosition,"a"),
                        MkDiv(NoPosition),
                        MkNumExp(NoPosition,4)
                      )
                    )
                  ),
                  MkOpExp(NoPosition,
                    MkIdExp(NoPosition,"a"),
                    MkMinus(NoPosition),
                    MkNumExp(NoPosition,1)
                  )
                )
              )
            )
          ),
          MkOpExp(NoPosition,
            MkNumExp(NoPosition,10),
            MkTimes(NoPosition),
            MkIdExp(NoPosition,"a")
          )
        )
      ),
      MkPrintStm(NoPosition,MkLastExpList(NoPosition,MkIdExp(NoPosition,"b")))
    )
  )
);
```

This macro is invoked in definition 53.

Once the tree has been constructed, `LIGA_ATTREVAL` can be invoked on it to carry out the specified computations. In this case, however, those computations depend on the memory ADT and that ADT must be initialized:

Code to carry out the computations over the tree[48]:

```
    Initialize the memory module[45]
    LIGA_ATTREVAL(tree);
```

This macro is invoked in definition 53.

3.5 Specification Files

A specification for the Eli system is made up of a number of files written in different languages. Each language is designed for a specific problem class, and is intended to reflect the "natural" way in which problems in that class are described by humans. The disadvantage of this approach is that a person wishing to use Eli must learn more than one language; the advantage is that specifications are more easily written and understood.

This section gathers together the components of the Eli specification for the straight-line program interpreter, combining them into files of the appropriate types.

3.5.1 straight.lido

A type-`lido` file associates computations with nodes of the abstract syntax tree.

straight.lido[49]:

Abstract syntax tree[32]
Complete Appel's tree specification[33]
Left-to-right evaluation[38]
Print statement sequencing[39]
Expression evaluation[35]

This macro is attached to a product file.

3.5.2 straight.ptg

A type-`ptg` file defines patterns for structured output.

straight.ptg[50]:

Output patterns[40]

This macro is attached to a product file.

3.5.3 straight.pdl

A type-`pdl` file defines the set of properties that can be associated with entities.

straight.pdl[51]:

Declare the ''Value'' property with integer values[43]

This macro is attached to a product file.

3.5.4 straight.h

A type-`h` file defines the interface for the handwritten C code.

straight.h[52]:

```
#ifndef DRIVER_H
#define DRIVER_H

Representation of of char*[34]
Operations exported by the memory abstract data type[41]

#endif
```

This macro is attached to a product file.

3.5.5 straight.c

A type-`c` file contains handwritten C code that is to be combined with the generated code to form the complete processor.

straight.c[53]:

```
#include "csm.h"
#include "envmod.h"
#include "pdl_gen.h"
#include "treecon.h"

Make the prototype of the evaluator available[46]
Implementation of the memory module[44]

int
main()
{ Code to construct the tree[47]
  Code to carry out the computations over the tree[48]
  return 0;
}
```

This macro is attached to a product file.

3.5.6 straight.head

A type-`head` file allows one to specify header files and C-preprocessor macros needed by the generated code.

straight.head[54]:

```
#include "straight.h"
```

This macro is attached to a product file.

3.5.7 straight.specs

A type-`specs` file lists Eli library modules needed to support the specification.

straight.specs[55]:

```
Include appropriate library modules[42]
```

This macro is attached to a product file.

3.5.8 Odinfile

An `Odinfile` controls the construction of a processor by Eli, just as a `Makefile` controls the construction of a processor by make.

Odinfile[56]:

```
%tryeli.specs == <<END
  straight.c
  straight.h
  straight.head
  straight.lido
  straight.pdl
  straight.ptg
  straight.specs

END

tryeli ! == %tryeli.specs +nomain +parser=none :exe
```

This macro is attached to a product file.

3.5.9 README.eli

A README file gives instructions about how to use the contents of a directory to reach some particular goal.

README.eli[57]:

```
This directory contains a set of specification files for use with Eli
(http://eli-project.sourceforge.net/).  They implement a solution to the
programming exercise in Chapter 1 of Appel's book "Modern Compiler
Construction in Java" (http://www.cs.princeton.edu/~appel/modern/java/).

In addition to Eli, you will need access to a compiler for ANSI C.  (The
implementation was tested with gcc-2.7.2.)

To obtain the solution, run the following command in this directory:

        eli tryeli

The result should be an executable file "tryeli" and the following three
lines of output:

2
8 7
80
```

This macro is attached to a product file.