Tree Abstractions for Programs

W. M. Waite

November 22, 2003

Abstract

Programs are often represented within a compiler by abstract trees. The first step in translating a program is then to build the abstract tree from the linear source text. Standard techniques are available to specify this process. Tools can implement the tree-building code directly from the specifications, or routines can be written by hand in a programming language.

This paper presents two implementations of the tree constructor 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.
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.

In Tree Abstractions for Programs (http://ece-www.colorado.edu/ecen4553/Reference/abstr/straight/) I gave two solutions to a programming exercise Appel set at the end of Chapter 1: implement a simple program analyzer and interpreter for the straight-line programming language. Appel did not want to worry about parsing the language at that point, so the programming exercise required that a tree be built for the following program by writing the necessary data constructors explicitly:

\[
a := 5 + 3; \ b := (\ \text{print}(a, a - 1), 10 \ast a); \ \text{print}(b)
\]

This document solves the structural analysis problem for the straight-line programming language, assuming the previously-developed abstract tree and interpreter.

Section 2 specifies the lexical and syntactic analysis problems, based on the definition of the abstract syntax tree given by Appel and formalized in Tree Abstractions for Programs. Eli accepts these specifications, in conjunction with those of the abstract syntax tree and its interpretation, and constructs a complete processor.

Section 3 applies the systematic hand-coding techniques discussed in most compiler construction textbooks to implement the specified scanner and parser in C++. The resulting programs can be combined with the C++ classes defining the abstract syntax tree and its interpreter to yield a complete processor.

Because Section 2 and Section 3 solve the same problem in the same way, they provide a direct comparison between generation and systematic hand coding. Both solutions are executable, so that one can guarantee that no details have been omitted in either case.

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 Generated Solution

This section describes the complete set of specifications used to generate a complete 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 2.1 recalls the abstract syntax tree definition and shows how to specify the necessary structural analysis. The interface conventions among the lexical analyzer, parser and tree constructor imposed by Eli require that all basic symbol values be expressed as integers. That requirement forces some minor changes in the specification for the interpreter, as explained in Section 2.2. Finally, Section 2.3 produces the specification files. The unchanged specifications from the solution to the previous problem are incorporated there so that the extent of the modifications is clear.
2.1 The Structural Analysis Problem

The problem of structural analysis is to build the abstract tree representation of a program from the linear text of that program. Here's the definition of the abstract tree representation for Appel's simple straight-line programming language. This LIDO specification was developed as part of a previous solution to the programming problem posed at the end of Chapter 1 of [Modern Compiler Construction in Java](Modern Compiler Construction in Java):

Previous abstract syntax tree definition[1]:

```
RULE Axiom: Program ::= Stm END;
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[10]

A compiler usually uses two techniques to build the tree from text:

- **Lexical analysis**: Grouping character sequences into meaningful units called “basic symbols” and attaching values to some of those.

- **Syntactic analysis**: Extracting the abstract structure from the sequence of basic symbols and forming the tree.

When the design of the structural analyzer begins with a tree grammar, it is most convenient to deal with the syntactic analysis first. Section 2.1.1 explains the approach and develops the necessary specifications. Once syntactic analysis has been completely specified, specifications for the lexical analysis task follow as described in Section 2.1.2.

2.1.1 Syntactic analysis

The phrase structure of an input text is usually specified by a context-free grammar. Since the phrase structure of the input text mirrors the structure of the abstract program tree, and the abstract program tree structure is specified by a context-free grammar, why is the syntactic analysis not already specified?

Unfortunately, the grammar used to describe the abstract syntax tree is usually ambiguous. It describes a structure, but it does not fix the representation of a text by that structure. Here's a simple text written in the straight-line programming language:

```
a := 5; b := 10; print(a, b)
```

Using the abstract grammar, the structure of this text could be described as:
Axiom(
  CompoundStm(
    CompoundStm(AssignStm(IdExp(a),5), AssignStm(IdExp(b),10)),
    PrintStm(PairExpList(IdExp(a),LastExpList(IdExp(b))))
  )
)

Unfortunately it could also be described as:

Axiom(
  CompoundStm(
    AssignStm(IdExp(a),5),
    CompoundStm(
      AssignStm(IdExp(b),10),
      PrintStm(PairExpList(IdExp(a),LastExpList(IdExp(b))))
    )
  )
)

This ambiguity can be resolved by providing additional grammar rules for the phrase structure of the text. The additional rules form a part of what is known as the concrete syntax:

**Rules to resolve the statement sequence ambiguity[2]:**

\[\text{StmList} ::= \text{Stm} / \text{StmList} \text{;} \text{'} \text{Stm} \text{.}\]

\[\text{Program} ::= \text{StmList} .\]
\[\text{Exp} ::= (\text{StmList} ,\text{' }\text{Exp} )'.\]

This macro is invoked in definition[11]

The general strategy is to define a new symbol, in this case \text{StmList}, to represent the sequence. Its definition is such that the first of the two phrase structures is always selected, resolving the ambiguity. All instances of \text{Stm} appearing on the right-hand sides of existing rules are then replaced by \text{StmList}.

These additional rules are only used to resolve an ambiguity in converting from a textual representation to the abstract program tree. \text{StmList} was introduced solely for this purpose; semantically, it’s identical to \text{Stm}. When the phrases described by the above rules are recognized, the syntactic analyzer should build \text{CompoundStm}, \text{Axiom} and \text{EseqExp} nodes. To make that happen, we need to provide the additional information that phrases named \text{StmList} should be considered to be \text{Stm} nodes:

**Map StmList phrases to Stm nodes[3]:**

\[\text{Stm} ::= \text{StmList} .\]

This macro is invoked in definition[12]

A second ambiguity in the tree grammar involves expressions. Issues of operator precedence and association are ignored, so there is no way to tell whether \text{b} is an operand of + or * in \text{a+b*c}. This is a well-understood problem, with a general solution that is basically the same as the solution to the statement list ambiguity: Define a set of symbols representing expressions with operators at different precedence levels and write rules intern of those symbols. A complete explanation of the process can be found in Section 4.3 of *An Introduction to Compiler Construction*, by Waite and Carter (HarperCollins, New York, 1993). Here is the result for Appel’s straight-line programming language:
**Rules to resolve the operand ambiguity**[4]:

```
Exp ::= Term / Exp Addop Term .
Term ::= Primary / Term Mulop Primary .
Primary ::= id / num .
Addop ::= '+' / '-' .
Mulop ::= '*' / '/' .
```

This macro is invoked in definition[11]

The symbols introduced here serve only to disambiguate the derivation process, and do not contribute to the semantics of the language. Thus additional equivalences must be defined:

**Map all expressions to Exp and operators to Binop**[5]:

```
Exp ::= Term Primary .
Binop ::= Addop Mulop .
```

This macro is invoked in definition[12]

It is important to realize that the specifications appearing in this section are the only ones needed to define the syntactic analysis task for the straight-line language. There is no need to write "semantic actions" that build tree nodes. Eli derives those actions by comparing the concrete and abstract rules, using the mapping specifications and general pattern-matching techniques. There is also no need to write a complete grammar for the phrase structure, because Eli can augment what was written with rules from the tree grammar where necessary.

### 2.1.2 Lexical analysis

The terminal symbols of the tree grammar for the straight-line programming language actually include all of the basic symbols. (This is not always the case, but if some terminal symbols are omitted from the tree grammar they must be introduced by the concrete grammar developed according to the techniques of Section 2.1.1.) Literal terminal symbols are completely defined by the grammar, and therefore need not be specified again. There is no description of the internal structure of the non-literal terminal symbols, however, so this must be given separately.

Lexical analysis must also establish values for basic symbols represented by non-literal terminals. The method by which the a value is to be established must therefore be specified. Here’s a **lexical analyzer specification** for the two non-literal terminals of the straight-line programming language:

**Specifications of the non-literal terminal symbols**[6]:

```
id: $[A-Za-z]+$ [mkidn]
num: $[0-9]+$ [mkint]
```

This macro is invoked in definition[13]

The name of the symbol is written at the beginning of a line, followed by a colon. A **regular expression** introduced by $, defines the set of character sequences acceptable as instances of the basic symbol. Finally, the name of a **token processor** enclosed in parentheses provides the method for establishing the basic symbol’s value. (A token processor is simply a routine that obeys a particular interface.)
Whatever value is established for a particular basic symbol, it must be passed through the parsing process and finally stored in the tree representing the program. The interfaces among the components generated by Eli dictate that the value must be an integer. No generality is lost by this restriction, because Eli provides \textbf{an unbounded vector of strings} in which to store the textual representation of the basic symbol. By storing the basic symbol’s string in this array, and making the value of the basic symbol the (integer) index of that string, the user guarantees that all available information is preserved.

There are a number of common ways to determine a value for a basic symbol, and token processors implementing them are included in Eli’s library. For example, \texttt{mkidn} stores the character form of the basic symbol in Eli’s string vector and makes the resulting index the value. Only one copy of each distinct string is stored, all appearances of that string getting the same index value. Thus all of the instances of a particular identifier in a program will have the same value.

Integer denotations like \texttt{num} are given their base-10 numeric value by the token processor \texttt{mkint}. This conversion is subject to the range limitation of the integer representation on the machine running the generated processor.

\subsection*{2.2 Interpreting the Program}

The only effect the addition of a structural analyzer has on the interpretation involves the value of the non-literal terminal symbol \texttt{id}. In the previous specification that value was a string, now it is an integer. That requires a change in the interface to the memory abstract data type:

\textit{Operations exported by the memory abstract data type\cite{7}}:

\begin{verbatim}
extern void Store(int, int);
extern int Fetch(int);
extern void Init(void);
\end{verbatim}

This macro is invoked in definition\cite{7}.

Notice that an initialization operation has been added. This is because the normal main program supplied by Eli, which assumes that a tree is to be built from text, is now appropriate. Thus this specification will \textit{not} specify a main program. The memory abstract data type becomes a closed module, which must be initialized. (The previous specification could, of course, have used a closed module for the memory abstract data type. A more open implementation was chosen to minimize the number of files.)

\textit{Implementation of the memory module\cite{8}}:

\begin{verbatim}
Environment env;

void Init(void)
{ env = NewEnv(); }

void Store(int sym, int val)
{ SetValue(DefineIdn(env, sym), val); }

int Fetch(int sym)
\end{verbatim}
This macro is invoked in definition 16.

This module is significantly simpler than the previous implementation. The reason is that the environment operations expect identifiers to be represented by unique integers. Thus the previous implementation had to use \texttt{mkidn} to obtain a unique integer for each string. That call is now done as part of the lexical analysis of identifiers.

The initialization operation must be invoked before the \texttt{Store} and \texttt{Fetch} operations can be used:

\textit{Initialize the memory module[9]}:

\begin{verbatim}
Init();
\end{verbatim}

This macro is invoked in definition 16.

\section{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.

\subsection{straight.lido}

A type-\texttt{lido} file associates computations with nodes of the abstract syntax tree. This file differs from that in the previous specification only by the deletion of the two \texttt{TERM} declarations. If a non-literal terminal symbol is not declared in \texttt{LIDO}, it is assumed to have an integer value. This assumption reflects the interface constraint discussed in Section 2.1.2.

\begin{verbatim}
straight.lido[10]:

\end{verbatim}

\textbf{Previous abstract syntax tree definition[1]}

\texttt{CHAIN MemoryDep: VOID;}

\texttt{RULE: Program ::= Stm}

\texttt{COMPUTE}

\texttt{CHAINSTART Stm.Memor}

\texttt{END;}

\texttt{RULE: Stm ::= id ':=' Exp}

\texttt{COMPUTE}

\texttt{Stm.MemorDep=Store(id,Exp.Value) <- Exp.MemorDep;}

\texttt{END;}

\texttt{RULE: Exp ::= id}

\texttt{COMPUTE}

\texttt{ }
Exp.Value=Fetch(id) <- Exp.MemoryDep;
Exp.MemoryDep=Exp.Value;
END;

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;

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
END;

RULE: Binop ::= '+'
COMPUTE
  Binop.Value=ADD(Binop.Left,Binop.Right);
END;

RULE: Binop ::= '-'
COMPUTE
  Binop.Value=SUB(Binop.Left,Binop.Right);
END;
This macro is attached to a product file.

### 2.3.2 straight.con

A type-con file defines the phrase structure of the input text. If a complete definition of the abstract tree structure is given, only the information needed to disambiguate that definition need be given in a type-con file. This file did not exist in the previous specification.

**straight.con**[11]:

- Rules to resolve the statement sequence ambiguity[2]
- Rules to resolve the operand ambiguity[4]

This macro is attached to a product file.

### 2.3.3 straight.map

A type-map file defines the relationship between the phrase structure and the tree structure in cases where simple pattern matching between rules does not suffice. This file did not exist in the previous specification.

**straight.map**[12]:

- MAPSYM

  - Map StmList phrases to Stm nodes[3]
  - Map all expressions to Exp and operators to Binop[5]

This macro is attached to a product file.

### 2.3.4 straight.gla

A type-gla file defines the non-literal terminal symbols, both their lexical structure and the way in which their values are determined. This file did not exist in the previous specification.

**straight.gla**[13]:

- Specifications of the non-literal terminal symbols[6]

This macro is attached to a product file.
2.3.5  straight.ptg

A type-ptg file defines patterns for structured output. This file is unchanged from the previous specification.

straight.ptg[14]:

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

This macro is attached to a product file.

2.3.6  straight.pdl

A type-pdl file defines the set of properties that can be associated with entities. This file is unchanged from the previous specification.

straight.pdl[15]:

    Value: int;

This macro is attached to a product file.

2.3.7  straight.c

A type-c file contains handwritten C code that is to be combined with the generated code to form the complete processor. In the previous specification, this file contained a main program in addition to the implementation of the memory module. The main program is now supplied by Eli.

straight.c[16]:

    #include <string.h>
    #include "csm.h"
    #include "envmod.h"
    #include "pdl_gen.h"

    \textbf{Implementation of the memory module}[8]

This macro is attached to a product file.

2.3.8  straight.h

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

straight.h[17]:

    ifndef STRAIGHT_H
    define STRAIGHT_H

    \textbf{Operations exported by the memory abstract data type}[7]

    \textbf{endif}

This macro is attached to a product file.
2.3.9 straight.INIT.phi

A type-INIT.phi file allows one to specify operations that should take place before the processor begins to analyze the input text. This file did not exist in the previous specification.

straight.INIT.phi[18]:

Initialize the memory module[9]

This macro is attached to a product file.

2.3.10 straight.HEAD.phi

A type-HEAD.phi file allows one to specify header files and C-preprocessor macros needed by the generated code. This file is unchanged from the previous specification.

straight.HEAD.phi[19]:

#include "straight.h"

This macro is attached to a product file.

2.3.11 straight.specs

A type-specs file lists Eli library modules needed to support the specification. This file is unchanged from the previous specification.

straight.specs[20]:

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

This macro is attached to a product file.

2.3.12 Odinfile

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

Odinfile[21]:

%tryeli.specs == <<END
   straight.HEAD.phi
   straight.INIT.phi
   straight.c
   straight.con
   straight.gla
   straight.h
   straight.lido
   straight.map
   straight.pdl
   straight.ptg
tryeli == %tryeli.specs :exe

This macro is attached to a non-product file.

2.3.13 README.eli

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

README.eli[22]:

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 non-product file.

3 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 3.1, representation of an abstract syntax tree is easily implemented by applying the “Composite” design pattern. Section 3.2 implements an interpreter by applying the “Visitor” pattern. Initialization and execution of the interpreter are discussed in Section 3.3. Finally, Section 3.4 collects the components and constructs the set of files making up the complete program.

3.1 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[23]:

```cpp
class Node {
public:
    virtual ~Node() {}
    // Empty Accept method[31];
};
```

This macro is invoked in definition[31].

The purpose and implementation of the `Accept` method are discussed in Section 3.2.1.

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**[24](\(\triangleright 1\)):

```cpp
class \(\triangleright 1\) : public Node {
public:
    virtual \(\triangleright 1()\) {}
    // Empty Accept method[31];
};
```

This macro is invoked in definition[25].

Here the parameter is the nonterminal symbol. Symbol classes must also accept the visitors that do the interpretation, as discussed in Section 3.2.1.

**Classes representing nonterminal symbols**[25]:

- `Class definition for nonterminal`[24](`Stm`)`
- `Class definition for nonterminal`[24](`Exp`)`
- `Class definition for nonterminal`[24](`ExpList`)`
- `Class definition for nonterminal`[24](`Binop`)`

This macro is invoked in definition[26].

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**[26](\(\triangleright 1\)):

```cpp
class \(\triangleright 1\) : public Binop {
public:
    \(\triangleright 1()\) {}
    virtual \(\triangleright 1()\) {}
    // Prototype for the Accept method[30];
};
```
This macro is invoked in definition \[28\].

(The **Accept** method is explained in Section \[3.2.1\].)
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**: 

```cpp
class AssignStm : public Stm {
public:
    AssignStm (string arg1, Exp* arg2) { child1 = arg1; child2 = arg2; }
    virtual ~AssignStm () {}
    string Child1() { return child1; }
    Exp* Child2() { return child2; }

private:
    string child1; Exp* child2;
};
```

This macro is invoked in definition \[28\].

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**: 

```cpp
class CompoundStm : public Stm {
public:
    CompoundStm (Stm* arg1, Stm* arg2) { child1 = arg1; child2 = arg2; }
    virtual ~CompoundStm () {}
    Stm* Child1() { return child1; }
    Stm* Child2() { return child2; }

private:
    Stm* child1, *child2;
};
```

**Class definition for rule AssignStm**: 

```cpp
class PrintStm : public Stm {
public:
    PrintStm (ExpList* arg1) { child1 = arg1; }
    virtual ~PrintStm () {}
    ExpList* Child1() { return child1; }

private:
    ExpList* child1;
};
```

This macro is invoked in definition \[28\].
class IdExp : public Exp {
public:
    IdExp (string arg1) { child1 = arg1; }
    virtual ~IdExp () {}
    string Child1() { return child1; }
[Prototype for the Accept method[30];
private:
    string child1;
};

class NumExp : public Exp {
public:
    NumExp (int arg1) { child1 = arg1; }
    virtual ~NumExp () {}
    int Child1() { return child1; }
[Prototype for the Accept method[30];
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[30];
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[30];
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; }

private:
   Exp* child1; ExpList* child2;
};

class LastExpList : public ExpList {
   public:
      LastExpList (Exp* arg1) { child1 = arg1; }
      virtual ~LastExpList () {} 
      Exp* Child1() { return child1; }
   
   private:
      Exp* child1;
   
   Class definition for Binop rule['Plus']
   Class definition for Binop rule['Minus']
   Class definition for Binop rule['Times']
   Class definition for Binop rule['Div']

   
   This macro is invoked in definition[10]

3.2 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 3.2.1 shows how these relationships are established.

Simple expression evaluation is covered in Section 3.2.2 and Section 3.2.3 adds the implementations of
variables and assignment. Print statements are the subject of Section 3.2.4 and the remaining rules are
wrapped up in Section 3.2.5.

3.2.1 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[29]:

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 17.

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[30]:**

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

This macro is invoked in definitions 26, 27, 28, and 31.

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[31]:**

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

This macro is invoked in definitions 23 and 24.

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

**Accept method definitions for rule classes[32]:**

```cpp
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 18.
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*[33]:

```cpp
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*[34]
    Declare the print line stack*[40]
    Declare the interpreter memory*[37]
};
```

This macro is invoked in definition[39]

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.

### 3.2.2 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 so the container to be adapted must be chosen. The container must be a sequence container, and for a number of reasons the vector seems to be the best choice. Since the values of the expressions in the simple serial programming language are all integers, the adapted container should be an integer vector:

*Declare the expression evaluation stack*[34]:

```c++
stack<vector<int> > ExpValues;
```

This macro is invoked in definition [53].

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

*Interpret the NumExp and OpExp rules*[35]:

```c++
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 [50].

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*[36](\circ 2):

```c++
void Interpreter::Visit\circ 1(\circ 1* node)
{
  int right, left;

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

This macro is invoked in definition [50].

Here the first parameter is the name of a rule node (e.g. *Plus*) and the second is the corresponding C++ operator (e.g. +).
3.2.3 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*[37]:

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

This macro is invoked in definition 33.

(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*[38]:

```cpp
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 50.

3.2.4 Print statements

On page 8 of [Modern Compiler Construction in Java][2], Appel states that:

\[
\text{print}(e_1, e_2, \ldots, e_n) \text{ displays the values of all the expressions, evaluated left to right, separated by spaces, terminated by a newline.}
\]

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*[39]:

```cpp
void Interpreter::VisitPrintStm(PrintStm* node)
{   vector<int> line;
    vector<int>::iterator i;
```
Compute all of the values on the print line[41]

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

This macro is invoked in definition[50]

The expressions whose values are to be printed are not directly available at the PrintStmt 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[40]:

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

This macro is invoked in definition[33]

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[41]:

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

This macro is invoked in definition[39]

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

Interpret the PairExpList and LastExpList rules[42]:

```cpp
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[50]

3.2.5 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[43]:

22
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 50.

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

### 3.3 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:

*Use explicit constructors to build a test tree* [44]:

```cpp
Stm* prog =
new CompoundStm(
    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 Minus(), new NumExp(5))
                                    ),
                                    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")))
);```

23
Once the tree has been constructed, it can be interpreted by instantiating an interpreter and invoking the \texttt{Accept} method of the tree root:

\textit{Interpret the test tree[45]:}

\begin{verbatim}
    Interpreter evaluate;
    prog->Accept(&evaluate);
\end{verbatim}

Termination of the \texttt{Accept} invocation indicates termination of the program being interpreted.

\section*{3.4 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.

\subsection*{3.4.1 abstree.h}

File \texttt{abstree.h} provides the interface for the module implementing the abstract syntax tree.

\texttt{abstree.h}[46]:

\begin{verbatim}
    #ifndef ABSTREE_H
    #define ABSTREE_H

    #include <string>

    class Visitor;

    Class representing all tree nodes[23]
    Classes representing nonterminal symbols[25]
    Classes representing rules[28]

    #endif
\end{verbatim}

This macro is attached to a product file.

The symbol \texttt{ABSTREE\_H} is used to protect against successive inclusions of the material from a header file: \texttt{ABSTREE\_H} will be defined at the first inclusion, and that definition will cause later inclusions to be omitted.

\subsection*{3.4.2 visitor.h}

File \texttt{visitor.h} defines the visitor class for the abstract syntax tree.

\texttt{visitor.h}[47]:

24
/*

#include "abstree.h"

Declar the abstract visitor class[29]

*/

This macro is attached to a product file.

### 3.4.3 abstree.cc

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

abstree.cc[48]:

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

Accept method definitions for rule classes[32]
```

This macro is attached to a product file.

### 3.4.4 interpreter.h

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

interpreter.h[49]:

```c
#ifndef INTERPRETER_H
#define INTERPRETER_H

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

Declar the concrete interpreter subclass[33]

#endif
```

This macro is attached to a product file.

### 3.4.5 interpreter.cc

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

interpreter.cc[50]:

```c
#include <iostream.h>
#include "abstree.h"
#include "interpreter.h"
```
Interpret VisitCompoundStm and VisitEseqExp rules
Interpret the AssignStm and IdExp rules
Interpret the PrintStm rule
Interpret the NumExp and OpExp rules
Interpret the PairExpList and LastExpList rules
Interpret any rule whose left-hand-side symbol is Binop

This macro is attached to a product file.

3.4.6 try.cc

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

try.cc:

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

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

This macro is attached to a product file.

3.4.7 Makefile

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

Makefile:

```bash
EXE    = trycpp
SRCS   = try.cc abstree.cc interpreter.cc
HDRS   = abstree.h visitor.h interpreter.h
CXX    = g++
CXXFLAGS =
OBJJS  = $(addsuffix .o,$(basename $(SRCS)))
DEPS   = $(addsuffix .dep,$(basename $(SRCS)))

all:    trycpp
        ./trycpp

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

clean:
        rm -f $(OBJJS)
```

This macro is attached to a product file.
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.

3.4.8 README.cpp

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

README.cpp[53]:

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.