Constructing Tree Abstractions from Text

W. M. Waite

August 31, 1998

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.

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.

In *Tree Abstractions for Programs* 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 := ( print(a, a - 1), 10 * a ); 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 scanner and parser specified by Section 2 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 systematic hand coding and generation. 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". The names of those macros are the names of the files that will be generated from the document.

FunnelWeb was used to process this document.

2 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 2.1 recalls the abstract syntax tree definition and shows how to specify the necessary structural analysis. 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 *Tree Abstractions for Programs* 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 straightline programming language. This LIDO specification was developed in *Tree Abstractions for Programs*:

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 the input text is usually specified by a context-free grammar. Since the phrase structure of the input text mirrors the structure of the abstract syntax tree, and the abstract syntax tree structure is specified by a context-free grammar (the so-called *abstract grammar*), why is the syntactic analysis not already specified?

Unfortunately, the abstract grammar is usually ambiguous. It describes a structure, but it does not fix the representation of that structure as linear text. Here is a simple text written in the straight-line programming language:

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

Using the abstract grammar given at the beginning of Section **??**, the structure of this text could be described either as:

```
Axiom(
  CompoundStm(
    CompoundStm(
      AssignStmt(IdExp(a),NumExp(5)),
      AssignStmt(IdExp(b),NumExp(10))
    ),
    PrintStm(PairExpList(IdExp(a),LastExpList(IdExp(b))))
  )
)
```

or as:

```
Axiom(
  CompoundStm(
    AssignStmt(IdExp(a),NumExp(5)),
    CompoundStm(
      AssignStmt(IdExp(b),NumExp(10))
      PrintStm(PairExpList(IdExp(a),LastExpList(IdExp(b))))
    ),
  )
)
```

This ambiguity can be resolved by providing additional rules to refine the phrase structure described by the abstract grammar. The additional rules form part of what is known as the *concrete grammar*:

Rules to resolve the statement sequence ambiguity[2]:

```
StmList ::= Stm / StmList ';' Stm .

Program ::= StmList .
Exp     ::= '(' StmList ',' Exp ')' .
```

This macro is invoked in definition 12.

The general strategy for resolving sequence ambiguities is to define a new symbol (in this case `StmList`) to represent the sequence. The original definition of the sequence (rule `CompoundStm` above) is replaced by a rule or rules that specify exactly one of the possible structures. All instances of `Stm` appearing on the right-hand sides of existing rules (for example the `Axiom` rule above) are then replaced by `StmList`.

These additional rules are used only to resolve an ambiguity in constructing an abstract syntax tree from the textual representation of a program. `StmList` was introduced solely for that purpose; semantically, it is identical to `Stm`. When the phrases described by the above rules are recognized, the syntactic analyzer should build `CompoundStm`, `Axiom`, and `EseqExp` nodes. That behavior requires the additional information that phrases named `StmList` should be considered to be `Stm` nodes:

Map StmList phrases to Stm nodes[3]:

```
Stm ::= StmList .
```

This macro is invoked in definition 13.

This line, although it looks like a grammar rule, is actually part of a completely different *mapping specification*.

A second ambiguity in the straight-line grammar involves expressions. Issues of operator precedence and association are ignored, so there's no way to tell whether `b` is an operand of `+` or `*` in `a+b*c`. This is a well-understood problem, with a general solution. that is similar to the solution of the sequence ambiguity: Define a set of symbols representing expression with operators at different precedence levels and write rules in terms of those symbols. A complete explanation of the process can be found in most compiler construction texts. Here is the result for Appel's stright-line programming language:

Rules to define operator precedence and association[4]:

```
Exp     ::= Term    / Exp  Addop Term .
Term    ::= Primary / Term Mulop Primary .
Primary ::= id / num .

Addop   ::= '+' / '-' .
Mulop   ::= '*' / '/' .
```

This macro is invoked in definition 12.

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

Map symbols defining operator precedence and association[5]:

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

This macro is invoked in definition 13.

It is important to understand 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 concrete and abstract rules, using the mapping specifications and general pattern-matching techniques. There is also no need to write a complete grammar to define the phrase structure, because Eli can augment the concrete grammar with rules from the abstract grammar where necessary.

2.1.2 Lexical analysis

The terminal symbols of the abstract 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 abstract grammar they must introduced by the concrete grammar developed according to the techniques of Section 2.1.1. Literal terminal symbols (like `+`) are completely defined by the grammar, and therefore need not be specified again. The grammar contains no description of the internal structure of the non-literal symbols, however, so this must be specified separately.

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

Specification of the non-literal symbols[6]:

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

This macro is invoked in definition 14.

The name of the symbol is written at the beginning of the line, followed by a colon. A regular expression, introduced by `$` and terminated by white space, defines the set of character sequences acceptable as instances of the basic symbol. Finally, the name of a *token processor* enclosed by brackets provides the method for establishing the basic symbol's value. (A token processor is simply a routine that obeys a specific 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 an unbounded vector of strings in which to store the textual representation of a basic symbol. By storing the basic symbol's string in this vector, 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 the value of a basic symbol, and token processor implementing them are included in Eli's library. The token processor `mkidn` used above is a library routine that 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. Thus all instances of a particular `id` in a straight-line program will have the same value stored at their tree nodes.

Integer denotations resulting in `num` nodes are given their base-10 numeric values by the token processor `mkint`. This conversion is limited by the integer representation on the machine running the generated processor.

2.2 Interpreting the Program

The only effect that the addition of a structural analyzer has on the interpretation defined in *Tree Abstractions for Programs* involves the value of the non-literal terminal symbol `id`. There this symbol's value was a string, now it is an integer. The requires a change in the interface to the memory abstract data type:

Operations exported by the memory abstract data type[7]:

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

This macro is invoked in definition 18.

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 the specification will *not* provide a main program. The memory abstract data type becomes a closed module, which must be initialized. (*Tree Abstractions for Programs* could, of course, have used a closed module for the memory abstract data type. A more open representation was chosen to minimize the number of files.)

Implementation of the memory module[8]:

```
Environment env;

void
InitMem(void)
```

```
{ env = NewEnv();
}

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

int
Fetch(int sym)
{ return GetValue(DefineIdn(env, sym),0);
}
```

This macro is invoked in definition 17.

This module is simpler than the one specified in *Tree Abstractions for Programs* because the conversion of the identifiers to unique integers has already been carried out during lexical analysis.

The initialization must be invoked before the `Store` and `Fetch` operations can be used:

Initialize the memory module[9]:

```
InitMem();
```

This macro is invoked in definition 19.

2.3 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.

2.3.1 straight.lido

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

straight.lido[10]:

Previous abstract syntax tree definition[1]

```
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;
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
  Exp[1].Value=Exp[2].Value;
END;
```

Implementation of a binary operator[11]('+','ADD')
Implementation of a binary operator[11]('-','SUB')
Implementation of a binary operator[11]('*','MUL')
Implementation of a binary operator[11]('/','DIV')

This macro is attached to a product file.

Implementation of a binary operator[11](◇2):

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

This macro is invoked in definition 10.

2.3.2 straight.con

A type-`con` file defines the phrase structure of the input text. If a complete defintition of the abstract tree structure is given, only the disambiguating information need appear in a type-`con` file. This file did not exist in *Tree Abstractions for Programs*.

straight.con[12]:

Rules to resolve the statement sequence ambiguity[2]
Rules to define operator precedence and association[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 of rules does not suffice. This file did not exist in *Tree Abstractions for Programs*.

straight.map[13]:

`MAPSYM`
Map StmList phrases to Stm nodes[3]
Map symbols defining operator precedence and association[5]

This macro is attached to a product file.

2.3.4 straight.gla

A type-`gla` file defines both the lexical structure of the non-literal terminal symbols and how the values of those symbols are established. This file did not exist in *Tree Abstractions for Programs*.

straight.gla[14]:

Specification of the non-literal 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 *Tree Abstractions for Programs*.

straight.ptg[15]:

```
Value: $ int
Seq:   $ { " " } $
Line:  $ "\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 *Tree Abstractions for Programs*.

straight.pdl[16]:

```
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 *Tree Abstractions for Programs* 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[17]:

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

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. In *Tree Abstractions for Programs* this file defined a character string for use by the memory module, as well as the module interface. Here the memory module uses integer operands only, so no such definition is required.

straight.h[18]:

```
#ifndef DRIVER_H
#define DRIVER_H

Operations exported by the memory abstract data type[7]

#endif
```

This macro is attached to a product file.

2.3.9 straight.init

A type-`init` 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 *Tree Abstractions for Programs*.

straight.init[19]:

Initialize the memory module[9]

This macro is attached to a product file.

2.3.10 straight.head

A type-`head` file allows one to specify header files and C-preprocessor macros needed by the generated code. This file is unchanged from *Tree Abstractions for Programs*.

straight.head[20]:

```
#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 one line shorter than it was in *Tree Abstractions for Programs*, due to the simplification of the memory abstract data type.

straight.specs[21]:

```
$/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. This file is longer than it was in *Tree Abstractions for Programs* because the number of specification files has increased. Also, the main program and parser are not suppressed when the executable is generated and the executable is not executed after being generated.

Odinfile[22]:

```
%tryeli.specs == <<END
  straight.c
  straight.con
  straight.gla
  straight.h
  straight.head
  straight.init
  straight.lido
  straight.map
  straight.pdl
```

```
  straight.ptg
  straight.specs
END

tryeli == %tryeli.specs :exe
```

This macro is attached to a non-product file.

2.3.13 eliinput

File `eliinput` provides test data. This file did not exist in *Tree Abstractions for Programs*.

eliinput[23]:

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

This macro is attached to a non-product file.

2.3.14 README.eli

A README file gives instructions about how to use the contents of a directory to reach some particular goal. This file is longer than the version in *Tree Abstractions for Programs* because the executable file must now be applied to input text to produce output.

README.eli[24]:

```
This directory contains a set of specification files for use with Eli
(http://www.cs.colorado.edu/~eliuser/).  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" that will interpret a
program submitted either as the standard input or as an explicit file:

        tryeli < eliinput
        tryeli eliinput

Either of these commands should yield the following three lines of output:

2
8 7
80
```

This macro is attached to a non-product file.

3 A C++ Solution

Structural analyzers have been implemented directly in programming languages for decades, and most textbooks on compiler construction provide viable plans for this activity. The general strategy is to proceed bottom up, first implementing modules for source text input and error reporting (Section 3.1). A lexical analyzer can be built on top of these modules as shown in Section 3.2, and verified independently. Section 3.3 completes the structural analyzer, describing a parser that calls appropriate constructor functions when it reduces phrases. The main program of Section 3.4 invokes the structural analyzer to build the tree and then applies the interpreter.

Section 3.5 gathers all of the necessary files, including those from the solution to the previous problem. Thus this description is self-contained.

3.1 Support Modules

A compiler interacts with its environment in three ways:

- It reads the text of the source program.
- It writes reports about the source program and the compilation process.
- It writes the text of the target program.

The program being implemented here writes the results of executing the program, rather than the text of a target program. That task is carried out by the interpreter visitor, which is unchanged from the previous solution. Files developed there appear in their entirety in Section 3.5.

Section 3.1.1 describes a module for source text input, while Section 3.1.2 implements an error-reporting module. These module interfaces are independent of the particular compiler, and the modules can be implemented in essentially the same manner in a number of different languages.

3.1.1 The source module

The subproblem that must be solved by the source module is to make the sequence of characters that constitutes the source program available to the scanner subtask of lexical analysis. This character sequence will be examined by the scanner, which will classify contiguous subsequences as basic symbols. The scanner is simplified if its input is stored in contiguous locations. Unfortunately, it is usually impractical to allocate an array large enough to hold an entire program.

A reasonable compromise is to use a string holding an integral number of lines as the interface data structure for the source module. C++ strings are unbounded, so this choice does not limit the length of a line.

Source module interface[25]:

```
extern string Line;     /* An integral number of lines,
                         * each terminated by a newline character.
                         ***/

extern int SourceLine();
/* Obtain additional lines of source text
 *   If there is no more source text then on exit-
 *     SourceLine=0
 *     Line is an empty string
 *   Else on exit-
```

```
*     SourceLine=1
*     Line contains the first unexamined line of source text
***/
```

This macro is invoked in definition 55.

The interface specification allows several different implementations of `SourceLine`: Each invocation might result in a single line being physically read from the source file into `Line`. Alternatively, the first invocation might result in many lines being read. Only when the information was exhausted would another physical read be necessary.

In order to keep the example code simple, the implementation given here is quite inefficient. Because the interface does not tightly constrain the implementation, however, the implementation can be changed without affecting anything else in the compiler. Thus this code can be replaced any time it proves to be a bottleneck:

Implementation of the source module[26]:

```
string Line;

int SourceLine()
{ Line = "";
  for (;;) {
    char c = cin.get();
    if (cin.eof()) return Line.length() != 0;
    Line += c;
    if (c == '\n') return 1;
  }
}
```

This macro is invoked in definition 56.

3.1.2 The error module

Many different situations could be reported by a compiler, ranging from a comment on programming style to an error made by the compiler itself. Error reporting is simplified by providing a single module that accepts *any* report and is responsible for delivering that report to the user. The information making up a report must be quite uniform across the spectrum of situations that might be encountered, or the module interface will be too broad and the operations too varied for reasonable implementation.

Violations of the rules of the source language or of the constraints imposed by the compiler may be detected during a particular compilation. Whenever such a violation is detected, the compiler must output a report of the violation and the location at which it occurred. These reports might be examined by the user directly, or a separate process might be used to merge them with a listing of the program or to focus an interactive text editor upon each report in turn. A coordinate system must therefore be provided to permit association of reports with source text positions. The simplest coordinate system makes use of the two-dimensional structure of the source text, specifying a line number and a column number.

Reports can be classified according to the severity of the consequences for the compilation of the situation they report:

- *Note*: Reports of situations that have no consequences for compilation or execution. Examples are the number of lines in the source text or the fact that a particular variable was never used.
- *Warning*: Reports of anomalies that may cause the program's results to be erroneous. The user should be asked for confirmation before the target program is allowed to execute. Examples are detection of a constant `0` as the right operand of division or the fact that a particular variable was used but not set.

- *Error*: Reports of errors that preclude a correct compilation. The user must change the source program and recompile it. Examples are programs with undeclared variables or unbalanced parentheses.
- *Deadly*: Reports of errors that make further processing impossible. Examples are programs that exhaust some compiler data structure or trigger latent bugs in the compiler itself.

Classification by severity is important because it can be used to control the gross behavior of the compiler in response to reports, without requiring this behavior to be keyed to individual situations. For example, the error module can terminate execution immediately upon receipt of a deadly error report, thus preventing the compiler from crashing and possibly losing *all* of the error reports in the process. Abnormal termination is often a tricky thing to get right. By centralizing it in the error module, the compiler designer avoids the problem of ensuring the correctness of many instances.

Reports of compiler errors might not be associated with any particular source text position, but rather with some position in the code of the compiler itself. This information is implicit in the report if it is issued only by one code sequence, but certain errors may be detected at a number of points in the compiler's code. Thus it is useful to be able to associate an *issuer* with each error report. Either a source text position or an issuer, or both, could be relevant for each report.

Interface for the error module[27]:

```
typedef struct {        /* Source text coordinates */
  int line;               /* Index of the source line */
  int col;                /* Index of the character position */
} POSITION;

typedef enum {          /* Classification of consequences */
  NOTE,                   /* No consequences */
  WARNING,                /* Target program may be erroneous */
  ERROR,                  /* Compilation cannot continue */
  DEADLY
} Severity;

typedef POSITION *CoordPtr;

void message(Severity severity, char* Msgtext, int issuer, CoordPtr source);
/* Report an error
 *   On entry-
 *     severity=error severity
 *     Msgtext=message text
 *     issuer=identification of the test that failed
 *     source=source coordinates at which the error was detected
 ***/
```

This macro is invoked in definition 57.

The coordinate system is effectively an abstract data type. In common with many abstract data types, it needs a "bottom" element that represents "no coordinates". Since there are two representations of coordinates, the actual structure and a pointer, "bottom" elements for both are required:

Interface specification for undefined coordinates[28]:

```
#define NoPosition ((CoordPtr)0)        /* Dummy position argument */
extern POSITION NoCoord;                /* The NULL coordinate */
```

This macro is invoked in definition 57.

An additional implementation is only necessary for the structure:

Implementation of the undefined coordinate structure[29]:

```
POSITION NoCoord = { 0, 0 };    /* The NULL coordinate */
```

This macro is invoked in definition 58.

Summary counts of the errors detected at different severity levels are useful. For example there is no point in translating a program if the analysis task has detected errors. It's also usually foolish to continue processing a program once the error *density* (number of errors per line) exceeds some threshhold. (A high error density often indicates either a systematic error or an input file that really isn't written in the source language.)

Variables supporting summary counts[30]:

```
extern int ErrorCount[];        /* Number of errors detected so far */
extern int LineNum;             /* Total number of lines seen so far */
```

This macro is invoked in definition 57.

`LineNum` is actually managed by the lexical analyzer (Section 3.2, and is used as the current input line number. Storage for both variables is provided by the error module:

Storage for summary count variables[31]:

```
int ErrorCount[] = { 0, 0, 0, 0 };
int LineNum = 0;
```

This macro is invoked in definition 58.

It is useful to be able to change two aspects of the error module's behavior: inclusion of issuer information in the error reports and termination when the error density exceeds the threshhold. Issuer information is useful in debugging the compiler, but usually annoying to normal users. Normal users are generally quite happy with the density limit, but sometimes it must be turned off to demonstrate some particular behavior of the compiler. A "customization" routine can be used to control these aspects:

Customization control interface[32]:

```
extern void ErrorControl(int Issuer, int ErrLimit);
/* Initialize the error module
 *   On entry-
 *     Issuer=1 to print issuer on error reports
 *     ErrLimit=1 to limit the number of errors reported
 ***/
```

This macro is invoked in definition 57.

The implementation uses two state variables:

Customization control implementation[33]:

```
static int IssuerOK = 0;        /* 1 to print issuer */
static int ErrorLimit = 1;      /* 1 to abort on high error density */

void ErrorControl(int Issuer, int ErrLimit)
{ IssuerOK = Issuer; ErrorLimit = ErrLimit; }
```

This macro is invoked in definition 58.

The error routine is straightforward:

Error report handler[34]:

```
void message(Severity severity, char* Msgtext, int issuer, CoordPtr source)
{ static char* key[] = {"NOTE", "WARNING", "ERROR", "DEADLY"};

  if (source == NoPosition) source = &NoCoord;

  cerr << "At (" << source->line << ',' << source->col
       << ") " << key[severity] << ": " << Msgtext;
  if (issuer>0 && IssuerOK) cerr << " Issuer=" << issuer;
  cerr << "\n";

  ErrorCount[severity]++;

  if (ErrorLimit && ErrorCount[ERROR] > LineNum/20 + 10) {
    cerr << "At (" << source->line << ',' << source->col
         << ") DEADLY: Too many errors\n";
    severity = DEADLY;
  }

  if (severity == DEADLY) exit(1);
}
```

This macro is invoked in definition 58.

3.2 Lexical Analysis

Basic symbols are the atoms of the source program, the smallest information-bearing units of its structure. A basic symbol may be represented in the text of the source program by a single character or by a character sequence. Whether a particular character is a complete basic symbol, a part of a basic symbol, or unconnected with any basic symbol depends upon its immediate context.

The lexical analyzer accepts the text of the source program (a sequence of characters) as input and produces a sequence of basic symbols as output. It must recognize each basic symbol, extract the information borne by that basic symbol from the source text, and express it in a form appropriate for processing by the rest of the compiler. It must also skip over source text (comments and white space) that is unconnected with any basic symbol and report any invalid characters or ill-formed basic symbols.

Section 3.2.1 describes the interface for lexical analysis, which remains constant over all source languages. The basic symbol definitions are used to parameterize a standard algorithm in Section 3.2.2, and Section 3.2.3 embeds that algorithm in a general-purpose skeleton to complete the lexical analyzer implementation.

3.2.1 The lexical analysis interface

The information carried by each basic symbol is represented internally by a public data type of the lexical analysis module called a `Token`:

Internal representation of a basic symbol[35]:

```
typedef enum {
  Define the set of basic symbols[37]
} SymbolClass;

typedef struct {
  POSITION coord;
  SymbolClass kind;
  int val;
  string idn;
} Token;
```

This macro is invoked in definition 59.

The elements of the `SymbolClass` enumeration are specific to the source language, although many source languages tend to have similar elements. Value information carried by `Token` (fields `val` and `idn` above) could vary from one source language to another, or it could be fixed for all basic symbols and therefore be language-independent. If a fixed type is desired, `int` and `string` seem to be the appropriate candidates.

Basic symbols are absorbed as they are recognized, so lexical analysis can be embodied in a routine that delivers the next basic symbol of the source text each time it is invoked:

Lexical analysis operation definition[36]:

```
extern void Lexical(Token*);
/* Obtain one basic symbol
 *   On exit-
 *     Token describes the first unexamined basic symbol
 ***/
```

This macro is invoked in definition 59.

Remember that "end-of-program" is a valid basic symbol. When the end of the source text is reached, all further invocations of `Lexical` will yield that token, at the coordinates of the first character position beyond the text.

3.2.2 Implementing a finite-state machine

Lexical analysis has two major subtasks:

- *Scanning*: Recognizing a sequence of characters in the source text as an instance of a particular kind of basic symbol.
- *Conversion*: Obtaining the value of the specific basic symbol that has been recognized.

Scanning is handled by a finite-state machine whose inputs are single characters. When the machine enters a final state corresponding to a basic symbol, code associated with that state sets the appropriate symbol class and carries out any necessary conversion.

The set of basic symbols is defined by the `SymbolClass` enumeration and associated comments:

Define the set of basic symbols[37]:

```
EOPT,           /* Endmarker */
IntegerT,       /* num       */
IdentifierT,    /* id        */
PrintT,         /* print     */
ColonEqT,       /* :=        */
LeftParenT,     /* (         */
PlusT,          /* +         */
MinusT,         /* -         */
AsteriskT,      /* *         */
SlashT,         /* /         */
RightParenT,    /* )         */
SemicolonT,     /* ;         */
CommaT          /* ,         */
```

This macro is invoked in definition 35.

The endmarker is the ASCII NUL, `num` and `id` are defined by the regular expressions given in Section 2.1.2, and all other basic symbols consist of the literal characters shown in the comments.

Here is the state table of a finite state machine implementing the scanner. It recognizes `print` as an `id`, leaving the conversion code to distinguish the two:

	letter	digit	:	=	(	+	−	*	/	)	;	,	NUL
1	2	3	4	0	5	6	7	8	9	10	11	12	0,ep
2	2	0,id	0,id	0,id	0,id	0,id	0,id	0,id	0,id	0,id	0,id	0,id	0,id
3	3	0,nu	0,nu	0,nu	0,nu	0,nu	0,nu	0,nu	0,nu	0,nu	0,nu	0,nu	0,nu
4	0	0	0	13	0	0	0	0	0	0	0	0	0
5	0,lp	0,lp	0,lp	0,lp	0,lp	0,lp	0,lp	0,lp	0,lp	0,lp	0,lp	0,lp	0,lp
6	0,pl	0,pl	0,pl	0,pl	0,pl	0,pl	0,pl	0,pl	0,pl	0,pl	0,pl	0,pl	0,pl
7	0,mi	0,mi	0,mi	0,mi	0,mi	0,mi	0,mi	0,mi	0,mi	0,mi	0,mi	0,mi	0,mi
8	0,as	0,as	0,as	0,as	0,as	0,as	0,as	0,as	0,as	0,as	0,as	0,as	0,as
9	0,sl	0,sl	0,sl	0,sl	0,sl	0,sl	0,sl	0,sl	0,sl	0,sl	0,sl	0,sl	0,sl
10	0,rp	0,rp	0,rp	0,rp	0,rp	0,rp	0,rp	0,rp	0,rp	0,rp	0,rp	0,rp	0,rp
11	0,sc	0,sc	0,sc	0,sc	0,sc	0,sc	0,sc	0,sc	0,sc	0,sc	0,sc	0,sc	0,sc
12	0,cm	0,cm	0,cm	0,cm	0,cm	0,cm	0,cm	0,cm	0,cm	0,cm	0,cm	0,cm	0,cm
13	0,ce	0,ce	0,ce	0,ce	0,ce	0,ce	0,ce	0,ce	0,ce	0,ce	0,ce	0,ce	0,ce

This machine displays three common characteristics of scanners:

- Many of the successors of the initial state have no successors.
- Only the initial state has many successors.
- Several of the states have themselves as successors.

These three properties can be exploited to simplify the implementation.

State 1 is implemented by a table indexed by the first character of the basic symbol:

Implementation of state 1[38]:

```
static int state1[] = {
 13,  0,  0,  0,  0,  0,  0,  0,        /* NUL SOH STX ETX EOT ENQ ACK BEL */
  0,  0, 15,  0,  0,  0,  0,  0,        /* BS  HT  LF  VT  FF  CR  SO  SI  */
```

```
  0,  0,  0,  0,  0,  0,  0,  0,        /* DLE DC1 DC2 DC3 DC4 NAK SYN ETB */
  0,  0,  0,  0,  0,  0,  0,  0,        /* CAN EM  SUB ESC FS  GS  RS  US  */
  0,  0,  0,  0,  0,  0,  0,  0,        /* SP  !   "   #   $   %   &   ’   */
  5, 10,  8,  6, 12,  7,  0,  9,        /* (   )   *   +   ,   -   .   /   */
  3,  3,  3,  3,  3,  3,  3,  3,        /* 0   1   2   3   4   5   6   7   */
  3,  3,  4, 11,  0,  0,  0,  0,        /* 8   9   :   ;   <   =   >   ?   */
  0,  2,  2,  2,  2,  2,  2,  2,        /* @   A   B   C   D   E   F   G   */
  2,  2,  2,  2,  2,  2,  2,  2,        /* H   I   J   K   L   M   N   O   */
  2,  2,  2,  2,  2,  2,  2,  2,        /* P   Q   R   S   T   U   V   W   */
  2,  2,  2,  0,  0,  0,  0,  0,        /* X   Y   Z   [   \   ]   ^   _   */
  0,  2,  2,  2,  2,  2,  2,  2,        /* ‘   a   b   c   d   e   f   g   */
  2,  2,  2,  2,  2,  2,  2,  2,        /* h   i   j   k   l   m   n   o   */
  2,  2,  2,  2,  2,  2,  2,  2,        /* p   q   r   s   t   u   v   w   */
  2,  2,  2,  0,  0,  0,  0,  0         /* x   y   z   {   |   }   ~   DEL */
};
```

This macro is invoked in definition 60.

Note that a state (15) has been added to recognize newline characters. Although newlines do not belong to basic symbols, they require that the coordinates be adjusted.

States other than state 1 are handled by code in a switch:

Implementation of the finite-state machine[39]:

```
/* On entry-
 *   c=input character
 *   start=index of the input character in string Line
 *   Current=start + 1
 *   t points to the token data structure
 * On exit-
 *   Current=index of the first unexamined character in string Line
 ***/


switch (state = state1[c]) {

case 0:
  message(ERROR, "Illegal symbol", 0, &(t->coord));
  break;

case 2:
  while (isalpha(Line[Current])) Current++;
  t->idn = Line.substr(start, Current-start);
  if (t->idn == "print") t->kind = PrintT; else t->kind = IdentifierT;
  SymFound = 1;
  break;

case 3:
  while (isdigit(Line[Current])) Current++;
  t->val = 0;
  while (start < Current) t->val = t->val * 10 + Line[start++] - ’0’;
```

```
  t->kind = IntegerT;
  SymFound = 1;
  break;

case 4:
  if (Line[Current] == '=') {   /* State 14 */
    Current++;
    t->kind = ColonEqT;
    SymFound = 1;
  } else message(ERROR, "Illegal symbol", 0, &(t->coord));
  break;

case 13:
  StartLine = 0; Current = 0;
  if (!SourceLine()) { t->kind = EOPT; SymFound = 1; }
  break;

case 15:
  LineNum++; StartLine = start;
  break;
default:
               /*
Single-character basic symbols */
  t->kind = Kind[state - 5];
  SymFound = 1;
}
```

This macro is invoked in definition 41.

Transitions out of state 1 on single-character basic symbols lead to a compact group of states numbered 5 through 12. `Kind` provides the `SymbolClass` value for each of these states:

State-to-SymbolClass table for single-character basic symbols[40]:

```
static SymbolClass Kind[] = {
  LeftParenT,
  PlusT,
  MinusT,
  AsteriskT,
  SlashT,
  RightParenT,
  SemicolonT,
  CommaT
};
```

This macro is invoked in definition 60.

3.2.3 Controlling the finite state machine

The code derived from the finite-state machine in Section 3.2.2 recognizes (and converts if necessary) a single basic symbol starting at the location in the source text indicated by `Current`. It advances `Current` to

indicate the first unexamined source text character. If no basic symbol can be found at the specified starting location, the code reports an error and advances `Current` by one character position.

The lexical analyzer is completed by embedding the code derived from the finite-state machine in a loop that applies it repeatedly until a basic symbol is found:

Control the application of the finite-state machine[41]:

```
SymFound = 0;

do {
  char c; int start, state;

  do { c = Line[Current++]; } while (c == ' ');
  t->coord.line = LineNum; t->coord.col = Current - StartLine;
  start = Current - 1;
  Implementation of the finite-state machine[39]
} while (!SymFound);
```

This macro is invoked in definition 60.

The initial loop skips spaces efficiently without introducing extra overhead in the case of single spaces. This is an important consideration, because measurements show that in most programming languages single spaces are the most common textual element.

Three state variables (`LineNum`, `StartLine` and `Current`) control the source text access and coordinates. `LineNum` is exported by the error module (Section 3.1.2, but its value is updated by the finite-state machine. The other state variables belong to the lexical analysis module:

Lexical analyzer state variables[42]:

```
static int StartLine = 0;
static int Current = 0;
```

This macro is invoked in definition 60.

`StartLine` indexes the first character of the current line in `Line`, the string exported by the source module (Section 3.1.1). `Current` is the index of the first character that has not yet been examined by the lexical analyzer.

3.3 Syntactic Analysis

A syntactic analyzer builds the source program tree corresponding to the sequence of basic symbols delivered by the lexical analysis module. This task can be divided into two parts: parsing and tree construction. The parser invokes the lexical analyzer to obtain basic symbols and invokes the tree constructor to build nodes. Thus the parser acts as the central control algorithm of the syntactic analysis module.

The behavior of the parser is specified by a parsing grammar that gives an unambiguous definition of the phrase structure of the source language and relates those phrases to the nodes of the abstract syntax tree. Here is parsing grammar derived from a combination of the rules appearing in Sections 2.1 and 3.3. The name of an abstract syntax tree node is given in double quotes (`"`) in each rule defining a corresponding phrase:

```
Program:
  StmList .

StmList:
  Stm /
  StmList ';' Stm "CompoundStm" .

Stm:
  id ':=' Exp "AssignStm" /
  'print' '(' ExpList ')' "PrintStm" .

Exp:
  '(' StmList ',' Exp ')' "EseqExp" /
  Term /
  Exp Addop Term "OpExp" .

Addop:
  '+' "Plus" /
  '-' "Minus" .

Term:
  Primary /
  Term Mulop Primary "OpExp" .

Mulop:
  '*' "Times" /
  '/' "Div" .

Primary:
  id "IdExp" /
  num "NumExp" .

ExpList:
  Exp ',' ExpList "PairExpList" /
  Exp "LastExpList" .
```

Although this grammar is unambiguous, it contains left recursion. That left recursion must be removed by transforming the grammar before a recursive descent parser can be implemented. In this case, the left recursion can be transformed to iteration:

```
Program:
  StmList .

StmList:
  Stm (';' Stm "CompoundStm")* .

Stm:
  id ':=' Exp "AssignStm" /
```

```
  'print' '(' ExpList ')' "PrintStm" .

Exp:
  '(' StmList ',' Exp ')' "EseqExp" /
  Term (Addop Term "OpExp")* .

Addop:
  '+' "Plus" /
  '-' "Minus" .

Term:
  Primary (Mulop Primary "OpExp")* .

Mulop:
  '*' "Times" /
  '/' "Div" .

Primary:
  id "IdExp" /
  num "NumExp" .

ExpList:
  Exp ',' ExpList "PairExpList" /
  Exp "LastExpList" .
```

You should convince yourself that these two grammars define the same set of strings, and specify the same trees.

A recursive descent parser is implemented by writing one procedure for each nonterminal symbol of the parsing grammar. The procedure returns a pointer to an object of the abstract class corresponding to that nonterminal symbol.

Each element of the right-hand side of a grammar rule corresponds to a code fragment within the procedure for the left-hand-side symbol:

- Code corresponding to a terminal absorbs the appropriate basic symbol.
- Code corresponding to a tree node name constructs that tree node.
- Code corresponding to a nonterminal invokes the nonterminal's procedure.
- Code for elements of a right-hand-side sequence is sequential, while code for alternatives is preceded by tests to determine which of the paths should be taken.
- An iteration in the code implements an iteration in the grammar.

The parser has a state variable that contains the first unaccepted basic symbol on entry to and exit from every routine corresponding to a nonterminal symbol:

Parser state variable[43]:

```
Token t;
```

This macro is invoked in definition 62.

Here is the procedure implementing `StmList`:

Recursive procedures implementing the grammar[44]:

```
Stm* ParseStmList()
{ Stm* result = ParseStm();
  while (t.kind == SemicolonT) {
    Lexical(&t);
    result = new CompoundStm(result, ParseStm());
  }
  return result;
}
```

This macro is defined in definitions 44, 45, 46, and 47.
This macro is invoked in definition 62.

`StmList` was introduced to avoid an ambiguity in the definition of `Stm` (see Section 3.3). Thus there is no class `StmList`; any `StmList` phrase corresponds to a `Stm` tree node, so `ParseStmList` returns a `Stm*` value.

The right-hand side of the `StmList` rule begins with the nonterminal symbol `Stm`. That symbol corresponds to a call of `ParseStm`, which returns a `Stm*` value. If the next basic symbol is a semicolon, it is accepted and one iteration takes place. After obtaining the next basic symbol, `ParseStmList` invokes `ParseStm` and builds a `CompoundStm` object with the result as the right child and the preceding `Stm*` value as the left child. A pointer to this created object becomes the tentative result of `ParseStmList`.

When the first unaccepted basic symbol is *not* a semicolon, the iteration terminates and the result is returned.

`ParseStm` illustrates the coding of alternatives within a rule and shows how errors are reported when the next basic symbol is not acceptable:

Recursive procedures implementing the grammar[45]:

```
Stm* ParseStm()
{ if (t.kind == IdentifierT) {
    Token id = t;
    Lexical(&t);
    if (t.kind != ColonEqT) message(DEADLY, ":= expected", 0, &(t.coord));
    Lexical(&t);
    return new AssignStm(id.idn, ParseExp());
  } else if (t.kind == PrintT) {
    Lexical(&t);
    if (t.kind != LeftParenT) message(DEADLY, "( expected", 0, &(t.coord));
    Lexical(&t);
    ExpList* child1 = ParseExpList();
    if (t.kind != RightParenT) message(DEADLY, ") expected", 0, &(t.coord));
    Lexical(&t);
    return new PrintStm(child1);
  }
  message(DEADLY, "Identifier or print keyword expected", 0, &(t.coord));
  return (Stm*)0;
}
```

This macro is defined in definitions 44, 45, 46, and 47.
This macro is invoked in definition 62.

Here the routine checks the first unaccepted basic symbol to determine which of the two alternative definitions of `Stm` to use. If that basic symbol is not acceptable, an error is reported at the symbol's coordinates and the compiler terminates. (Normally the compiler would attempt to recover from this error and continue, but that would make the example more complex.)

Most of the remaining routines do not introduce new concepts:

Recursive procedures implementing the grammar[46]:

```
Exp* ParseExp()
{ if (t.kind == IdentifierT || t.kind == IntegerT) {
    Exp* child1 = ParseTerm();
    while (t.kind == PlusT || t.kind == MinusT) {
      Binop* child2 = ParseAddop();
      child1 = new OpExp(child1, child2, ParseTerm());
    }
    return child1;
  } else if (t.kind == LeftParenT) {
    Lexical(&t);
    Stm* child1 = ParseStm();
    if (t.kind != CommaT) message(DEADLY, ", expected", 0, &(t.coord));
    Lexical(&t);
    Exp* child2 = ParseExp();
    if (t.kind != RightParenT) message(DEADLY, ") expected", 0, &(t.coord));
    Lexical(&t);
    return new EseqExp(child1, child2);
  }
  message(DEADLY, "Expression or ( expected", 0, &(t.coord));
  return (Exp*)0;
}


ExpList* ParseExpList()
{ Exp* child1 = ParseExp();
  if (t.kind == CommaT) {
    Lexical(&t);
    return new PairExpList(child1, ParseExpList());
  } else
    return new LastExpList(child1);
}


Exp* ParseTerm()
{ Exp* child1 = ParsePrimary();
  while (t.kind == AsteriskT || t.kind == SlashT) {
    Binop* child2 = ParseMulop();
    child1 = new OpExp(child1, child2, ParsePrimary());
  }
  return child1;
}
```

```
Exp* ParsePrimary()
{ if (t.kind == IdentifierT) {
    Exp* result = new IdExp(t.idn);
    Lexical(&t);
    return result;
  } else if (t.kind == IntegerT) {
    Exp* result = new NumExp(t.val);
    Lexical(&t);
    return result;
  }
  message(DEADLY, "Identifier or integer expected", 0, &(t.coord));
  return (Exp*)0;
}


Binop* ParseAddop()
{ if (t.kind == PlusT) {
    Lexical(&t);
    return new Plus();
  } else if (t.kind == MinusT) {
    Lexical(&t);
    return new Minus();
  }
  message(DEADLY, "+ or - expected", 0, &(t.coord));
  return (Binop*)0;
}


Binop* ParseMulop()
{ if (t.kind == AsteriskT) {
    Lexical(&t);
    return new Times();
  } else if (t.kind == SlashT) {
    Lexical(&t);
    return new Div();
  }
  message(DEADLY, "* or / expected", 0, &(t.coord));
  return (Binop*)0;
}
```

This macro is defined in definitions 44, 45, 46, and 47.
This macro is invoked in definition 62.

The top-level routine must establish the invariant on `t` by invoking the lexical analyzer before attempting to parse the program. It must also verify on completion that the first unaccepted token is `EOPT`, showing that the entire program has been parsed:

Recursive procedures implementing the grammar[47]:

```
Stm* ParseProgram()
{ Lexical(&t);
```

```
  Stm* result = ParseStmList();
  if (t.kind != EOPT) message(DEADLY, "End-of-file expected", 0, &(t.coord));
  return result;
}
```

This macro is defined in definitions 44, 45, 46, and 47.
This macro is invoked in definition 62.

`ParseProgram` is invoked to obtain the complete tree:

Parse the input text and construct the tree[48]:

```
Stm* program = ParseProgram();
```

This macro is invoked in definition 63.

3.4 Initialization and Execution

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[49]:

```
Interpreter evaluate;

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

This macro is invoked in definition 63.

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

3.5 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.

3.5.1 abstree.h

File `abstree.h` defines the abstract syntax tree classes. This file is unchanged from *Tree Abstractions for Programs*.

abstree.h[50]:

```
#ifndef ABSTREE_H
#define ABSTREE_H

using namespace std;

#include <string>
```

```
class Visitor;

class Node {
  public:
    virtual ~Node() {}
    virtual void Accept (Visitor*) = 0;
};

class Stm : public Node {
  public:
    virtual ~Stm() {}
    virtual void Accept (Visitor*) = 0;
};

class Exp : public Node {
  public:
    virtual ~Exp() {}
    virtual void Accept (Visitor*) = 0;
};

class ExpList : public Node {
  public:
    virtual ~ExpList() {}
    virtual void Accept (Visitor*) = 0;
};

class Binop : public Node {
  public:
    virtual ~Binop() {}
    virtual void Accept (Visitor*) = 0;
};

class CompoundStm : public Stm {
  public:
    CompoundStm (Stm* arg1, Stm* arg2) { child1 = arg1; child2 = arg2; }
    virtual ~CompoundStm () {}
    Stm* Child1() { return child1; }
    Stm* Child2() { return child2; }
    virtual void Accept (Visitor*);
  private:
    Stm* child1, *child2;
};

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

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

class PrintStm : public Stm {
  public:
    PrintStm (ExpList* arg1) { child1 = arg1; }
    virtual ~PrintStm () {}
    ExpList* Child1() { return child1; }
    virtual void Accept (Visitor*);
  private:
    ExpList* child1;
};

class IdExp : public Exp {
  public:
    IdExp (string arg1) { child1 = arg1; }
    virtual ~IdExp () {}
    string Child1() { return child1; }
    virtual void Accept (Visitor*);
  private:
    string child1;
};

class NumExp : public Exp {
  public:
    NumExp (int arg1) { child1 = arg1; }
    virtual ~NumExp () {}
    int Child1() { return child1; }
    virtual void Accept (Visitor*);
  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; }
    virtual void Accept (Visitor*);
  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; }
    virtual void Accept (Visitor*);
  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; }
    virtual void Accept (Visitor*);
  private:
    Exp* child1; ExpList* child2;
};

class LastExpList : public ExpList {
  public:
    LastExpList (Exp* arg1) { child1 = arg1; }
    virtual ~LastExpList () {}
    Exp* Child1() { return child1; }
    virtual void Accept (Visitor*);
  private:
    Exp* child1;
};

class Plus : public Binop {
  public:
    Plus () {}
    virtual ~Plus () {}
    virtual void Accept (Visitor*);
};

class Minus : public Binop {
  public:
    Minus () {}
    virtual ~Minus () {}
    virtual void Accept (Visitor*);
};

class Times : public Binop {
  public:
    Times () {}
    virtual ~Times () {}
    virtual void Accept (Visitor*);
};

class Div : public Binop {
  public:
    Div () {}
```

```
    virtual ~Div () {}
    virtual void Accept (Visitor*);
};

#endif
```

This macro is attached to a product file.

3.5.2 visitor.h

File `visitor.h` defines the visitor class for the abstract syntax tree. This file is unchanged from *Tree Abstractions for Programs*.

visitor.h[51]:

```
#ifndef VISITOR_H
#define VISITOR_H

using namespace std;

#include "abstree.h"

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;
};

#endif
```

This macro is attached to a product file.

3.5.3 abstree.cc

File `abstree.cc` defines the `Accept` operations that take a visitor as an argument. This file is unchanged from *Tree Abstractions for Programs*.

abstree.cc[52]:

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

```
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 attached to a product file.

3.5.4 interpreter.h

File `interpreter.h` is the definition of the interpreter class. This file is unchanged from *Tree Abstractions for Programs*.

interpreter.h[53]:

```
#ifndef INTERPRETER_H
#define INTERPRETER_H

using namespace std;

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

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

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

#endif
```

This macro is attached to a product file.

3.5.5 interpreter.cc

File `interpreter.cc` implements the methods of the interpreter class. This file is unchanged from *Tree Abstractions for Programs*.

interpreter.cc[54]:

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

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);
}

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()]);
}

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

  PrintLine.push(&line); (node->Child1())->Accept(this); PrintLine.pop();
  for (i = line.begin(); i != line.end(); i++) cout << *i << ' ';
  cout << '\n';
}

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);
}

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();
}

void Interpreter::VisitPlus(Plus* node)
{ int right, left;

  right = ExpValues.top(); ExpValues.pop();
  left = ExpValues.top(); ExpValues.pop();
  ExpValues.push(left + right);
}

void Interpreter::VisitMinus(Minus* node)
{ int right, left;

  right = ExpValues.top(); ExpValues.pop();
  left = ExpValues.top(); ExpValues.pop();
  ExpValues.push(left - right);
}

void Interpreter::VisitTimes(Times* node)
{ int right, left;

  right = ExpValues.top(); ExpValues.pop();
  left = ExpValues.top(); ExpValues.pop();
  ExpValues.push(left * right);
}

void Interpreter::VisitDiv(Div* node)
{ int right, left;

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

This macro is attached to a product file.

3.5.6 source.h

File `source.h` defines the source module. This file did not exist in *Tree Abstractions for Programs*.

source.h[55]:

```
#ifndef SOURCE_H
#define SOURCE_H

using namespace std;

#include <string>

```
Source module interface[25]

```
#endif
```

This macro is attached to a product file.

3.5.7 source.cc

File `source.cc` implements the source module. This file did not exist in *Tree Abstractions for Programs*.

source.cc[56]:

```
#include <iostream>
#include "source.h"

```
Implementation of the source module[26]

This macro is attached to a product file.

3.5.8 err.h

File `err.h` defines the error reporting module. This file did not exist in *Tree Abstractions for Programs*.

err.h[57]:

```
#ifndef ERR_H
#define ERR_H

using namespace std;

```
Interface for the error module[27]
Interface specification for undefined coordinates[28]
Variables supporting summary counts[30]
Customization control interface[32]

```
#endif
```

This macro is attached to a product file.

3.5.9 err.cc

File `err.cc` implements the error reporting module. This file did not exist in *Tree Abstractions for Programs*.

err.cc[58]:

```
#include <iostream>
#include "err.h"

Storage for summary count variables[31]
Implementation of the undefined coordinate structure[29]
Customization control implementation[33]
Error report handler[34]
```

This macro is attached to a product file.

3.5.10 scan.h

File `scan.h` defines the lexical analysis module. This file did not exist in *Tree Abstractions for Programs*.

scan.h[59]:

```
#ifndef SCAN_H
#define SCAN_H

using namespace std;

#include <string>
#include "err.h"

Internal representation of a basic symbol[35]
Lexical analysis operation definition[36]

#endif
```

This macro is attached to a product file.

3.5.11 scan.cc

File `scan.cc` implements the lexical analysis module. This file did not exist in *Tree Abstractions for Programs*.

scan.cc[60]:

```
#include <cctype>
#include <iostream>
#include "err.h"
#include "source.h"
#include "scan.h"

Lexical analyzer state variables[42]

void Lexical(Token* t)
{ int SymFound;
  Implementation of state 1[38]
  State-to-SymbolClass table for single-character basic symbols[40]
  Control the application of the finite-state machine[41]
}
```

This macro is attached to a product file.

3.5.12 parse.h

File `parse.h` defines the syntactic analysis module. This file did not exist in *Tree Abstractions for Programs*.

parse.h[61]:

```
#ifndef PARSER_H
#define PARSER_H

using namespace std;

#include "abstree.h"

extern Stm* ParseProgram(void);

#endif
```

This macro is attached to a product file.

3.5.13 parse.cc

File `parse.cc` implements the syntactic analysis module. This file did not exist in *Tree Abstractions for Programs*.

parse.cc[62]:

```
#include "err.h"
#include "scan.h"
#include "abstree.h"
#include "parse.h"
```

Parser state variable[43]

```
Stm*     ParseProgram(void);
Stm*     ParseStmList(void);
Stm*     ParseStm(void);
Exp*     ParseExp(void);
Exp*     ParseTerm(void);
Exp*     ParsePrimary(void);
ExpList* ParseExpList(void);
Binop*   ParseAddop(void);
Binop*   ParseMulop(void);
```

Recursive procedures implementing the grammar[44]

This macro is attached to a product file.

3.5.14 main.cc

File `main.cc` builds the tree and invokes the interpreter to evaluate it. This file did not exist in *Tree Abstractions for Programs*.

main.cc[63]:

```
#include "parse.h"
#include "interpreter.h"

int
main()
{
  Parse the input text and construct the tree[48]
  Interpret the test tree[49]
  return 0;
}
```

This macro is attached to a product file.

3.5.15 cppinput

File `cppinput` provides test data. This file did not exist in *Tree Abstractions for Programs*.

cppinput[64]:

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

This macro is attached to a product file.

3.5.16 Makefile

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

Makefile[65]:

```
EXE     = trycpp

SRCS    = source.cc err.cc scan.cc parse.cc \
          abstree.cc interpreter.cc \
          main.cc

HDRS    = source.h err.h scan.h parse.h \
          abstree.h interpreter.h visitor.h

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

all:    $(EXE)
        ./$(EXE) < cppinput

$(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.

3.5.17 README.cpp

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

README.cpp[66]:

```
This directory contains a set of C++ files that implement a complete
interpreter for the simple straight-line programming language defined
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.