2
A Simple Compiler

Do not copy

In this chapter, we provide an overview of how the compilation process can be organized by considering in detail how a compiler can be built for a very small programming language. We begin in Section 2.1 by informally defining this language, which we call ac. In the rest of the chapter, we present the phases of a simple compiler for ac.

2.1 An Informal Definition of the ac Language

Our language is called ac (for *adding calculator*). It is a very simple language, yet it possesses components that are found in most programming languages. We use this language to examine the phases and data structures of a compiler. Following is the informal definition of ac.

- There are two data types: integer and float. An integer type is a decimal integer, as found in most programming languages. A float type allows five fractional digits after the decimal point.
- There are three reserved keywords: `f` (declares a float variable), `i` (declares an integer variable), and `p` (prints the value of a variable).
- The data type of an identifier is explicitly declared in the program. There are only 23 possible identifiers, drawn from the lowercase Roman alphabet. The exceptions are the three **reserved keywords** `f`, `i`, and `p`.
- When expressions are computed, conversion from integer type to float type is accomplished automatically. Conversion in the other direction is not allowed.

1	Prog	$\rightarrow$	Dcls Stmts $
2	Dcls	$\rightarrow$	Dcl Dcls
3		|	λ
4	Dcl	$\rightarrow$	floatdcl id
5		|	intdcl id
6	Stmts	$\rightarrow$	Stmt Stmts
7		|	λ
8	Stmt	$\rightarrow$	id assign Val ExprTail
9		|	print id
10	ExprTail	$\rightarrow$	plus Val ExprTail
11		|	minus Val ExprTail
12		|	λ
13	Val	$\rightarrow$	id
14		|	num

Figure 2.1: Context-free grammar for ac.

For the target of translation, we chose the Unix program dc (for *desk calculator*), which is a stacking (Reverse Polish) calculator. The target instructions must be acceptable to the dc program and faithfully represent the operations specified in an ac program. Compilation of ac to dc can be viewed as a study of larger systems, such as the portable Pascal and Java compilers, which produce a stack language as an intermediate representation [?].

2.2 Structure of an ac Compiler

The rest of this chapter presents a simple compiler for ac. The compiler's structure is based on the illustration in Figure Figure:one:onepointthree. Following are the main compilation steps.

1. The scanner reads a source program as a text file and produces a stream of tokens.

2. The parser processes tokens, determines the syntactic validity of the compiler's input, and creates an **abstract syntax tree** (AST) suitable for the compiler's subsequent activities.

3. The AST is walked to create a symbol table. This table associates type and other contextual information with the input program's variables.

4. Semantic checking is performed on the AST to determine the semantic validity of the compiler's input. The AST may be decorated with information required for subsequent compiler activities.

5. A translator walks the AST to generate dc code.

Do not copy

2.3 Formal Syntax Definition of ac

Before proceeding with creating the compiler's parts, in this section we specify the ac language more formally. For this, we use a **context-free grammar** (CFG) to specify the syntax of ac. The full grammar is shown in Figure 2.1. CFGs, first mentioned in Chapter Chapter:global:one, are discussed thoroughly in Chapter Chapter:global:four. Here, we view a CFG as a set of *rewriting rules*. A rewriting rule is also called a **production**. Following are two productions taken from the grammar for ac.

$$\begin{array}{lcl} \text{Stmt} & \rightarrow & \text{id assign Val ExprTail} \\ & | & \text{print id} \end{array}$$

On its **left-hand side** (LHS), each production has exactly one symbol; on its **right-hand side** (RHS), it may have zero or more symbols. The symbols after the arrow or bar are a production's RHS. Stmt is the LHS of each production shown here. In a production, *any* occurrence of its LHS symbol can be replaced by the symbols on its RHS. The productions shown here specify that a Stmt can be replaced by two different strings of symbols. The top production lets a Stmt be rewritten as an assignment to an identifier, while the bottom production lets a Stmt be rewritten to specify the printing of an identifier's value.

Productions contains two kinds of symbols: *terminals* and *nonterminals*. A **terminal** is a grammar symbol that cannot be rewritten. Thus id and assign are symbols for which there are no productions specifying how they can be changed. The **nonterminal** symbols Val and ExprTail have productions that define how they can be rewritten. To ease readability in the grammar, we adopt the convention that nonterminals begin with an uppercase letter and terminals are all lowercase letters.

The purpose of a CFG is to specify a (typically infinite) set of legal token strings. A CFG does this in a remarkably elegant way. It starts with a single nonterminal symbol called the **start symbol**. Then it applies productions, rewriting nonterminals until only terminals remain. Any string of terminals that can be produced in this manner is considered syntactically valid. A string of terminals that *cannot* be produced by any sequence of nonterminal replacements is deemed illegal.

There are two special symbols that appear in the grammars of this text. Both symbols are regarded as terminal symbols, but they lie outside the normal terminal alphabet and cannot be supplied as input.

- The special symbol λ represents the **empty**, or **null**, string. When present, it appears as the only symbol on a production's RHS. In Rule 7 of Figure 2.1, λ indicates that the symbol Stmts can be replaced by *nothing*, effectively causing its erasure.

- The special symbol \$ represents termination of input. An input stream conceptually has an unbounded number of \$ symbols following the actual input.

Do not copy

Sentential Form	Production Number
Prog	
Dcls Stmts $	1
Dcl Dcls Stmts $	2
floatdcl id *Dcls* Stmts $	4
floatdcl id *Dcl* Dcls Stmts $	2
floatdcl id intdcl id *Dcls* Stmts $	5
floatdcl id intdcl id *Stmts* $	3
floatdcl id intdcl id *Stmt* Stmts $	6
floatdcl id intdcl id id assign *Val* ExprTail Stmts $	8
floatdcl id intdcl id id assign num *ExprTail* Stmts $	14
floatdcl id intdcl id id assign num *Stmts* $	12
floatdcl id intdcl id id assign num *Stmt* Stmts $	6
floatdcl id intdcl id id assign num id assign *Val* ExprTail Stmts $	8
floatdcl id intdcl id id assign num id assign id *ExprTail* Stmts $	13
floatdcl id intdcl id id assign num id assign id plus *Val* ExprTail Stmts $	10
floatdcl id intdcl id id assign num id assign id plus num *ExprTail* Stmts $	14
floatdcl id intdcl id id assign num id assign id plus num *Stmts* $	12
floatdcl id intdcl id id assign num id assign id plus num *Stmt* Stmts $	6
floatdcl id intdcl id id assign num id assign id plus num print id *Stmts* $	9
floatdcl id intdcl id id assign num id assign id plus num print id $	7

Figure 2.2: Derivation of an ac program using the grammar in Figure 2.1.

Do not copy

Terminal	Regular Expression
floatdcl	`f`
intdcl	`i`
print	`p`
id	$\{$`a,b,c,...,z`$\}$ `-` $\{$`f,i,p`$\}$
assign	`=`
plus	`+`
minus	`-`
num	$\{0,1,\ldots,9\}^+ \mid \{0,1,\ldots,9\}^+.\{0,1,\ldots,9\}^+$

Figure 2.3: Formal definition of ac terminal symbols.

Grammars often generate a list of symbols from a nonterminal. Consider the productions for Stmts. They allow an arbitrary number of Stmt symbols to be produced. The first production for Stmts is recursive, and each use of this production generates another Stmt. The recursion is terminated by applying the second production for Stmts, thereby causing the final Stmts symbol to be erased.

To show how the grammar defines legal ac programs, the derivation of one such program is given in Figure 2.2, beginning with the start symbol Prog. Each line represents one step in the derivation. In each line, the leftmost nonterminal (in italics) is replaced by the underscored text shown on the next line. The right column shows the production number by which the derivation step is accomplished.

The derivation shown in Figure 2.2 explains how the terminal string is generated for the sample ac program. Although the CFG defines legal terminal sequences, how these terminals are “spelled” is another aspect of the language's definition. The terminal assign represents the assignment operator (typically spelled as `=` or `:=`). The terminal id represents an identifier. In most programming languages, an **identifier** is a word that begins with a letter and contains only letters and digits. In some languages, words such as `if`, `then`, and `while` are **reserved** and cannot be used as identifiers. It is the job of the *scanner* to recognize the occurrence of a string (such as `xy4z`) and return the corresponding token (id). Of course, the language designer must specify this correspondence.

Chapter Chapter:global:three defines *regular expressions* and shows how they can specify terminal spellings and automate the generation of scanners. By way of example, the regular expressions for terminals in ac are shown in Figure 2.3. The keywords are unusually terse so that the scanner can be readily constructed in Section 2.4. The specification given id allows any lowercase character except `f`, `i`, and `p`. The regular expression for num is the most complicated: The $|$ symbol denotes choice, and the $^+$ symbol denotes repetition. Thus a num can be a sequence of digits *or* a sequence of digits followed by a decimal point followed by another sequence of digits. For the remainder of this chapter, we consider translation of the ac program shown in Figure 2.4. The figure also shows the sequence of terminal symbols that corresponds to the input program. The derivation shown textually in Figure 2.2 can be represented as a derivation (or parse) tree, also shown in Figure 2.4.

In the ensuing sections, we examine each step of the compilation process for the ac language, assuming an input that would produce the derivation shown in Figure 2.2. While the treatment is somewhat simplified, the goal is to show the activity and data structures of each phase.

2.4 The Scanner's Job

The scanner's job is to translate a stream of characters into a stream of *tokens*. A **token** represents an instance of a terminal symbol. Rigorous methods for constructing scanners based on regular expressions (such as those shown in Figure 2.3) are covered in Chapter Chapter:global:three. Here, we are content with

Do not copy

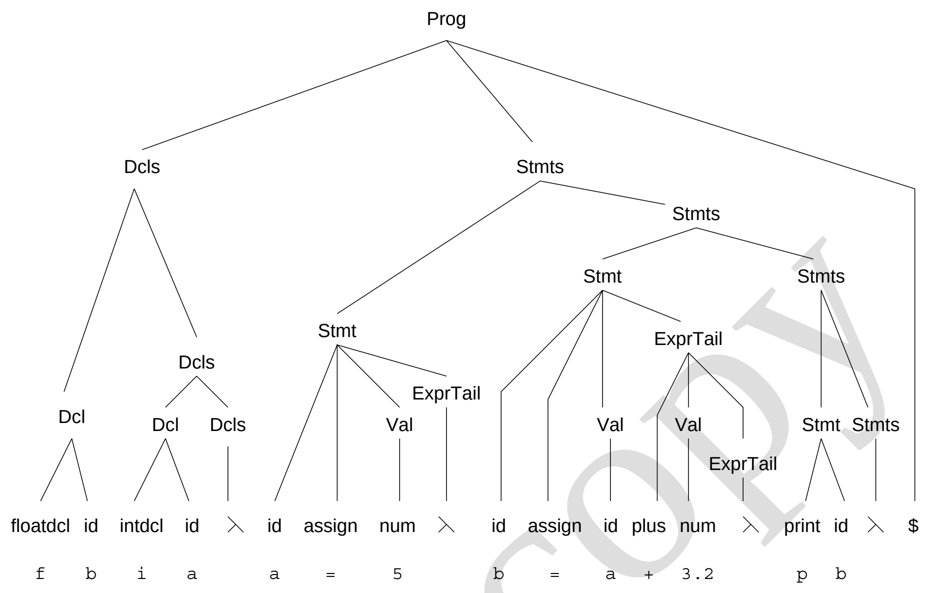
Prog
Dcls
Stmts
Stmts
Stmt
Stmts
ExprTail
Dcls
Stmt
ExprTail
Dcl
Dcl
Dcls
Val
Val
Val
Stmt
Stmts
ExprTail
floatdcl id intdcl id λ id assign num λ id assign id plus num λ print id λ $
f b i a a = 5 b = a + 3.2 p b

Figure 2.4: An ac program and its parse tree.

crafting an *ad hoc* scanner. While the automatic methods are more robust, the job at hand is sufficiently simple to attack manually.

Figure 2.5 shows the pseudocode for the basic scanner. This scanner examines an input stream and returns the stream's next token. As seen in this code, a token actually has two components, as follows.

- A token's *type* explains the token's membership in the terminal alphabet. All instances of a given terminal have the same token type.
- A token's *semantic value* provides additional information about the token.

For some tokens, such as plus and assign in ac, instance-specific information is unnecessary. Other tokens, such as id and num, require semantic information so that the compiler can record *which* identifier or number has been scanned. The code for scanning a number is the most complex, so it is relegated to the separate procedure SCANDIGITS. Although this scanner is written *ad hoc*, a principled approach supports its construction. The logic is patterned after the num token's regular expression. A recurring theme of this text is that the algorithms that enable automatic construction of a compiler's parts can often guide the manual construction of those parts.

The scanner, when called, must find the beginning of some token. Scanners are often instructed to ignore blanks and comments. Patterns for such input sequences are also specified using regular expressions, but the action triggered by such patterns does nothing. If the scanner encounters a character that cannot

Do not copy

```
function SCANNER( s ) : Token
    if s.EOF( )
    then  ans.type ← $
    else
        while s.PEEK( ) = blank  do  call s.ADVANCE( )
        if s.PEEK( ) ∈ {0, 1, 2, 3, 4, 5, 6, 7, 8, 9}
        then
            ans.type ← num
            ans.val ← STRINGTOINT( SCANDIGITS( ) )
        else
            ch ← s.ADVANCE( )
            switch (ch)
                case { a . . . z } − { i, f, p }
                    ans.type ← id
                    ans.val ← ch
                case f
                    ans.type ← floatdcl
                case i
                    ans.type ← intdcl
                case p
                    ans.type ← print
                case =
                    ans.type ← assign
                case +
                    ans.type ← plus
                case -
                    ans.type ← minus
                case default
                    call LEXICALERROR( )
    return (ans)
end
```

Figure 2.5: Scanner for the ac language.

Do not copy

```
function SCANDIGITS( $s$ ) : String
    $str \leftarrow$ “ ”
    while $s$.PEEK( ) $\in \{0,1,2,3,4,5,6,7,8,9\}$ do
        $str \leftarrow str + s$.ADVANCE( )
    if $s$.PEEK( ) $=$ “.”
    then
        $str \leftarrow str + s$.ADVANCE( )
        if $s$.PEEK( ) $\notin \{0,1,2,3,4,5,6,7,8,9\}$
        then call ERROR( “Expected a digit” )
        while $s$.PEEK( ) $\in \{0,1,2,3,4,5,6,7,8,9\}$ do
            $str \leftarrow str + s$.ADVANCE( )
    return ($str$)
end
```

Figure 2.6: Digit scanning for the ac language.

begin a token, then a **lexical error message** is issued; some scanners attempt to recover from such errors. A simple approach is to skip the offending character and continue scanning. This process continues until the beginning of *some* token is found. Once the scanner has found the beginning of a token, it then matches the longest possible character sequence that comprises a legal token.

Tokens come in many forms. Some are one character in length and cannot begin any other token. These are very easy to scan—a single character is read and the corresponding token is returned. Other tokens, such as + (in Java and C) are more difficult. When a + is seen, the next character must be inspected (but not yet read) to see if it extends the current token (for example, ++). Scanners can generally require a peek at the next character to determine whether the current token has been fully formed. For example, the end of a num token is signaled by the presence of a nondigit. Modular construction of the scanner is facilitated by allowing the next input character to be examined—perhaps many times—prior to consumption. This is typically accomplished by *buffering* the input. The method PEEK returns the contents of the buffer. The method ADVANCE returns the buffer's contents *and* advances the next character into the buffer.

Variable-length tokens, such as identifiers, literals, and comments, must be matched character-by-character. If the next character is part of the current token, it is consumed. When a character that cannot be part of the current token is reached, scanning is complete. When scanning is resumed, the last character inspected will be part of the next token.

2.5 The Parser's Job

The parser is responsible for determining if the stream of tokens provided by the scanner conforms to the language's grammar specification. In most compilers, the grammar serves not only to define the syntax of a programming language

Do not copy

```
procedure STMT( ts )
    if ts.PEEK( ) = id                                   1
    then
        call MATCH( ts, id )                             2
        call MATCH( ts, assign )
        call VAL( )
        call EXPRTAIL( )
    else
        if ts.PEEK( ) = print                            3
        then
            call MATCH( ts, print )
            call MATCH( ts, id )
        else    call ERROR( )
end
```

Figure 2.7: Recursive-descent parsing procedure for Stmt.

but also to guide the automatic construction of a parser, as described in Chapters Chapter:global:five and Chapter:global:six. In this section, we build a parser for ac using a well-known parsing technique called *recursive descent*, which is described more fully in Chapter Chapter:global:five. We also consider a representation for the parsed program that serves as a record of the parse and as a means of conveying information between a compiler's components.

2.5.1 Recursive-Descent Parsing

Recursive descent is one of the simplest parsing techniques used in practical compilers. The name is taken from the recursive parsing routines that, in effect, descend through the derivation tree that is recognized as parsing proceeds. In recursive-descent parsing, each nonterminal A has an associated *parsing procedure*. This procedure must determine if a portion of the program's input contains a sequence of tokens derivable from A. The productions for A guide the parsing procedure as follows.

- The procedure contains conditionals that examine the next input token to predict which of A's productions should be applied. If no production can be applied then then an error message is issued. For the grammar shown in Figure 2.1, the parsing procedures associated with the nonterminals Stmt and Stmts are shown in Figures 2.7 and 2.8.

  - id predicts the production Stmt$\rightarrow$id assign Val ExprTail and
  - print predicts the production Stmt$\rightarrow$print id.

  The production is selected by the conditionals at Steps **1** and **3**.

Do not copy

```
procedure STMTS( ts )
    if ts.PEEK( ) = id or ts.PEEK( ) = print                              4
    then
        call STMT( )
        call STMTS( )
    else
        if ts.PEEK( ) = $
        then
            /*              do nothing for λ-production              */   5
        else    call ERROR( )
end
```

Figure 2.8: Recursive-descent parsing procedure for Stmts.

- Having selected a production $A \rightarrow \alpha$, the parsing procedure next recognizes the vocabulary symbols in α.
  - For terminal t, the procedure MATCH(*ts*, t) is called to consume the next token in stream *ts*. If the token is not t then an error message is issued.
  - For a nonterminal B, the parsing procedure associated with B is invoked.

  In Figure 2.7, the block of code at Step **2** determines that id, assign, Val, and ExprTail occur in sequence.

  The parsing procedures can be recursive, hence the name "recursive descent." In the full parser, each nonterminal has a procedure that is analogous to the ones shown in Figure Figure:two:RecDesStmt,RecDesStmts.

In our example, each production for Stmt begins with a distinct terminal symbol, so it is clear by inspection that id predicts Stmt's first production and print predicts the second production. In general, an entire set of symbols can predict a given production; determining these **predict sets** can be more difficult.

- The production Stmts → Stmt Stmts begins with the nonterminal Stmt. This production is thus predicted by terminals that predict *any* production for Stmt. As a result, the predicate at Step **4** in Figure 2.8 checks for id or print as the next token.

- The production Stmts → λ offers no clues in its RHS as to which terminals predict this production. Grammar analysis can show that \$ predicts this production, because \$ is the only terminal that can appear *after* Stmts. No action is performed by the parser at Step **5** when applying the production Stmts → λ, since there are no RHS symbols to match.

Do not copy

The grammar analysis needed to compute predict sets for an arbitrary CFG is discussed in Chapter Chapter:global:four.

Mechanically generated recursive-descent parsers can contain redundant tests for terminals. For example, the code in Figure 2.7 tests for the presence of an id at Steps **1** and **2**. Modern software practice tolerates such redundancy if it simplifies or facilitates compiler construction. The redundancy introduced by one compiler component can often be eliminated by another, as discussed in Exercise 5.

Syntax errors arise when no production for the current nonterminal can be applied, or when a specific terminal fails to be matched. When such errors are discovered, a parser can be programmed to recover from the error and continue parsing. Ambitious parsers may even attempt to repair the error.

In practice, parsing procedures are rarely created *ad hoc*; they are based on the theory of *LL*(k) *parsing*, which is described in Chapter Chapter:global:five. Given a grammar for a language, the LL(k) method determines automatically if a suitable set of procedures can be written to parse strings in the grammar's language. Within each procedure, the conditionals that determine which production to apply must operate independently and without backtracking. Most tools that perform such analysis go one step further and automatically create the parsing code or its table representation.

2.5.2 Abstract Syntax Trees

What output should our recursive descent parser generate? It could generate a derivation tree corresponding to the tokens it has parsed. As Figure 2.4 shows, such trees can be rather large and detailed, even for very simple inputs. Moreover, CFGs often contain productions that serve only to *disambiguate* the grammar rather than to lend practical structure to the parsed inputs. In this section, we consider how to represent parsed inputs faithfully while avoiding unnecessary detail.

An **abstract syntax tree** (AST) contains the essential information in a derivation tree; unnecessary details are “abstracted out.” For example, an AST can elide inessential punctuation and delimiters (braces, semicolons, parentheses, and so on). An AST's design is also influenced by the needs of post-parsing activities. As a result, the AST's design is often revisited and modified during compiler construction. Chapter Chapter:global:seven considers the construction of ASTs in more detail. For our purposes, we place code in the recursive-descent parser to create and link the AST nodes into a tree. For example, our Stmt procedure becomes responsible for creating (and returning) a subtree that models the parsed Stmt.

The AST is designed to retain essential syntactic structure in a form that is amenable to subsequent processing. An AST for ac should be designed as follows.

- Declarations need not be retained in source form. However, a record of identifiers and their declared types must be retained to facilitate *symbol table construction* and *semantic type checking*, as described in Section 2.6.

Do not copy

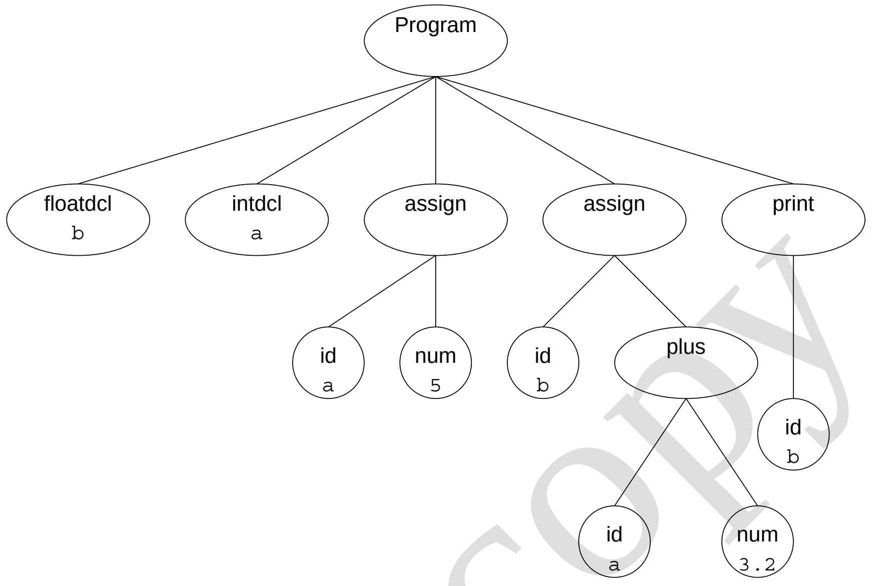

Figure 2.9: An abstract syntax tree for the ac program shown in Figure 2.4.

- The order of the executable statements is important and must be explicitly represented, so that *code generation* (Section 2.7) can issue instructions in the proper order.
- An assignment statement must retain the identifier that will hold the computed value and the expression that computes the value.
- A print statement must retain the name of the identifier to be printed.

Figure 2.9 shows the AST resulting from the sample ac program. Cleaner and simpler than the derivation tree, the AST still contains all essential program structure and detail.

2.6 Semantic Analysis

After the parser builds a program's AST, the next step is **semantic analysis**, which is really a catch-all term for checks to ensure that the program truly conforms to a language's definition. Any aspect of the language's definition that cannot be readily formulated by grammar productions is fodder for semantic analysis, as follows.

- Declarations and scopes are processed to construct a *symbol table*.
- Language- and user-defined types are examined for consistency.

Do not copy

```
procedure SymVisitNode( n )
    if n.kind = floatdcl
    then  call EnterSymbol( n.id, float )
    else
        if n.kind = intdcl
        then  call EnterSymbol( n.id, integer )
end
procedure EnterSymbol( name, type )
    if SymbolTable[name] = undefined
    then  SymbolTable[name] ← type
    else  call error( "duplicate declaration" )
end
```

Figure 2.10: Symbol table construction for ac.

- Operations and storage references are processed so that type-dependent behavior can become explicit in the program representation.

For example, in the assignment statement `x=y`, `x` is typically interpreted as an address and `y` is interpreted as a value. After semantic analysis, an AST identifier node always references the *address* of the identifier. Explicit operations are inserted into the AST to generate the value of an identifier from its address. As another example, some languages allow a single operator to have multiple meanings depending on the types of its operands. Such **operator overloading** greatly extends the notational power of a language. Semantic analysis uses type information to map an overloaded operator to its specific definition in context.

Most programming languages offer a set of **primitive types** that are available to all programs. Some programming languages allow additional types to be created, effectively extending the language's type system. In languages such as Java, C++, and Ada, semantic analysis is a significant task, considering the rules that must be enforced and the richness of the languages' primitive and extended types. The ensuing sections describe the comparatively simple semantic analysis for ac.

2.6.1 The Symbol Table

In ac, identifiers must be declared prior to use, but this requirement cannot be enforced at parse time. Thus, the first semantic-processing activity traverses the AST to record all identifiers and their types in a **symbol table**. Although the set of potential identifiers is infinite in most programming languages, we have simplified ac so that programs can mention at most 23 distinct identifiers. As a result, an ac symbol table has 23 entries, indicating each identifier's type: integer, float, or undefined. In most programming languages, the type information associated with a symbol includes other attributes, such as the identifier's scope, storage class, and protection.

Do not copy

Symbol	Type	Symbol	Type	Symbol	Type
a	integer	k		t	
b	float	l		u	
c		m		v	
d		n		w	
e		o		x	
g		q		y	
h		r		z	
j		s			

Figure 2.11: Symbol table for the ac program from Figure 2.4.

To create an ac symbol table, we traverse the AST, counting on the presence of a *declaration node* to trigger effects on the symbol table. In Figure 2.10, SYMVISITNODE shows the code to be applied as each node of the AST is visited. As declarations are discovered, ENTERSYMBOL checks that the given identifier has not been previously declared. Figure 2.11 shows the symbol table constructed for our example ac program. Blank entries in the table indicate that the given identifier is undefined.

2.6.2 Type Checking

Once symbol type information has been gathered, ac's executable statements can be examined for consistency of type usage. This process is called **type checking**. In an AST, expressions are evaluated starting at the leaves and proceeding upward to the root. To process nodes in this order, we perform type checking bottom-up over the AST. At each node, we apply SEMVISITNODE, shown in Figure 2.12, to ensure that operand types are either consistent or that a legal type conversion can be inserted to render the types consistent, as follows.

- For nodes that represent addition or subtraction, the computation is performed in float if either subtree has type float.
- For nodes representing the retrieval of a symbol's value, the type information is obtained by consulting the symbol table.
- Assignment demands that the type of the value match the type of the assignment's target.

Most programming language specifications include a **type hierarchy** that compares the language's types in terms of their generality. Our ac language follows in the tradition of Java, C, and C++, in which a float type is considered **wider** (*i.e.*, more general) than an integer. This is because every integer can be represented as a float. On the other hand, **narrowing** a float to an integer loses precision for some float values.

Do not Copy

```
procedure SemVisitNode( $n$ )
    switch ($n.kind$)
        case plus
            $n.type \leftarrow$ Consistent( $n.child1, n.child2$ )
        case minus
            $n.type \leftarrow$ Consistent( $n.child1, n.child2$ )
        case assign
            $n.type \leftarrow$ Convert( $n.child1, n.child2$ )
        case id
            $n.type \leftarrow$ RetrieveSymbol( $n.name$ ).$type$
        case num
            if ContainsDot( $n.name$ )
            then  $n.type \leftarrow$ float
            else  $n.type \leftarrow$ integer
end
function Consistent( $c1, c2$ ) : $Type$
    $m \leftarrow$ Generalize( $c1.type, c2.type$ )
    if $c1.type \neq m$
    then  call Convert( $c1, m$ )
    else
        if $c2.type \neq m$
        then  call Convert( $c2, m$ )
    return ($m$)
end
function Generalize( $t1, t2$ ) : $Type$
    if $t1 =$ float or $t2 =$ float
    then  $ans \leftarrow$ float
    else  $ans \leftarrow$ integer
    return ($ans$)
end
procedure Convert( $n, t$ )
    if $n.type =$ float and $t =$ integer
    then  call ERROR( "Illegal type conversion" )
    else
        if $n.type =$ integer and $t =$ float
        then  /* replace node $n$ by convert-to-float of node $n$ */
        else  /* nothing needed */
end
```

Figure 2.12: Type analysis for ac.

Do not copy

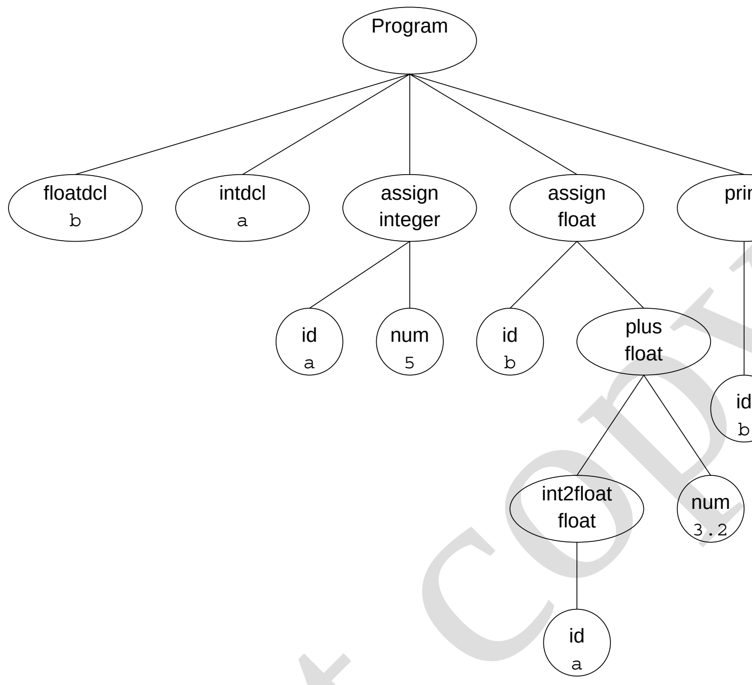

Figure 2.13: AST after semantic analysis.

Most languages allow automatic widening of type, so an integer can be converted to a float without the programmer having to specify this conversion explicitly. On the other hand, a float cannot become an integer in most languages unless the programmer explicitly calls for this conversion.

CONSISTENT, shown in Figure 2.12, is responsible for reconciling the type of a pair AST nodes using the following two steps.

1. The GENERALIZE function determines the least general (*i.e.*, simplest) type that encompasses its supplied pair of types. For ac, if either type is float, then float is the appropriate type; otherwise, integer will do.

2. The CONVERT procedure is charged with transforming the AST so that the ac program's *implicit* conversions become *explicit* in the AST. Subsequent compiler passes (particularly code generation) can assume a type-consistent AST in which all operations are explicit.

The results of applying semantic analysis to the AST of Figure 2.9 are shown in Figure 2.13.

Do not copy

2.7 Code Generation

With syntactic and semantic analysis complete, the final task undertaken by a compiler is the formulation of target-machine instructions that faithfully represent the semantics (*i.e.*, meaning) of the source program. This process is called **code generation**. Our translation exercise consists of generating code that is suitable for the dc program, which is a simple calculator based on a *stack machine* model. In a **stack machine**, most instructions receive their input from the contents at or near the top of an operand stack. The result of most instructions is pushed on the stack. Programming languages such as Pascal and Java are frequently translated into a portable, stack-machine representation [?].

Chapters Chapter:global:twelve, Chapter:global:thirteen, and Chapter:global:fifteen discuss code generation in detail. Automatic approaches generate code based on a description of the target machine. Such code generators greatly increase the portability of modern programming languages. Our translation task is sufficiently simple for an *ad hoc* approach. The AST has been suitably prepared for code generation by the insertion of type information. Such information is required for selecting the proper instructions. For example, most computers distinguish between instructions for float and integer data types.

We walk the AST starting at its root and let the nodes trigger code generation that is appropriate for their functions in the AST. The code generator shown in Figure 2.14 is recursive. In most programming languages, this conveniently accommodates the translation of AST constructs wherever they appear. The overall effect is to generate code for the program represented by the AST, as follows.

- For plus and minus, the code generator recursively generates code for the left and right subtrees. The resulting values are then at top-of-stack, and the appropriate operator is placed in the instruction stream to add or subtract the values.

- For assign, the value is computed and the result is stored in the appropriate dc register. The calculator's precision is then reset to integer by setting the fractional precision to zero; this is shown at Step **6** in Figure 2.14.

- Use of an id causes the value to be loaded from dc's register and pushed onto the stack.

- The print node is tricky because dc does not discard the value on top-of-stack after it is printed. The instruction sequence `si` is generated at Step **7**, thereby popping the stack and storing the value in dc's `i` register. Conveniently, the ac language precludes a program from using this register because the `i` token is reserved for spelling the terminal symbol integer.

- The change of type from integer to float at Step **8** requires setting dc's precision to five fractional decimal digits.

Do not copy

```
procedure CodeGen( $n$ )
    switch ( $n.kind$ )
        case Program
            foreach $c \in Children(n)$ do call CodeGen( $c$ )
        case assign
            call CodeGen( $n.child2$ )
            call Emit( “s” )
            call Emit( $n.child1.name$ )
            call Emit( “0 k” )                                        6
        case plus
            call CodeGen( $n.child1$ )
            call CodeGen( $n.child2$ )
            call Emit( “+” )
        case minus
            call CodeGen( $n.child1$ )
            call CodeGen( $n.child2$ )
            call Emit( “-” )
        case id
            call Emit( “l” )
            call Emit( $n.name$ )
        case print
            call CodeGen( $n.child1$ )
            call Emit( “p” )
            call Emit( “si” )                                         7
        case int2float
            call CodeGen( $n.child1$ )
            call Emit( “5 k” )                                        8
        case num
            call Emit( $n.name$ )
end
```

Figure 2.14: Code generation for ac

Do not copy

Code	Source	Comments
`5`	`a = 5`	Push 5 on stack
`sa`		Store into the `a` register
`0 k`		Reset precision to integer
`la`	`b = a + 3.2`	Push the value of the `a` register
`5 k`		Set precision to float
`3.2`		Push `3.2` on stack
`+`		Add
`sb`		Store result in the `b` register
`0 k`		Reset precision to integer
`lb`	`p b`	Push the value of the `b` register
`p`		Print the value
`si`		Pop the stack by storing into the `i` register

Figure 2.15: Code generated for the AST shown in Figure 2.9.

Figure 2.15 shows how code is generated for the AST shown in Figure 2.9. The horizontal lines delineate the code generated in turn for each child of the AST's root. Even in this *ad hoc* code generator, one can see principles at work. The code sequences triggered by various AST nodes dovetail to carry out the instructions of the input program. Although the task of code generation for real programming languages and targets is more complex, the theme still holds that pieces of individual code generation contribute to a larger effect.

This finishes our tour of a compiler for the ac language. While each of the phases becomes more involved as we move toward working with real programming languages, the spirit of each phase remains the same. In the ensuing chapters, we discuss how to automate many of the tasks described in this chapter. We develop the skills necessary to craft a compiler's phases to accommodate issues that arise when working with real programming languages.

Do not copy

Exercises

1. The CFG shown in Figure 2.1 defines the syntax of ac programs. Explain how this grammar enables you to answer the following questions.

   (a) Can an ac program contain only declarations (and no statements)?

   (b) Can a print statement precede all assignment statements?

2. Sometimes it is necessary to modify the syntax of a programming language. This is done by changing the CFG that the language uses. What changes would have to be made to ac's CFG (Figure 2.1) to implement the following changes?

   (a) All ac programs must contain at least one statement.

   (b) All integer declarations must precede all float declarations.

   (c) The first statement in any ac program must be an assignment statement.

3. Extend the ac scanner (Figure 2.5) so that the following occurs.

   (a) A floatdcl can be represented as either `f` or `float`. (That is, a more Java-like declaration may be used.)

   (b) An intdcl can be represented as either `i` or `int`.

   (c) A num may be entered in exponential (scientific) form. That is, an ac num may be suffixed with an optionally signed exponent (`1.0e10`, `123e-22` or `0.31415926535e1`).

4. Write the recursive-descent parsing procedures for all nonterminals in Figure 2.1.

5. The recursive-descent code shown in Figure 2.7 contains redundant tests for the presence of some terminal symbols. Show how these tests can be eliminated.

6. In ac, as in many computer languages, variables are considered *uninitialized* after they are declared. A variable needs to be given a value (in an assignment statement) before it can be correctly used in an expression or print statement.

   Suggest how to extend ac's semantic analysis (Section 2.6) to detect variables that are used before they are properly initialized.

7. Implement semantic actions in the recursive-descent parser for ac to construct ASTs using the design guidelines in Section 2.5.2.

8. The grammar for ac shown in Figure 2.1 requires all declarations to precede all executable statements. In this exercise, the ac language is extended so that declarations and executable statements can be interspersed. However, an identifier cannot be mentioned in an executable statement until it has been declared.

Do not copy

(a) Modify the CFG in Figure 2.1 to accommodate this language extension.

(b) Discuss any revisions you would consider in the AST design for ac.

(c) Discuss how semantic analysis is affected by the changes you envision for the CFG and the AST.

9. The code in Figure 2.10 examines an AST node to determine its effect on the symbol table. Explain why the order in which nodes are visited does or does not matter with regard to symbol-table construction.

Do not copy