Compiler Design ReferencePoint Suite

Table of Contents

Point 1: Introducing Compiler Design

Vikas Singhal

A compiler is a translator that reads a high-level language (HLL) program and converts it to a low-level language (LLL) program using various modules. The most common compiler is a two-pass compiler. In the first pass, the compiler converts the source code to an intermediate code. In the second pass, the compiler converts the intermediate code to the object code, which is interpreted by a computer. The first pass is called the front-end and the second pass is called the back-end. The front-end depends on the language of the source code and the back-end depends on the computer used to convert the source code to object code.

This ReferencePoint describes a compiler and its structure. In addition, it explains the various modules of a compiler.

Compiler Structure

Converting an HLL program to an LLL program is a complex process. The compilation process occurs in multiple steps, where each step denotes the processing of a module. Figure 1-1-1 shows the modules of a compiler:

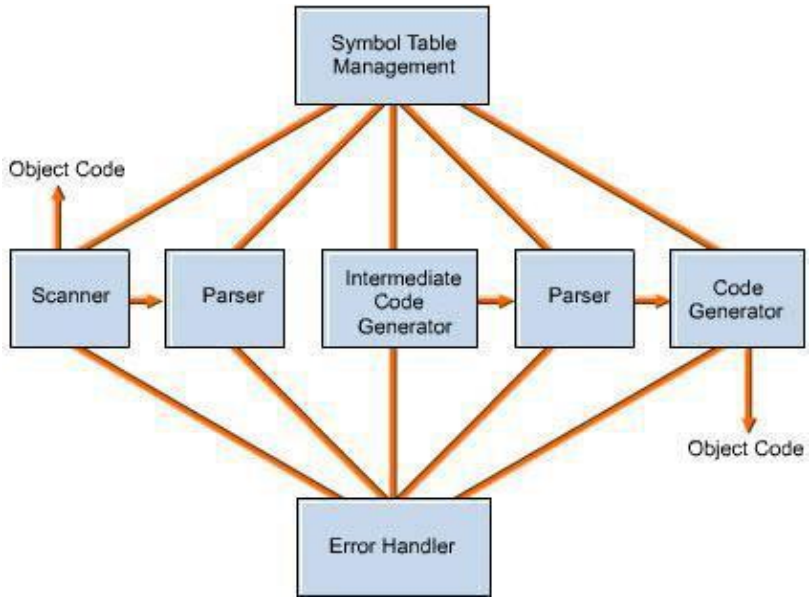

Figure 1-1-1: Modules of a Compiler

The various modules available in a compiler are:

- Scanner: Groups the characters present in the source code based on their behavior. These groups are called tokens. A token can be a group of identifiers, operands, or keywords.
- Parser: Accepts the tokens returned by the scanner as input. The parser represents a token in the form of an integer. For example, 1 can represent keywords and 2 can represent identifiers. The parser also creates a syntactic structure, which is a tree structure with leaf nodes representing tokens. The intermediate leaves of the tree structure represent expressions and statements.
- Intermediate Code Generator: Creates instructions based on the input provided by the parser. The intermediate code generator creates instruction in the three-address code format. Each expression contains a group of operands and a single operator.

- Code optimizer: Optimizes the code provided by the intermediate code generator. This module is optional in a compiler structure and helps the code utilize less storage space. The code optimizer also ensures that the optimized code runs faster than the intermediate code.

- Code generator: Creates the object code. This module decides where to store the object program and how to use the object program to access data. The code generator also enables computations in specified registers.

The routines that the modules of the compiler structure access are:

- Symbol Table Management: Stores the names and symbols used while converting the source code to object code. A Symbol table also stores the data type of each symbol in the table.

- Error Handler: Returns an error-free source code as an input to the next module. It is invoked when an error occurs in the source code. The compiler processes the first two modules, the scanner and the parser, of the source code that contains errors. As a result, the compiler detects the maximum number of errors in a single compilation.

Passes

To make the compiler design simple, you can combine the various sections of the compiler structure modules. The grouping of modules is called a pass, which:

- Reads the source code or takes the output of the last pass as input.

- Transforms one module to another based on specifications.

- Writes the results of an intermediate file.

The intermediate file is transferred to another pass. The number of passes for the compiler structure depends on the language used to write the source code.

It is convenient to work with a single-pass compiler. But in certain languages, a two-pass compiler generates object code. For example, Algorithmic Language 68 (ALGOL 68) enables you to define variables before they are declared. ALGOL 68 requires a two-pass compiler because a single-pass compiler cannot generate code for undeclared variables. The first pass declares the variable and the second pass generates code for the specified variable.

The issues that you need to consider before deciding the number of passes are:

- A two-pass compiler occupies less space than a single-pass compiler because the second pass can reuse space used by the first pass.

- A two-pass compiler is slower than a single-pass compiler because a two-pass compiler creates an intermediate file.

Computers with a large RAM use single-pass compilers and computers with less memory use two-pass compilers. You can reduce the number of passes needed to process the source code by using backpatching.

Backpatching

Backpatching is a technique in which multiple compiler modules are grouped in a single-pass. This technique does not require the reading or writing of intermediate files between compiler modules. If the output of a module depends on the input of other modules, output is obtained along with slots. The slots are memory spaces reserved for the results of various modules. These slots are filled when the modules run. An example of a compiler statement is:

```
GOTO M, where M is the Label
```

This statement requires a two-pass compiler for processing. In the first pass, the compiler enters the symbol table, which contains labels and variables used in source code with their equivalent machine address. In the second pass, object-level equivalents replace mnemonic code, such as GOTO, and computer-address equivalents replace variables.

A single-pass compiler creates the machine-instruction equivalent for the identifier GOTO at its first instance. Next, single-pass compiler adds the address of the machine instruction to the list of instructions to be backpatched. Backpatching is performed when the compiler retrieves the machine address for the label M.

For compilers, backpatching is performed on small lines of code. For example, labels are backpatched within a subroutine. The backpatched source code should be accessible until backpatching is complete. A language, such as ALGOL, where source code is made of a single subroutine, requires a two-pass compilation. This is because a single jump traverses the entire subroutine during backpatching. Unlike ALGOL, Fortran ensures that lengthy source code is divided into subroutines. As a result, backpatching is performed on one subroutine at a time.

Scanning

A scanner groups the characters from the source code in the form of tokens. A scanner uses regular expressions to decide the type of tokens used for a specific source language. After deciding the type of tokens, the transition diagram and finite automata techniques recognize the tokens. In addition, a scanner performs certain actions for the recognized tokens, such as inserting the identifier value in the symbol table.

The scanner can be a pass, which writes the results in an intermediate file. As a rule, the parser and the scanner are in the same pass. The scanner returns the representation of the searched token to the parser. The various token representations are:

- Integer code: Represents simple tokens, such as commas or colons.
- Integer code and a pointer to the symbol table: Represents complex tokens, such as identifiers. An integer returns the token type and the pointer points to the value of the token in the symbol table.

The scanner uses the input buffer to read the characters from the source code. Figure 1-1-2 shows the scheme used by the input buffer to read the characters from the source code:

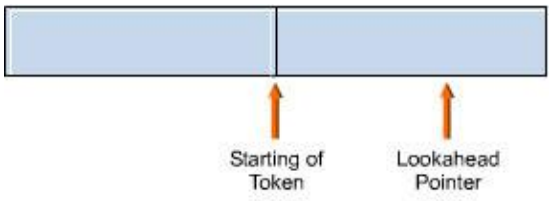

Figure 1-1-2: The Input Buffer
The input buffer is divided into two halves.The first half is already scanned while the second half contains two pointers. The first pointer points to the start of a token. The second pointer, which is the Lookahead pointer, scans the input buffer until the token is completely recognized. For example, in the code:

```
PROCEDURE (PAR 1, PAR 2, PAR 3...PAR n)
```

When the first pointer scans the character P of the word Procedure, the Lookahead pointer starts scanning the rest of the keyword. The keyword is recognized only when the Lookahead pointer finds the second E of the word Procedure. The Lookahead pointer can scan the second half before moving to the first half only if the first half of the input buffer is loaded with the rest of the characters from the source code.

Design of Scanner

You can use a flowchart to define how the scanner reads the character from the input buffer or the source code. The flowchart used to design a scanner is called a transition diagram. In a transition diagram, the circles that denote the processing in the flowchart are called states. The various states in a transition diagram are linked with arrows called edges. The labels on the edges denote the output character from a particular state. A transition diagram is shown in Figure 1-1-3:

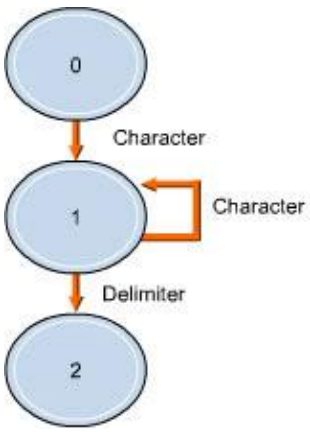

Figure 1-1-3: A Transition Diagram
The design created by the transition diagram is converted to code. The code is developed separately for each state of the transition diagram. The code developed for state zero is:

```
CH := GETCH();
if ( ALPHA (CH) )
        GOTO state 1
else
        EXIT ()
```

The above code shows that the function GETCH () reads the character from the input buffer. The ALPHA () function checks whether or not the character is an alphabet. If the character is an alphabet, the state changes to one.

The code for state one is shown in Listing 1-1-1:

Listing 1-1-1: State One Code

```
CH := GETCH();
if ( ALPHA (CH) OR DIGIT (CH) )
        GOTO state 1
else if ( END (CH) )
        GOTO state 2
else
        EXIT ()
```

The above code shows that state 2 is iterated if the character searched is either an alphabet or a digit. The DIGIT () function checks whether or not the character searched is a digit. The END () function returns the true value if the character searched is neither an alphabet nor a digit.

Regular Expressions

Regular expressions are used to describe tokens for programs that generate the scanner automatically. Regular expressions define the subset of languages called regular sets. The regular expression notations are used for tokens, which are a sequence of strings. For example, the string ABABABABAB is the sequence of the string AB. With regular expressions, the notation used is:

```
alphabet (alphabet | digit)*
```

The vertical bar | denotes the combination of the subsets in parenthesis. The closure operator asterisk denotes the subset within a parenthesis that can have multiple instances. For example, the expression ABCBCBCBCBC is defined with regular expressions A (BC)*.

The rules to define the regular expressions over the alphabet are:

- is a regular expression that specifies the language and can contain only an empty string, { }.
- If b is the symbol in then b is a regular expression that denotes the language using one string, {b}.
- If r and s are regular expressions denoting languages L (r) and L (s) respectively, then:
  - (s)* is a regular expression, which denotes L (r)*.
  - (r). (s) is a regular expression, which denotes L (r).L (s).
  - (r) | (s) is a regular expression, which denotes (L (r)) U (L (s)).

The precedence rule avoids the formation of regular expression using parenthesis. The precedence in ascending order is the vertical bar (|), dot (.), and asterisk (*).

Finite Automata

Finite Automata is a transition diagram used to convert a regular expression to a recognizer. A recognizer is a program that recognizes the token in the source code. The finite automaton is also known as Nondeterministic Finite Automaton (NFA). Figure 1-1-4 illustrates the NFA, which identifies the language (a | b)* bcc:

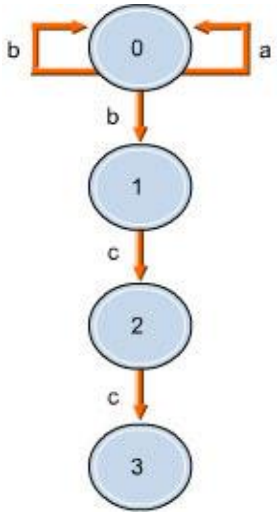

Figure 1-1-4: Nondeterministic Finite Automaton

Transition tables are used to represent the NFA in a tabular form. For the NFA discussed in Figure 1-1-4, the transitions are listed in Table 1-1-1:

Table 1-1-1: Transition Table for Language (a | b)*bcc

State	Input a	Input b	Input c
0	{0}	{0, 1}	-
1	-	-	{2}
2	-	-	{3}

Lex

Lex is the tool used to generate the lexical analyzer, which is also called a scanner. Lex creates the scanner, which is in the form of a stream of regular expressions. Lex contains regular expressions along with the action specified for each regular expression. The action is the source code, which is invoked, whenever the regular expression is recognized. Lex source code indicates the parser with the token recognized and makes an entry of the recognized token in the symbol table.

Lex can be viewed as the compiler for an HLL, which writes the code for scanners. Unlike other programming languages, Lex does not provide the details of the computations that have to be done. The output of the Lex code is the simulation of the finite automation. The transition table is the input to this program. The transition table is the part of the Lex output, which is directly created from the input to the Lex program as illustrated in Figure 1-1-5:

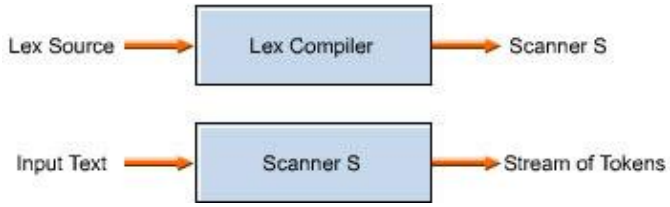

Figure 1-1-5: Lex Functioning

The Lex program is made of two parts. The first part contains the stream of auxiliary definitions and the second part is made up of a sequence of translation rules. The auxiliary statements are in the form:

```
Di = Ri,
```

Where, Di is the unique name and Ri is the regular expression. The symbols of regular expression are selected form the characters of the previously defined names, which is U {D1, D2, ...Di-1}, where denotes the character set. The translation statements for the Lex program are in the form:

```
P1  {A1}
P2  {A2}
.
.
Pn  {An}
```

Where, each P is the regular expression, which is also called a pattern. The regular expression, P, is implemented over the character set, . Ai specifies the piece of code that is invoked to perform the action for the scanned token. All Ai are compiled to object code to create the scanner S.

The scanner S scans the input, one character at a time, until it searches the longest prefix of the input stream that matches with any regular expression, Pi. When the prefix is found, it is removed from the input stream and stored in the buffer called a token. This token can also be the group of two pointers that points to the beginning and end of the matched string from the input character set. After storing the prefix in the token, the scanner invokes the code represented by Ai. After the Ai is compiled, the scanner transfers the control to the parser.

Parsing

Parser converts the stream of tokens into a parse tree. Parsers also check whether the input string is a part of the context-free grammar, which specifies the syntax of a specified language. To know more about context-free grammar, refer to an example of a do-while statement of C programming language:

```
do (statement) while (statement)
```

The above statement is the concatenation of the do keyword, a statement, the while keyword, and another statement. If statements are denoted by s, the do-while statement is written as:

```
s -> do (s) while (s)
```

The above rule is known as production. In this rule, the scanner elements are known as tokens and variables are called non-terminals. The elements of context-free grammar are:

- A terminal symbol, which is a stream of tokens.
- A non-terminal symbol.
- Productions, which are made of an arrow, non-terminal, and stream of non-terminals and/or tokens. This stream makes the left side of the production and non-terminal makes right side of the production.

- A start symbol, which is a non-terminal.

Parsers create the parser tree for the input string, if and only if the input string is a sentence of the specified grammar. If the input string is not a sentence of the specified grammar, an error message appears.

Parser scans the input character one at a time from left to right. The two approaches used by parsers are top-down and bottom-up. In the top-down approach, parser creates the parse tree from the root node to the leaf node. In the bottom-up approach, the parser creates the parse tree from the leaf node to the root node.

Bottom-Up Parsing

Bottom-up parsing uses the shift-reduce technique that shifts the character from the right side of the production to the stack until the character in the extreme right is on the top of the stack. Later, the left side characters of the production are moved to the right side and same process of shifting the characters to the stack is repeated. Shift-reduce parsing reduces the input string of the non-terminal start symbol of the grammar. Each character of the input string is matched with the productions and if a match is found, the input character is replaced by the right side of the production. For example, in the grammar:

```
S -> aAcBe
A -> Ab
B -> d
```

The input string is abbcde. Shift-reduce parsing reduces the string to the start symbol S. The string abbcde is matched with the right side of the productions and is replaced with the right side of the production for the matched characters.

For example, b and d are two characters that can be replaced in the string because both b and d are the characters in the extreme right of the productions. Replacing the character in the extreme left b with A creates the intermediate string aAbcde. The characters that can be replaced are e, d, b, and Ab. If you replace Ab with the right side of the given production, the resultant string is aAcde. The character that can be replaced is d, so the resultant string is aAcBe, which is represented by the start symbol S.

Replacing the right side of the production with its left side is called reduction. The substrings, which are the right side of the production and are used to reduce the input strings, are called handles. Examples of handles are Ab and d. The right side of the production, which cannot convert the input string into the start symbol, is not the handle.

Top-Down Parsing

Top-down parsing starts from the root node and moves toward the leaf nodes. It also uses backtracking to iterate through the input stream. Top-down parsing searches for the replacement of the left most character of the input string. It creates the parse tree for the input stream in the preorder starting with the root node. For example, a parse tree is created for the grammar:

```
S -> cAd
A -> abc | a
```

The input stream provided to the top-down parser is:

```
cad
```

The parse tree is initially created with a single node called the start node. Presently, the input pointer points to the first character of the input stream. The pointer is pointing to the character c. The parse tree at this stage is shown in Figure 1-1-6:

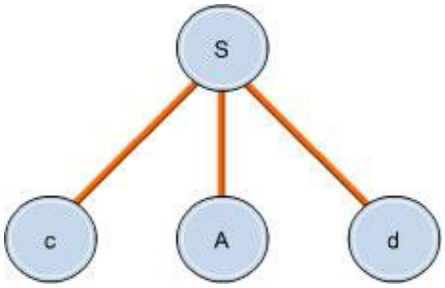

Figure 1-1-6: Parse Tree for Root Node

The first character of the input stream matches with the left most leaf. The input pointer moves to the second character a. The top-down parser scans the second node A and expands it as shown in Figure 1-1-7:

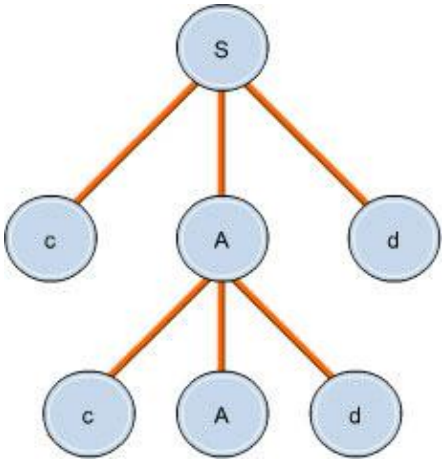

Figure 1-1-7: Parse Tree with the Second Node Expanded

The input pointer moves back to the second character of the input stream a. You can try the second option for production to obtain the parse tree as shown in Figure 1-1-8:

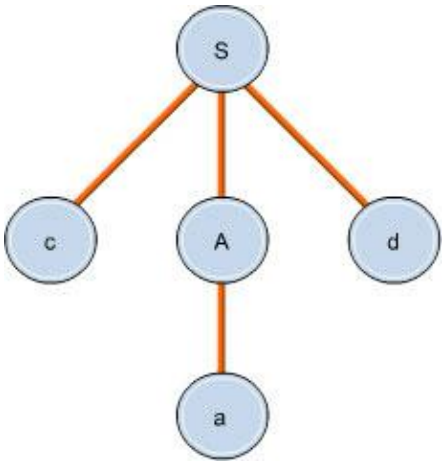

Figure 1-1-8: Parse Tree with the Second Option

Left to Right Parsing

Left to Right (LR) parser is a shift-reduce parser, which generates larger tables than Left-to-Left (LL) parsers. LR parser is made of a stack and a state table. There are three types of LR parsers: Simple LR (SLR), Canonical LR (CLR), and Look Ahead LR (LALR). An SLR parser is the easiest to implement but it fails to generate a parsing table for ambiguous grammar. CLR works on a large class of context-free grammar. LALR is implemented well to construct the parse tree.

Similar to the shift-reduce parser, an LR parser also reads the input from left to right. The benefits of using an LR parser are:

- Detects syntactic errors.
- Recognizes context-free grammar constructs.

The components of the LR parser are the driver routine and the parsing table. All LR parsers contain the same driver routine but different parsing tables. The LR parser is shown in Figure 1-1-9:

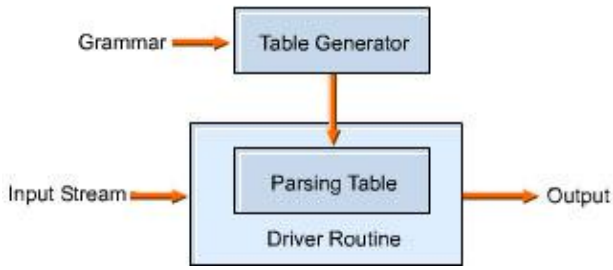

Figure 1-1-9: LR Parser

Yacc

Yacc is a parser generator that is used to implement a number of compilers. Yacc evaluates the input character stream and checks whether the input stream is syntactically correct. Yacc is used as a command in the Unix operating system. Yacc is illustrated in Figure 1-1-10:

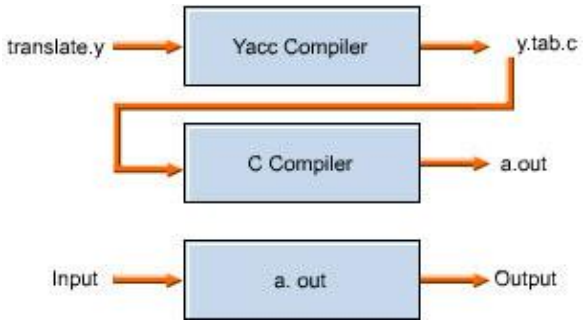

Figure 1-1-10: Yacc as Input/Output Translator

The translate.y file is converted into a HLL, C, using the command:

```
yacc translate.y
```

The converted file is named y.tab.c. This C file contains all the subroutines made by you and a parser, which is also written in C language. The C file is compiled as:

```
cc y.tab.c -ly
```

In the above command, ly is the library, which contains the parsing code. The name, ly, depends on the system so you may get some other name for the library ly. The object file prepared receives the input and generates the output stream. If you need some other user-defined or built-in functions, you can load them with y.tab.c. Then you can compile the combined file to obtain the object file a.out, which does the translation.

The input to the yacc translator is a group of context-free grammar rules that specify the structure to be used in the program. For example, a grammar rule can be:

```
address : street  ','  city  ',' and zip;
```

The structures, street, city, and zip are the input to the parser generator. The colon and semicolon are a part of the syntax and are not produced with the input. The comma within single quotes appears in the input. So the address is specified as:

```
87 Baker Rd , Michigan , 49048
```

The scanner completes the process of scanning the input characters. The scanner scans the tokens and intimates the parser regarding the scanned tokens.

Intermediate Code Generation

The compiler develops intermediate code, which is a language that reduces the complexities of source code. The intermediate code generator creates instructions based on the input provided by the parser. If the intermediate code generator is not used, the compiler has to convert source code directly to object code. This requires a large number of registers to hold the values of intermediate calculations. You need the intermediate code generator to effectively use the registers. Various intermediate codes used by the compiler are postfix notation, syntax tree, and three-address coding. The intermediate code must have two characteristics:

- It must be developed with less effort and should use minimum registers.
- It must be easily and effectively converted to object code.

The postfix notation reduces the complexities because it removes all parenthesis used in the expression and places the operator after the operands. This way a single stack can be used to evaluate the expression.

The syntax tree eliminates all the redundant information from the parse tree. This results in effective use of the compiler resources such as registers. The nodes in the syntax tree represent the operator and the leaf nodes represent operands. In three-address coding, the instructions created contain a single operator and three operands.

Postfix Notation

The conventional way to write algebraic expressions is to insert operator between the operands. The postfix notation is also called the reverse polish notation. The polish notation is the prefix notation where the operator is written before the operands. The addition of two operands, a and b, is denoted as a + b. This notation of writing algebraic expressions is known as the infix notation. The postfix notation places the operators on the right side of the operands. The infix expression a + b is written as ab+ in postfix notation. The postfix notation does not contain parenthesis.

The postfix notation represents operands as a single letter and operators as a single character. These letters and characters are called tokens. The infix expression (a + b) / c is written as ab+c/ in postfix notation. To convert infix expression to postfix expression, you must know the precedence hierarchy of the operators. Multiplication (*) and division (/) have the same precedence and the highest priority. These are followed by addition (+) and subtraction (-) that have the same precedence. A terminating symbol is placed to the extreme right of the infix expression to mark the end of the infix expression. This expression is also pushed on to the stack. The rules to convert the infix expression to postfix expression are:

- Start scanning the infix expression from left to right, one character at a time, and place the operand searched to the output stream.
- Push the left parenthesis to the stack.

- If a right parenthesis is scanned, pop all the operators from the stack to the output stream until a left parenthesis is encountered. Finally, both left parenthesis and right parenthesis are deleted.

- If the operator being scanned has higher priority than that of the operator on top of the stack, the scanned operator is pushed to the stack.

- If the operator being scanned has equal or lower priority than the operator on top of the stack, all the operators in the stack are moved to the output stream.

- When the terminating symbol is scanned, the elements of the stack are moved to the output stream until the terminating symbol is searched on the stack. Then this algorithm terminates.

Evaluating a postfix is easier than creating a postfix expression. The postfix expression to be evaluated is scanned from left to right. The scanned operands are moved to the stack. When an operator is scanned, the top most two operands are moved out of the stack and the expression is evaluated using the scanned operator. The result obtained is placed back in the stack. This process is repeated until the last operand in the input stream is moved out of the stack. For example, the postfix expression 3 2 + 8 3 – / is evaluated in the steps:

1. Move 3 and 2 to the top of the stack.

2. Perform the calculations with the addition operator and place the result 5 back in the stack.

3. Move 8 and 3 to the stack.

4. Perform the calculations with the subtraction operator and place the result 5 back in the stack.

5. When division (/) operator is scanned, perform the calculations with the remaining two operands as 5 / 5 and the result obtained is 1.

Syntax Tree

The syntax tree is used to represent the intermediate representation of the source code independent of scanning and parsing. The syntax tree is used to present language constructs. It is obtained from the parse tree. The syntax tree for the production P -> if S then P1 else P2 is illustrated in the Figure 1-1-11:

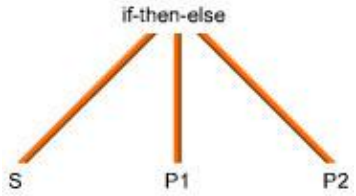

Figure 1-1-11: Syntax Tree

Syntax trees are developed in the same manner as the postfix expressions are created. Syntax tree contains nodes for all operators and operands in the expression for which the syntax tree has to be created. The operator node contains child nodes. These child nodes are the root nodes for the operands of that particular operator. Each operator node contains various fields. One field specifies the operator and the other fields contain the pointer to the operands that are linked with the operator. The operator is also expressed as the label of a node. The functions that are used to create the nodes of a syntax tree are:

- mkleaf (lbl, val): Creates an integral node with the label lbl and a field containing the value val for an integer.
- mkleaf (lbl, ent): Creates an identifier node with label lbl and a field containing the pointer ent that points to the entry of the identifier in the symbol table.
- mknode (lbl, lft, rht): Creates an object node with the label lbl and two fields that point to the left and the right subnodes.

Three-Address Code

Three-address code is the stream of statements in the form:

```
a := b operator c,
```

where a, b, and c are variables or constants. These variables can also be the temporary variables to hold the value of the operands. For example, the source code expression such as a * b + c – d is changed to an intermediate form using the three-address code:

```
t1 := a * b
t2 := t1 + c
t3 := t2 - d
```

where, t1, t2, and t3 are the temporary variables that are created by the compiler to temporarily hold the variable values. The characteristics of the three-address code are:

- Three-address statement can have a maximum of one operator along with the assignment operator. The compiler has to take care of the order of the operators. The three-address instructions must be made in appropriate order.
- Compiler must create the temporary variables to hold the values of the intermediate calculations.
- The three-address instructions can also have less than three operands.

Code Optimization

Code optimizer improves the code provided by the intermediate code generator. The optimized code takes less storage space and runs faster than the intermediate code. The optimization of the code depends on the environment and the type of intermediate code. For a large intermediate code, the time taken to run the code is more important than the space required to store the code.

Code optimization is done for the intermediate codes that have to be used at frequent intervals. Before optimizing the intermediate code, the compiler checks for resources needed to optimize the code. In certain conditions, optimization is not needed and intermediate code is directly converted to object code.

Code optimization can be performed either locally on the source code or inside the loop. The code optimization performed inside the loop is called loop optimization while the optimization performed locally at source level is called local optimization. The optimization sources refer to the location in the intermediate code where the optimization is performed.

Optimization Sources

Code optimization methods are used after the source code has gone through the scanner and the parser module. An optimization method checks for loopholes in the intermediate code and replaces the loopholes with well-formed constructs. Registers, which are used to hold the variables, are responsible for improving the performance of the intermediate code.

After the registers are optimally allocated, the compiler looks for the inner loops in the intermediate code. The local loops use more than 90 percent of the compiler resources during compilation. The existence of loops also forces compilers to use the registers to temporarily hold the values of intermediate calculations. The optimizer reduces the loops used in intermediate code to the minimum possible number.

Some HLL can be optimized at the source level. These languages enable the efficient use of compilers. On the other hand, certain HLLs do not enable the optimization to be performed at the source level. For example, Fortran uses arrays that do not refer through pointers. You cannot make offset calculations with arrays or perform optimization at the source level.

Another optimization source is the identification of run-time calculations. An efficient source code must have the least number of run-time calculations and the maximum number of compile-time calculations. For example, a single expression:

```
arr1 [k + 1] := arr2 [k + 1] ;
```

is easier to write as compared to:

```
i := k + 1;
arr1 [i]  := arr2 [i];
```

Local Optimization

Local optimization refers to the local changes made in the intermediate code to improve the performance of the code. The local optimization is done on unconditional control statements. Listing 1-1-2 shows how the local optimization is done directly in code:

Listing 1-1-2: Intermediate Code for the GOTO Statement

```
10:     if num1 > num2  goto  20
        goto 30
20:     temp1 := num2 + 5
        if num1 < temp1 goto 40
        goto 30
40:     num1 := num1 + num2
        goto 10
30:
```

This code uses multiple GOTO statements. It checks for a condition and uses the GOTO statement to redirect the control for processing. An example for using conditional redirections is:

```
if num1 > num2  goto  20
        goto 30
```

The above code is converted to object code by creating the compiler statements:

1. Set the condition codes after setting num1 and num2.

2. Move to label 20, if the code is set for >.

3. Move to label 30.

If num1 is greater than num2 for more than fifty percent of the time then step1 and step 2 are executed all the time, and step 3 is executed half of the time. Therefore, 2.5 steps are executed on an average.

The compiler replaces the lines of code that use multiple GOTO statements by a single line of code, which is:

```
If num1 < OR = num2 goto 30
```

The above code is converted to object code by creating the compiler statements:

1. Set the condition codes after setting num1 and num2.

2. Move to label 30 if the code is set for < or =.

If num1 is greater than num2 for fifty percent of the time then step1 and step 2 are executed every time, saving half the execution time. If all the variables take the same storage space then the single line expression also saves the storage space.

Loop Optimization

A large part of the execution time for a program is spent in loop processing. The code inside the loops is reduced to achieve loop optimization. This can result in an increase of code outside the loop, which will not have any adverse reaction on loop optimization. The technique used to reduce the code inside the loop is called code motion. The transformation removes those expressions from the loop that returns the same result every time. These expressions are placed just above the loop so that they are not iterated each time the loop is run, as shown in this example:

```
for (i = 1; i < j + 20; i + +)
```

In the above code, each time the loop is iterated, the comparison is done with j + 20. The code motion will move this expression out of the loop as:

```
k = j + 20;
for (i = 1; i < k; i + +)
```

The above code helps reduce the execution time of the code.

The code to multiply two matrix, a[i] and b[i], of order 1 X 10 and 10 X 1, respectively, is:

```
mul = 0;
i =1;
for (i = 1; i <= 10; i + +)
mul = mul + a[i] * b[i];
```

The above code shows that two matrices a and b are multiplied to obtain the product mul. The equivalent machine code using three-address code scheme is shown in Listing 1-1-3:

Listing 1-1-3: Three-Address Code for Matrix Multiplication

```
1.      mul := 0
2.      i := 1
3.      loc := 4 * i
4.      arr1 := addr(a) -4
5.      val1 := arr1[loc]
6.      arr2 := addr(b) -4
7.      val2 := arr2[loc]
8.      prod := val1 * val2
9.      sum := sum + prod
10.     i := i + 1;
11.     If i < = 20 goto 3
```

The above code is the equivalent machine code for the program to multiply two matrix, a[i] and b[i]. This machine takes 4 bytes for each word so that the location variable, loc, is multiplied by 4 every time. The variable, addr, gives the address of the array. 4 bytes are reduced from the address of the array to reach the initial address of the array. The optimization is done in step 10. Instead of writing temp := i + 1 and i := temp, the optimization is done by writing i := i + 1.

Code Generation

Code generation is the last phase in the compilation of the source code. The code generator generates the object code. It enables the intermediate calculation to be performed in registers. A simple three-address expression x := y + z is represented in object code as:

```
LOAD y
ADD z
STORE x
```

As a result, the generated object code contains a lot of redundancy. To reduce redundancy, the code generator keeps hold of the run-time variables that reside in the registers and effectively allocates the registers. The register allocation is the process of using the fast speed register in the optimal way. This involves storing only frequently used operands in the registers.

The code generator takes the input from the optimizer. The input provided is the optimized intermediate code, which has to be converted to the equivalent object code, as illustrated in Figure 1-1-12:

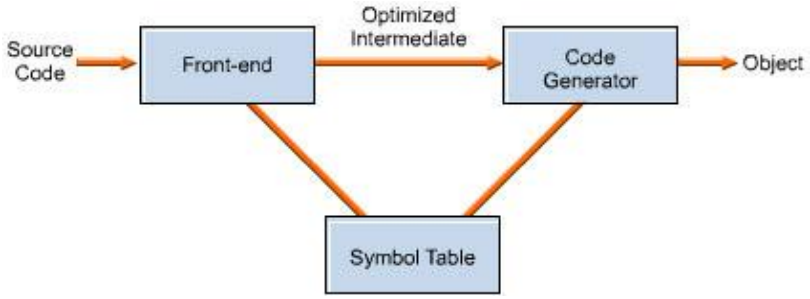

Figure 1-1-12: Code Generator

Code Generating System

Code generating system consists of memory with 232 bytes where the length of each word is 32 bits. Eight general-purpose registers, R0, R1...R7 are used, each of a word length. The format for writing expressions is:

```
OP src, dst
```

In the above expression, OP is the four-bit operator code and the source src and the destination dst are the six bit fields. The addressing modes used are:

- Register mode: Contain operands. The mnemonic code for register mode is r.
- Indirect Register mode: Contains the address of the operand, which resides in register mode. The mnemonic code for indirect register mode is * r.
- Indexed mode: Contains the value B, which resides in the word following the register instruction. The value B is added to the content of the register r to obtain the address of the operand. The mnemonic code for the indexed mode is B (r).
- Indirect Indexed mode: Contains the value B, which resides in the word following the register instruction. The value, B, is added to the content of the register r to obtain the address of the word that holds the address of the operand. The mnemonic code for indirect indexed mode is * B (r).
- Immediate mode: Contains the literal operand B, which resides in the word after the register instruction. The mnemonic code for immediate mode is # B.

The op-code used by the target system are:

- MOV: Moves the content of the source to the destination.
- SUB: Subtracts the content of the source from the destination.
- ADD: Adds the content of the source to the destination.

For example,

MOV R0, R1, copies the content of register 0 to register 1.

ADD #1, R2, adds constant 1 to the content of register 2.

SUB 3(R1), * 4(R2), subtracts ((R1) + 3) from (((R2) + 4)), where (B) specifies the content of register B.

Code Generating Issues

The issues that must be resolved before designing the code generator are memory management, instruction selection, allocation of registers, and input provided to the code generator.

The intermediate code acts as the input to the code generator. The input also contains references to the symbol table, which contains the run-time addresses of the variables used in the intermediate code. The code generation module assumes that the intermediate code provided by the intermediate code generator is error-free. As a result, no syntactical checking is done at the code-generating module.

The front-end and code generator collectively links the variables in source code to their addresses in run-time memory. The variables are declared in the symbol table depending on their storage space. The variable data type, which is mentioned in the source code, specifies the storage space needed by the variable in the symbol table.

For specific source code, the target system must support all the data types. If the target computer does not support all the data types, exception handling is needed to specify all the data types that are not supported by the target system. The instruction selection is also the key factor needed to create an efficient object code. The quality of generated object code is evaluated by its speed and its size. The target system with a large instruction set can present the same expression in a number of ways. For example, the increment instruction INC represents a := a + 1, as INC a.

The code generator uses registers for fast access of variables. The compiler stores the frequently accessed variables in registers. Certain target system requires the register-pairs to perform multiplication and division. The register-pair contains an even-numbered and an odd-numbered register. For example, multiplication is shown as:

```
mul a, b
```

In the above expression, the multiplicand a is the even register and multiplier b is the odd register. The result of the multiplication is stored in the complete register-pair.

An Example of Code Generator

A target generator is an instance of the code generator. A target generator develops the machine code for the stream of three-address expressions. Processor such as IntelTM Pentium with the word length of 32 bits is an example of a target generator. To develop a code generator, a machine code operator is available for each operator of the three-address expression. The calculated results are stored in registers and are shifted to memory if:

- The registers are needed for other intermediate calculations.
- The compiler scans any control statement, which requires jump to some other function or block of code.

The registers are emptied before jumping to other blocks of code. This is because when the control is transferred to some other function or block, the compiler has no way to determine which data resides in which register. To avoid occurrence of any error, all the registers are emptied before jumping to another block of code.

For a three-address statement, x := y + z, the optimal object code can be generated by using a single expression ADD Rm, Rn, where the calculated result is stored in Rn. This is possible if y resides in Rn, z resides in Rm, and the results are copied to Rn.

If the value z does not exist in the register but is stored in memory, the object code is generated as

```
ADD Z, Rn,
```

where Z is the memory location of the operand z.

Or

```
MOV Z, Rm
ADD Rm, Rn
```

The second example shows more optimal machine code because the value of the operand z is transferred from memory location Z to the register Rm, and the variable can be accessed faster from the registers. The code generator generates the machine code depending on the location and use of the operands.

Symbol Table

The compiler records all the variables, keywords, and identifiers that appear in the source code. The symbol table is the data structure that contains information about all these names that appear in the source code. The information that is stored in the symbol table is the name of the variable, the identifier, or the keyword, its data type, and the memory location where the name is stored.

Information related to any name that appears in the source code is entered in the symbol table during scanning and parsing. While reading the source code, the symbol table is scanned for each encountered name. If the symbol table does not have the entry of the name scanned, the entry is made in the symbol table. All the modules of the compiler structure use the symbol table. For example, the parser uses the symbol table to check whether the name used in source code is consistent with the declaration in the symbol table.

The symbol table is also used to store error messages. For example, if a compiler encounters an undefined variable, it generates an error message stating that the variable X is not defined. The symbol table will record the message and will not generate this message if the same variable is encountered again.

During scanning, all the attributes of the variable scanned are not detected. For example, in the Pascal declaration:

```
var a, b, c : int;
```

the data type of the variables a, b, and c is unknown to the scanner. When parser encounters the same variables, it checks for the data type of the variables. If the variables do not have an entry for the data types, the entry is made in the symbol table. At an abstract level, you can say that the symbol table is made of two fields, the name field and the information field. The information field contains information regarding the name scanned in the source code.

Symbol Table Contents

The symbol table capabilities, which are needed by the compiler, are:

- Capability to add a new name to the symbol table.
- Capability to add information about the name that already exists in the symbol table.
- Capability to check whether or not a name exists in the symbol table.
- Capability to retrieve information related to the specified name from the symbol table.

- Capability to remove the specified name from the symbol table.

The symbol table contains the names of different types of variables stored, such as operands, identifiers, and variables. The compiler creates multiple symbol tables to store names because the space needed by each name varies depending on the frequency of use of the name in the source code. If the HLL, in which the source code is written, does not reserve keywords, the keywords entry must be done in the symbol table.

The data structures used to implement the symbol table are the linear list, the hash table, and the tree structures. The choice of data structure depends on how effectively you can access the elements of the symbol table. The characteristics of different data structures that implement the symbol table are:

- The liner list is easy to implement but takes longer to be accessed.
- Hash table can be accessed fast but is not easy to implement.
- The tree structure provides the form that is intermediate to both the linear list and the hash table.

The information that can be added to the information field in the symbol table is:

- The stream of characters that specify the name used in the symbol table. If the same name is to be used in multiple blocks of code, the block to which the name belongs must be specified.
- Offset, which specifies the location of the name stored in the memory.
- Arguments, such as the limits of the array along with dimensions.

Some languages enable the identifier to be used as different types. For example, ABC can be a label, a field name, or a variable. If this is the case, the scanner returns the identifier to the parser rather than to the pointer of the symbol table.

Related Topics

For related information on this topic, you can refer to:

- Principles of Compiler Design
- Introducing Optimization in Compiler Design
- Parsing Techniques