Compiler Syllabus

Introduction of Compilers and languages
Compilers Phases
Symbol table
Type of Symbol table
Lexical analysis
Regular expressions& Finite state automata
Finite state automata: Nondeterministic and deterministic finite automata
Syntax analysis
Top-down parsing: Introduction & Eliminating left recursion in a grammar
Predictive parsers: LL(1) grammars& Construction of first and follow.
Bottom-up parsing: Shift-reduce parsers& SLR(1) parsing
CLR(1) parsing& LALR parsers.
Semantic analysis and Intermediate code generation

References

- Aho, Lam, Sethi, Ullman: Compilers: Principles, Techniques, and Tools (2nd Edition).
- Garenra Sharma, Compiler Design.

1.1 Introduction

Compiler: a compiler is a program that can read a program in one language - the source language - and translate it into an equivalent program in another language - the target language; see Fig. 1.1. An important role of the compiler is to report any errors in the source program that it detects during the translation process.

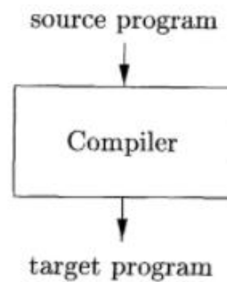


Figure 1.1: A compiler

An important part of any compiler is the detection and reporting of errors. Commonly, the source language is a high-level programming language (i.e. a problem-oriented language), and the target language is a machine language or assembly language (i.e. a machine-oriented language). Thus compilation is a fundamental concept in the production of software: it is the link between the (abstract) world of application development and the low-level world of application execution on machines.

A compiler can be treated as a single box that maps a source program into a semantically equivalent target program. If we open up this box a little, we see that there are two parts to this mapping: **analysis** and **synthesis**.

- The analysis part breaks up the source program into constituent pieces and imposes a grammatical structure on them. It then uses this structure to create an intermediate representation of the source program. If the analysis part detects that the source program is either syntactically ill formed or semantically unsound, then it must provide informative messages, so the user can take corrective action. The analysis part also collects information about the source program and stores it in a data structure called a symbol table, which is passed along with the intermediate representation to the synthesis part.

- The synthesis part constructs the desired target program from the intermediate representation and the information in the symbol table. The analysis part is often called the front end of the compiler; the synthesis part is the back end.

1.2 The Structure of Compiler

The compilation procedure is nothing but a series of different phases. Each stage acquires input from its previous phase. A typical decomposition of a compiler into phases is shown in Fig. 1.2. In practice, several phases may be grouped together, and the intermediate representations between the grouped phases need not be constructed explicitly. The symbol table, which stores information about the entire source program, is used by all phases of the compiler.

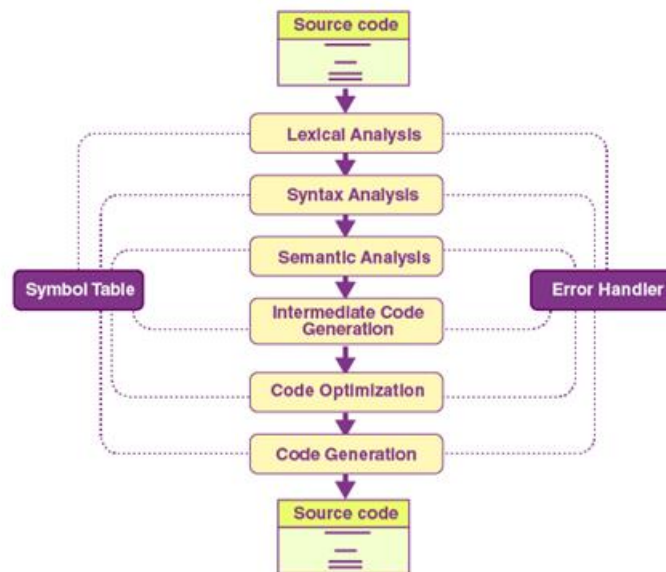


Figure 1.2 Compiler Phases

1.2.1 Lexical analysis

Lexical Analysis is the first phase when compiler scans the source code. This process can be left to right, character by character, and group these characters into tokens. Here, the character stream from the source program is grouped in meaningful sequences by identifying the tokens. It makes the entry of the corresponding tickets into the symbol table and passes that token to next phase. The primary functions of this phase are:

- Identify the lexical units in a source code

- Classify lexical units into classes like constants, reserved words, and enter them in different tables. It will Ignore comments in the source program
- Identify token which is not a part of the language

Example:

x = y + 10

Tokens

X	identifier
=	Assignment operator
Y	identifier
+	Addition operator
10	Number

1.2.2 Syntax Analysis

Syntax analysis is all about discovering structure in code. It determines whether or not a text follows the expected format. The main aim of this phase is to make sure that the source code was written by the programmer is correct or not. Syntax analysis is based on the rules based on the specific programing language by constructing the parse tree with the help of tokens. It also determines the structure of source language and grammar or syntax of the language. Here, is a list of tasks performed in this phase:

- Obtain tokens from the lexical analyzer
- Checks if the expression is syntactically correct or not
- Report all syntax errors
- Construct a hierarchical structure which is known as a parse tree

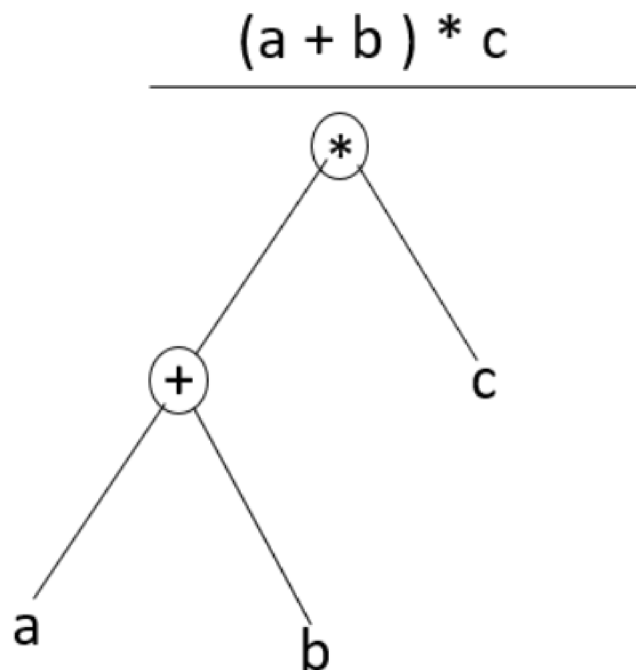
Example

Any identifier/number is an expression

If x is an identifier and y+10 is an expression, then x= y+10 is a statement.

Consider parse tree for the following example

```
(a+b)*c
```



1.2.3 Semantic Analysis

Semantic analysis checks the semantic consistency of the code. It uses the syntax tree of the previous phase along with the symbol table to verify that the given source code is semantically consistent. It also checks whether the code is conveying an appropriate meaning. Semantic Analyzer will check for Type mismatches, incompatible operands, a function called with improper arguments, an undeclared variable, etc. An important part of semantic analysis is type checking, where the compiler checks that each operator has matching operands. For example, many programming language definitions require an array index to be an integer; the

compiler must report an error if a floating-point number is used to index an array. Functions of Semantic analyses phase are:

- Helps you to store type information gathered and save it in symbol table or syntax tree
- Allows you to perform type checking
- In the case of type mismatch, where there are no exact type correction rules which satisfy the desired operation a semantic error is shown
- Collects type information and checks for type compatibility
- Checks if the source language permits the operands or not

Example

```
float x = 20.2;
float y = x*30;
```

In the above code, the semantic analyzer will typecast the integer 30 to float 30.0 before multiplication

1.2.4 Intermediate Code Generation

Once the semantic analysis phase is over the compiler, generates intermediate code for the target machine. It represents a program for some abstract machine. Intermediate code is between the high-level and machine level language. This intermediate code needs to be generated in such a manner that makes it easy to translate it into the target machine code.
Functions on Intermediate Code generation:

- It should be generated from the semantic representation of the source program
- Holds the values computed during the process of translation
- Helps you to translate the intermediate code into target language
- Allows you to maintain precedence ordering of the source language
- It holds the correct number of operands of the instruction

Example:

p=i*r*t;

temp1=id3*id4

temp2=id2*temp1

id1=temp2

1.2.5 Code Optimization

The next phase of is code optimization or Intermediate code. This phase removes unnecessary code line and arranges the sequence of statements to speed up the execution of the program without wasting resources. The main goal of this phase is to improve on the intermediate code to generate a code that runs faster and occupies less-space.

The primary functions of this phase are:

- It helps you to establish a trade-off between execution and compilation speed
- Improves the running time of the target program
- Generates streamlined code still in intermediate representation
- Removing unreachable code and getting rid of unused variables
- Removing statements which are not altered from the loop

Example:

Consider the following code

```
a = into float (10)
b = c * a
d = e + b
f = d
```

Can become

```
b =c * 10.0
f = e+b
```

1.2.6 Code Generation

Code generation is the last and final phase of a compiler. It gets inputs from code optimization phases and produces the page code or object code as a result. The objective of this phase is to allocate storage and generate relocatable machine code. It also allocates memory locations for the variable. The instructions in the intermediate code are converted into machine instructions. This phase coverts the optimize or intermediate code into the target language. The target language is the machine code. Therefore, all the memory locations and registers are also selected and allotted during this phase. The code generated by this phase is executed to take inputs and generate expected outputs.

Example:

P=i*r*t

MOVE id4,R2

MULT id3,R2

MOVE id2,R1

MULT R2,R1

MOVE R1,id1