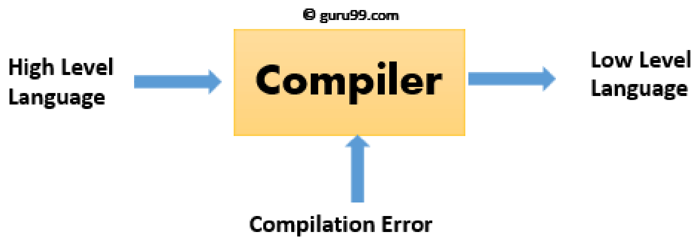
CSC 335 Notes 3

What is a Compiler Design? Types, Construction Tools, Example

What is a Compiler?

A compiler is a computer program which helps you transform source code written in a high-level language into low-level machine language. It translates the code written in one programming language to some other language without changing the meaning of the code. The compiler also makes the end code efficient, which is optimized for execution time and memory space.

The compiling process includes basic translation mechanisms and error detection. The compiler process goes through lexical, syntax, and semantic analysis at the front end and code generation and optimization at the back-end.



Features of Compilers

- Correctness
- Speed of compilation
- Preserve the correct the meaning of the code
- The speed of the target code
- Recognize legal and illegal program constructs
- Good error reporting/handling
- Code debugging help

Types of Compiler

Following are the different types of Compiler:

- Single Pass Compilers
- Two Pass Compilers
- Multipass Compilers

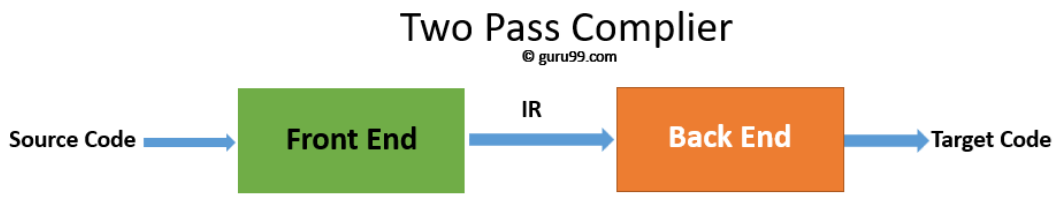
Single Pass Compiler

Single Pass Complier
© guru99.com
Source Code
Compiler
Target Code

Single Pass Compiler

In single pass Compiler source code directly transforms into machine code. For example, Pascal language.

Two Pass Compiler



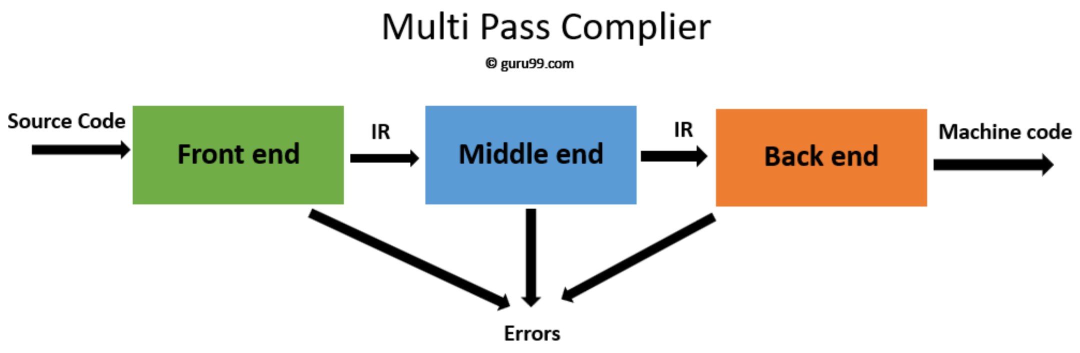
Two Pass Compiler

Two pass Compiler is divided into two sections, viz.

1. **Front end:** It maps legal code into Intermediate Representation (IR).
2. **Back end:** It maps IR onto the target machine

The Two pass compiler method also simplifies the retargeting process. It also allows multiple front ends.

Multipass Compilers



Multipass Compilers

The multipass compiler processes the source code or syntax tree of a program several times. It divided a large program into multiple small programs and process them. It develops multiple intermediate codes. All of these multipass take the output of the previous phase as an input. So it requires less memory. It is also known as 'Wide Compiler'.

Tasks of Compiler

The main tasks performed by the Compiler are:

- Breaks up the up the source program into pieces and impose grammatical structure on them
- Allows you to construct the desired target program from the intermediate representation and also create the symbol table
- Compiles source code and detects errors in it
- Manage storage of all variables and codes.
- Support for separate compilation
- Read, analyze the entire program, and translate to semantically equivalent
- Translating the source code into object code depending upon the type of machine

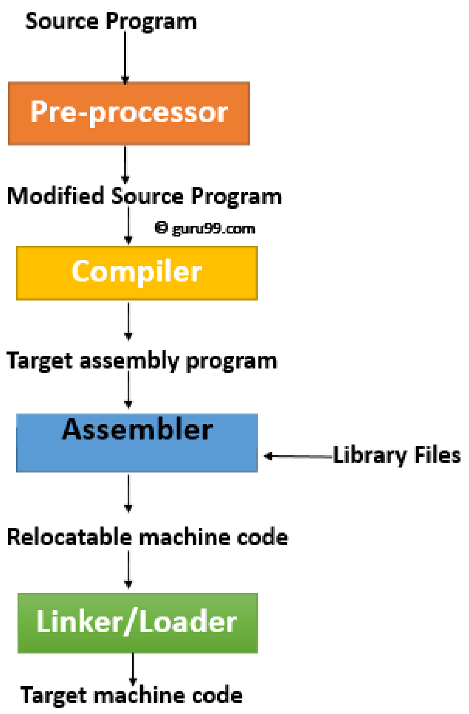
History of Compiler

Important Landmark of Compiler's history is as follows:

- The "compiler" word was first used in the early 1950s by Grace Murray Hopper.
- The first compiler was build by John Backum and his group between 1954 and 1957 at IBM.
- COBOL was the first programming language which was compiled on multiple platforms in 1960
- The study of the scanning and parsing issues were pursued in the 1960s and 1970s to provide a complete solution.

Steps for Language processing systems

Before knowing about the concept of compilers, you first need to understand a few other tools which work with compilers.



Steps for Language processing systems

- **Preprocessor**: The preprocessor is considered as a part of the Compiler. It is a tool which produces input for Compiler. It deals with macro processing, augmentation, language extension, etc.
- **Interpreter**: An interpreter is like Compiler which translates high-level language into low-level machine language. The main difference between both is that interpreter reads and transforms code line by line. Compiler reads the entire code at once and creates the machine code.
- **Assembler**: It translates assembly language code into machine understandable language. The output result of assembler is known as an object file which is a combination of machine instruction as well as the data required to store these instructions in memory.
- **Linker**: The linker helps you to link and merge various object files to create an executable file. All these files might have been compiled with separate assemblers. The main task of a linker is to search for called modules in a program and to find out the memory location where all modules are stored.
- **Loader**: The loader is a part of the OS, which performs the tasks of loading executable files into memory and run them. It also calculates the size of a program which creates additional memory space.
- **Cross-compiler**: A Cross compiler in compiler design is a platform which helps you to generate executable code.
- **Source-to-source Compiler**: Source to source compiler is a term used when the source code of one programming language is translated into the source of another language.

Compiler Construction Tools

Compiler construction tools were introduced as computer-related technologies spread all over the world. They are also known as a compiler- compilers, compiler- generators or translator.

These tools use specific language or algorithm for specifying and implementing the component of the compiler. Following are the example of compiler construction tools.

- **Scanner generators**: This tool takes regular expressions as input. For example LEX for Unix Operating System.
- **Syntax-directed translation engines**: These software tools offer an intermediate code by using the parse tree. It has a goal of associating one or more translations with each node of the parse tree.
- **Parser generators:** A parser generator takes a grammar as input and automatically generates source code which can parse streams of characters with the help of a grammar.
- **Automatic code generators**: Takes intermediate code and converts them into Machine Language.
- **Data-flow engines**: This tool is helpful for code optimization. Here, information is supplied by the user, and intermediate code is compared to analyze any relation. It is also known as data-flow analysis. It helps you to find out how values are transmitted from one part of the program to another part.

Why use a Compiler?

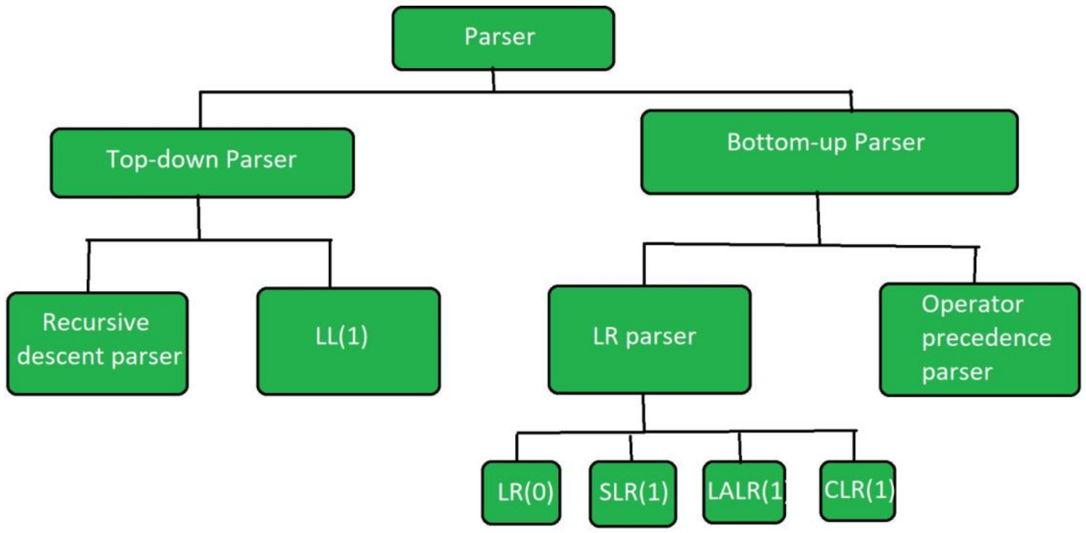
- Compiler verifies entire program, so there are no syntax or semantic errors.
- The executable file is optimized by the compiler, so it is executes faster.
- Allows you to create internal structure in memory.
- There is no need to execute the program on the same machine it was built.
- Translate entire program in other language.
- Generate files on disk.
- Link the files into an executable format.
- Check for syntax errors and data types.
- Helps you to enhance your understanding of language semantics.
- Helps to handle language performance issues.
- Opportunity for a non-trivial programming project.
- The techniques used for constructing a compiler can be useful for other purposes as well.

Application of Compilers

- Compiler design helps full implementation Of High-Level Programming Languages.
- Support optimization for Computer Architecture Parallelism.
- Design of New Memory Hierarchies of Machines.
- Widely used for Translating Programs.
- Used with other Software Productivity Tools.

Types of Parsers in Compiler Design

The **parser** is that phase of the compiler which takes a token string as input and with the help of existing grammar, converts it into the corresponding Intermediate Representation. The parser is also known as *Syntax Analyzer*.



Types of Parser:

The parser is mainly classified into two categories, i.e. Top-down Parser, and Bottom-up Parser. These are explained below:

Top-Down Parser:

The top-down parser is the parser that **generates parse for the given input string** with the help of grammar productions by expanding the non-terminals i.e. it starts from the start symbol and ends on the terminals. It uses left most derivation.
Further Top-down parser is classified into 2 types: Recursive descent parser, and Non-recursive descent parser.

1. **Recursive descent parser** is also known as the Brute force parser or the backtracking parser. It basically generates the parse tree by using brute force and backtracking.
2. **Non-recursive descent parser** is also known as LL(1) parser or predictive parser or without backtracking parser or dynamic parser. It uses a parsing table to generate the parse tree instead of backtracking.

Bottom-up Parser:

Bottom-up Parser is the parser that generates the parse tree for the given input string with the help of grammar productions by compressing the non-terminals i.e. it starts from non-terminals and ends on the start symbol. It uses the reverse of the rightmost derivation.
Further Bottom-up parser is classified into two types: LR parser, and Operator precedence parser.

- **LR parser** is the bottom-up parser that generates the parse tree for the given string by using unambiguous grammar. It follows the reverse of the rightmost derivation.
  LR parser is of four types:

(a)LR(0)
(b)SLR(1)
(c)LALR(1)
(d)CLR(1)

- **Operator precedence parser** generates the parse tree form given grammar and string but the only condition is two consecutive non-terminals and epsilon never appear on the right-hand side of any production.

Example

Consider the following grammar-

$$S \rightarrow ABC$$

$$A \rightarrow a$$

$$B \rightarrow b$$

$$C \rightarrow c$$

Consider a string w = abc.

Derivation-01:

S → ABC

→ aBC (Using A → a)

→ aBc (Using C → c)

→ abc (Using B → b)

Derivation-02:

S → ABC

→ AbC (Using B → b)

→ abC (Using A → a)

→ abc (Using C → c)

Derivation-03:

S → ABC

→ AbC (Using B → b)

→ Abc (Using C → c)

→ abc (Using A → a)

The other 2 derivations are leftmost derivation and rightmost derivation.

Example

Consider the following grammar-

$$S \rightarrow aS / \in$$

The language generated by this grammar is-

$$L = \{ a^n , n>=0 \} \text{ or } a^*$$

All the strings generated from this grammar have their leftmost derivation and rightmost derivation exactly same.

Let us consider a string w = aaa.

Leftmost Derivation-

S → a**S**

→ aa**S** (Using S → aS)

→ aaa**S** (Using S → aS)

→ aaa∈

→ aaa

Rightmost Derivation-

S → a**S**

→ aa**S** (Using S → aS)

→ aaa**S** (Using S → aS)

→ aaa∈

→ aaa

Clearly,

Leftmost derivation = Rightmost derivation

Similar is the case for all other strings.

Point-06:

- For a given parse tree, we may have its leftmost derivation exactly same as rightmost derivation.

Point-07:

- If for all the strings of a grammar, leftmost derivation is exactly same as rightmost derivation, then that grammar may be ambiguous or unambiguous.

Example

Consider the following grammar-

$$S \rightarrow aS \,/\, \in$$

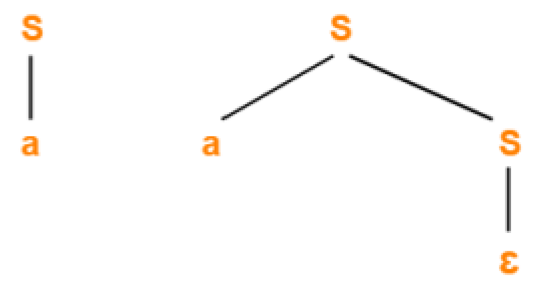
- This is an example of an unambiguous grammar.
- Here, each string have its leftmost derivation and rightmost derivation exactly same.

Now, consider the following grammar-

$$S \rightarrow aS \,/\, a \,/\, \in$$

- This is an example of ambiguous grammar.
- Here also, each string have its leftmost derivation and rightmost derivation exactly same.

Consider a string w = a.



Parse Tree-01 Parse Tree-02

Since two different parse trees exist, so grammar is ambiguous.

Leftmost derivation and rightmost derivation for parse tree-01 are-

$$S \rightarrow a$$

Leftmost derivation and rightmost derivation for parse tree-02 are-

$$S \rightarrow aS$$

$$S \rightarrow a\in$$

$$S \rightarrow a$$

Parse Tree | Derivations | Automata

Parse Tree-

- The process of deriving a string is called **derivation**.
- The geometrical representation of a derivation is called as a **parse tree** or **derivation tree.**

1. Leftmost Derivation-

- The process of deriving a string by expanding the leftmost non-terminal at each step is called **leftmost derivation**.
- The geometrical representation of leftmost derivation is called a **leftmost derivation tree**.

Example-

Consider the following grammar-

S → aB / bA

S → aS / bAA / a

B → bS / aBB / b

(**Unambiguous Grammar**)

Let us consider a string w = aaabbabbba

Now, let us derive the string w using leftmost derivation.

Leftmost Derivation-

S → a**B**

→ aa**B**B (Using B → aBB)

→ aaa**B**BB (Using B → aBB)

→ aaab**B**B (Using B → b)

→ aaabb**B** (Using B → b)

→ aaabba**B**B (Using B → aBB)

→ aaabbab**B** (Using B → b)

→ aaabbabb**S** (Using B → bS)

→ aaabbabbb**A** (Using S → bA)

→ aaabbabbba (Using A → a)

2. Rightmost Derivation-

- The process of deriving a string by expanding the rightmost non-terminal at each step is called **rightmost derivation**.
- The geometrical representation of rightmost derivation is called a **rightmost derivation tree**.

Example-

Consider the following grammar-

S → aB / bA

S → aS / bAA / a

B → bS / aBB / b

(**Unambiguous Grammar**)

Let us consider a string w = aaabbabbba

Now, let us derive the string w using rightmost derivation.

Rightmost Derivation-

S → a**B**

→ aaB**B** (Using B → aBB)

→ aaBaB**B** (Using B → aBB)

→ aaBaBb**S** (Using B → bS)

→ aaBaBbb**A** (Using S → bA)

→ aaBa**B**bba (Using A → a)

→ aa**B**abbba (Using B → b)

→ aaaB**B**abbba (Using B → aBB)

→ aaa**B**babbba (Using B → b)

→ aaabbabbba (Using B → b)

NOTES

- For unambiguous grammars, Leftmost derivation and Rightmost derivation represents the same parse tree.
- For ambiguous grammars, Leftmost derivation and Rightmost derivation represents different parse trees.

Here,

- The given grammar was unambiguous.
- That is why, leftmost derivation and rightmost derivation represents the same parse tree.

Leftmost Derivation Tree = Rightmost Derivation Tree

Properties Of Parse Tree-

- Root node of a parse tree is the start symbol of the grammar.
- Each leaf node of a parse tree represents a terminal symbol.
- Each interior node of a parse tree represents a non-terminal symbol.
- Parse tree is independent of the order in which the productions are used during derivations.

Yield Of Parse Tree-

- Concatenating the leaves of a parse tree from the left produces a string of terminals.
- This string of terminals is called as **yield of a parse tree**.

PRACTICE PROBLEMS BASED ON DERIVATIONS AND PARSE TREE-

Problem-01:

Consider the grammar-

S → bB / aA

A → b / bS / aAA

B → a / aS / bBB

For the string w = bbaababa, find-

1. Leftmost derivation
2. Rightmost derivation
3. Parse Tree

Solution-

1. Leftmost Derivation-

S → b**B**

→ bb**B**B (Using B → bBB)

→ bba**B** (Using B → a)

→ bbaa**S** (Using B → aS)

→ bbaab**B** (Using S → bB)

→ bbaaba**S** (Using B → aS)

→ bbaabab**B** (Using S → bB)

→ bbaababa (Using B → a)

2. Rightmost Derivation-

S → b**B**

→ bbB**B** (Using B → bBB)

→ bbBa**S** (Using B → aS)

→ bbBab**B** (Using S → bB)

→ bbBaba**S** (Using B → aS)

→ bbBabab**B** (Using S → bB)

→ bb**B**ababa (Using B → a)

→ bbaababa (Using B → a)

3. Parse Tree-

- Whether we consider the leftmost derivation or rightmost derivation, we get the above parse tree.
- The reason is given grammar is unambiguous.

Problem-02:

Consider the grammar-

S → A1B

A → 0A / ∈

B → 0B / 1B / ∈

For the string w = 00101, find-

1. Leftmost derivation
2. Rightmost derivation
3. Parse Tree

Solution-

1. Leftmost Derivation-

S → **A**1B

→ 0**A**1B (Using A → 0A)

→ 00**A**1B (Using A → 0A)

→ 001**B** (Using A → ∈)

→ 0010**B** (Using B → 0B)

→ 00101**B** (Using B → 1B)

→ 00101 (Using B → ∈)

2. Rightmost Derivation-

S → A1**B**

→ A10**B** (Using B → 0B)

→ A101**B** (Using B → 1B)

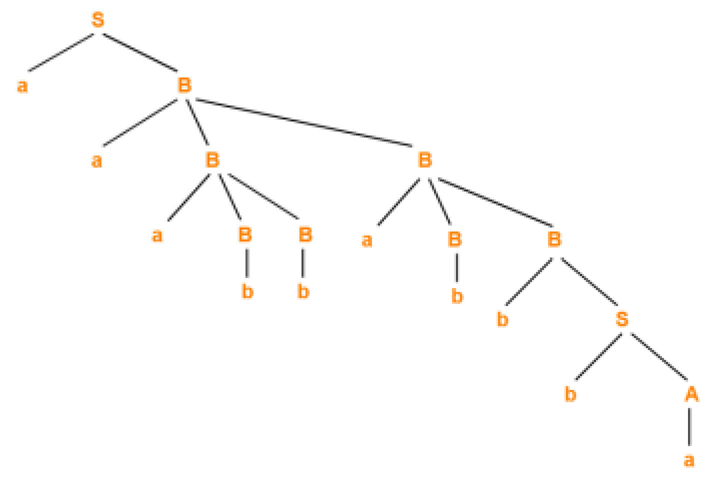
→ **A**101 (Using B → ∈)

→ 0**A**101 (Using A → 0A)

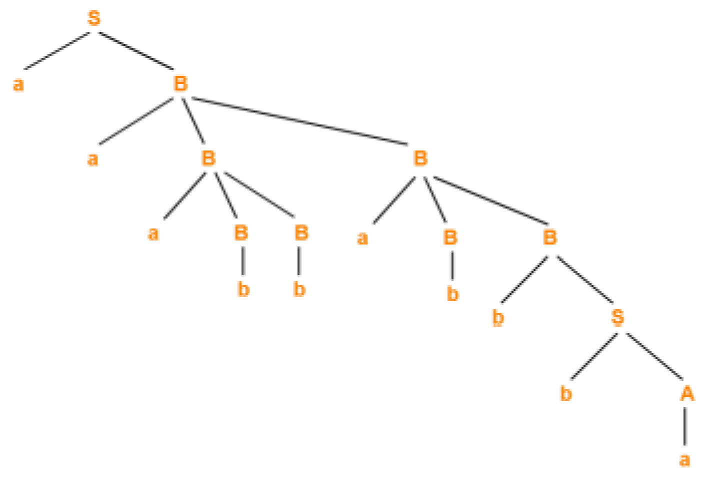
→ 00**A**101 (Using A → 0A)

→ 00101 (Using A → ∈)

3. Parse Tree-



Leftmost Derivation Tree



Rightmost Derivation Tree

- Whether we consider the leftmost derivation or rightmost derivation, we get the above parse tree.
- The reason is given grammar is unambiguous.