1 Introduction to Compiler Design

Language processing system:

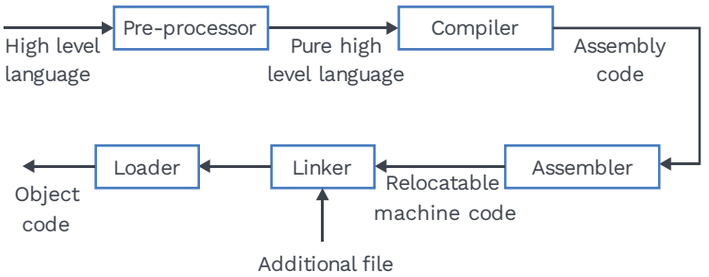


Fig. 1.1 Language Processing System

- The pre-processor is the first program to get invoked.
- The main function of a pre-processor is to substitute the macros with the exact value.
- The compiler is the next code to run. This program accepts a preprocessed file as input and outputs assembly language.
- The assembler command is used to translate the assembly file into a relocatable object file in the third stage. The relocatable object file is not in a format that can be read by humans. (It comes in ELF (executable and linking format) and COFF formats) (common object file format).
- The final step is the invocation of the linker and loader to generate the executable file. The linker utility links the relocatable object file with the system-wide startup objects file and makes it executable.

Compiler and translator:

A translator is a program that takes a program as input written in one programming language (the source language) and produce a program as output in another language (The object or target language).

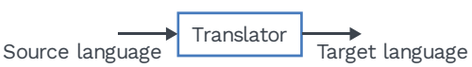


Fig. 1.2 Translator

If the source language is a high-level language and the object language is a low-level language such as assembly language, Then such translator is called compiler.

Fig. 1.3 Compiler

Compilation and execution:

A program written in a high-level language is usually executed in two steps.

Step 1:
First, the source program must be compiled, or converted into the object program.

Fig. 1.4 Compiler

Step 2:
The generated object program is loaded into memory and executed.

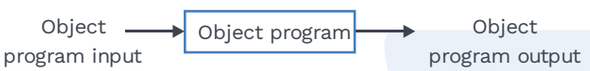


Fig. 1.5 Object Program

Note

- The translator is known as an assembler if the source language is assembly, and the target language is machine language.
- Pre-processors are sometimes used for translators that take programs in one high-level language into an equivalent programs in another high-level language.

Rack Your Brain

Why do we need translators?

Phases of a compiler:
A phase is a logically consistent procedure that accepts one representation of a source programme as input and outputs another one.

- Lexical analysis, often known as the scanner divides characters from the source language into tokens, or groupings of characters that logically belong together.
- The tokens are the basic unit of programming language, which can not be broken up further.
- Syntax analysis phase takes tokens as an input and produce parse tree as output.

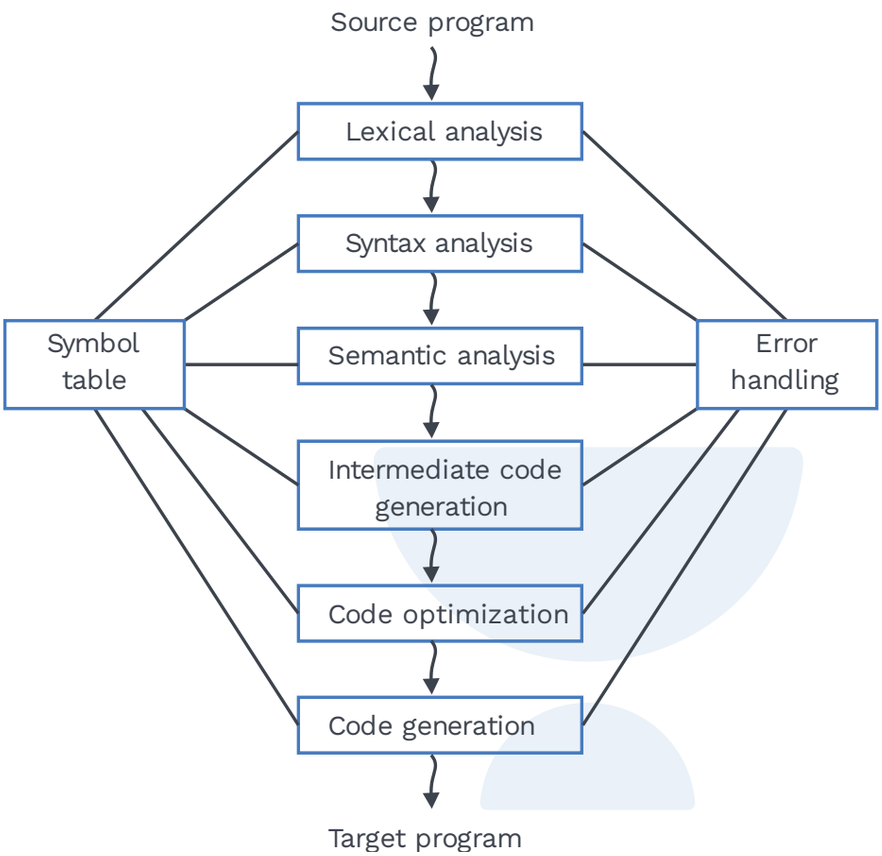


Fig. 1.6 Flow of a Program Execution

- Semantic analysis phase takes the parse tree as input and produces annotated parse tree (decorated parse tree/ semantically verified parse tree) as output.
- The intermediate code generator creates a stream of simple instructions using the structure generated by the semantic analyzer. (for example 3-address code).
- Code optimization is an optional phase that aims to enhance intermediate code in order to make the final object program run quicker and/or take-up less space.
- Code generation generates object code by deciding on data memory locations.
- The symbol table is a data structure that keeps track of the names used by the program and records important information about each one, such as its type (integer, real, etc.).
- When an error in the source program is discovered, the error handler is invoked.

Note

- One of the most difficult aspects of compiler design, both practically and theoretically, is creating a C code generator that produces truly efficient object program.
- Symbol table and error handling routines interact with all phases of the compiler

Analysis and synthesis phase:

- There are two major parts of the compiler:
  1) Analysis phase
  2) Synthesis phase
- Analysis phase contains a lexical analyzer, syntax analyzer and semantic analyzer.
- Synthesis phase contains an Intermediate code generator code optimiser and target code generator.

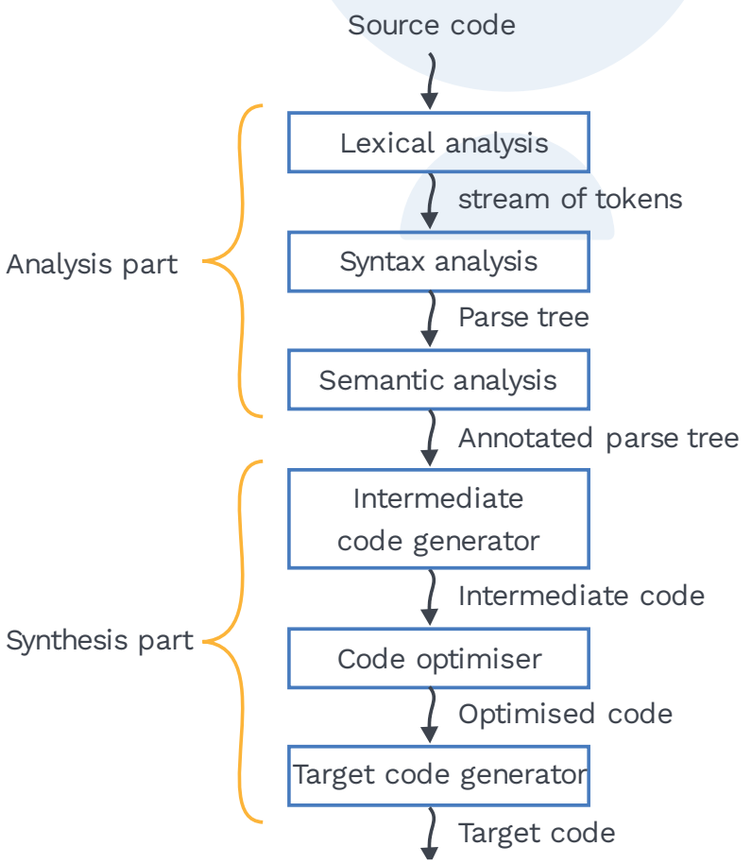


Fig. 1.7 Parts of Flow of Program Execution

Front end and back end of a compiler:

- The front end is divided into two phases: one that is dependent on the input (source language) and the other that is independent of the destination machine (target language).
- The phases of the compiler's front end are as follows.

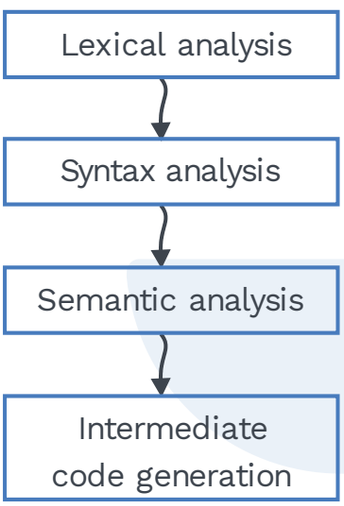


Fig. 1.8 Analysis Part

- Back end is subdivided into phases of a compiler that depend on the target machine and Independent of the source language.
- The following phases constitute the compiler's back end.

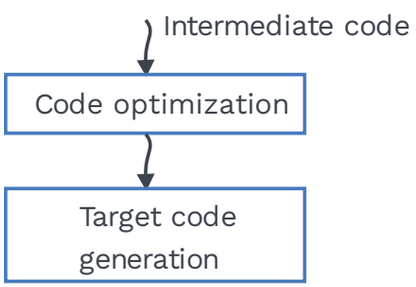
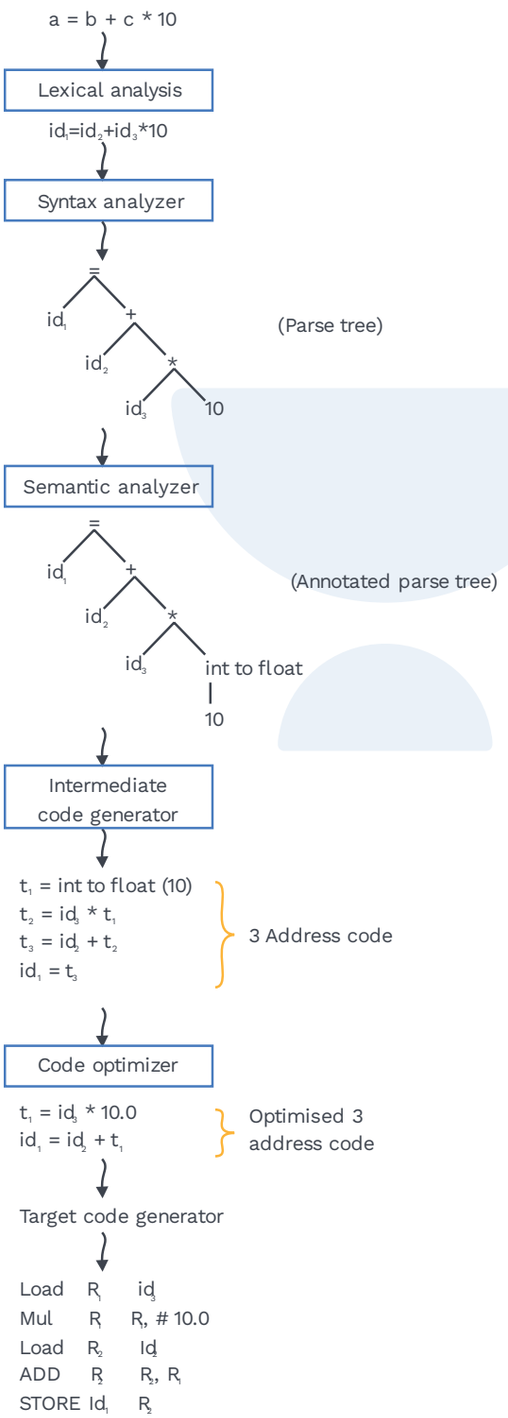


Fig. 1.9 Synthesis Part

SOLVED EXAMPLES

Q1 **Consider the C statement a = b + c * 10, and write the output of each phase after compilation.**

Sol:


a = b + c * 10
Lexical analysis
id1=id2+id3*10
Syntax analyzer
=
id1
+
id2
*
id3
10
(Parse tree)
Semantic analyzer
=
id1
+
id2
*
id3
int to float
10
(Annotated parse tree)
Intermediate code generator
t1 = int to float (10)
t2 = id3 * t1
t3 = id2 + t2
id1 = t3
3 Address code
Code optimizer
t1 = id3 * 10.0
id1 = id2 + t1
Optimised 3 address code
Target code generator
Load R1 id3
Mul R1 R1, # 10.0
Load R2 Id2
ADD R2 R2, R1
STORE Id1 R2

Passes:

- A pass is a module that contains sections of one or more phases of a compiler implementation.
- A pass takes the source programme or the output of the previous pass, executes the transformation defined by its phases, and writes the output into an intermediate file that can be read by the next pass.
- If all phases are grouped into one single module is called single-pass compiler.
- A multipass compiler processes the source code of a program multiple times.
- In a multipass compiler, we divide phases into two passes. The front end is considered as the first pass, while the back end is considered as the second pass.
- Multipass compiler, also known as Two-pass compiler.
- A multipass compiler can be developed to take up less space than a single pass compiler since the space taken by the compiler software for one pass can be reused by the following pass.
- Because each pass reads and writes an Intermediate file, a multipass compiler is slower than a single pass compiler.

Data structures in a compiler:

Compiler phases communicate with one another using a shared data structure. These are two crucial tables that are used during the compilation process. They are as follows:

1) Symbol table
2) Literal table

- A symbol table is a table that contains information about the identifiers that were used in the input source program. The strings and constants encountered in the input source program are stored in a literal table.
- The literal table's principal purpose is to save memory by reusing constants and strings.

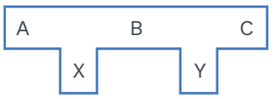
Bootstrapping:

- The source language, the object language, and the language in which it is written are the three languages that define a compiler.
- A cross compiler is a compiler that can run on one machine and create object code for another machine.

Example:

1) Compiler runs on machine X, which takes input language A and produce output language B.

2) Compiler runs on machine X and produces code B for another machine Y.



Chapter Summary

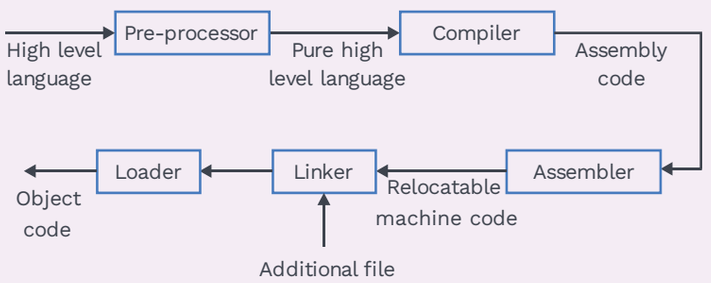


Fig. 1.10 Flow of Program

- Translator translates the source language into the target language.
- Compiler converts high–level language to low–level language.
- Translator is also known as an assembler.
- **Phases of the compiler:**

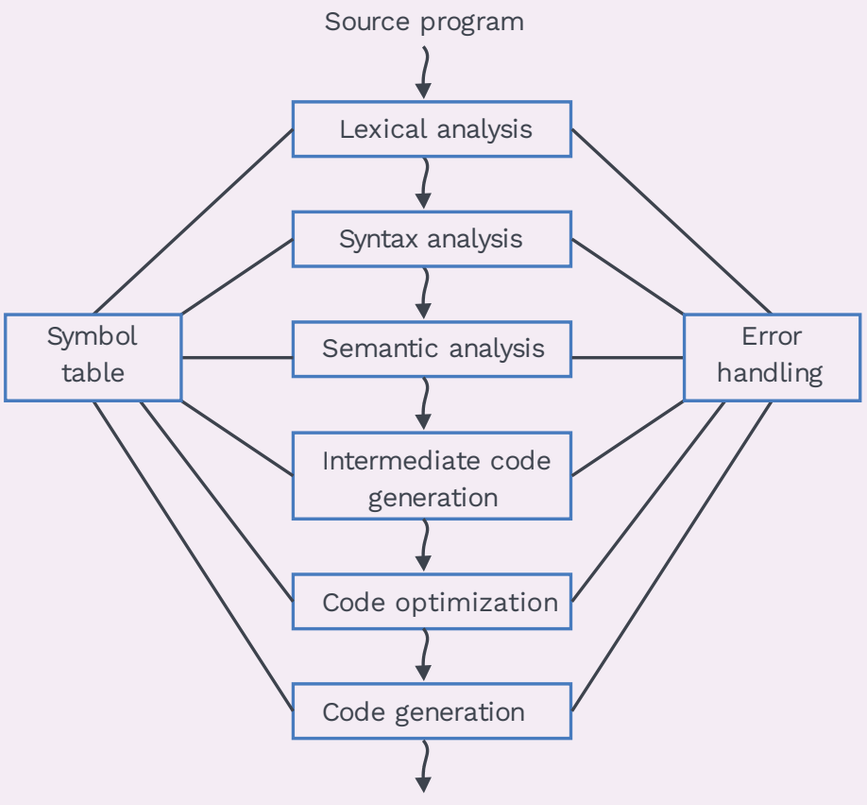


Fig. 1.11 Flow of Program Execution

- Data structures used in the compiler:
- Symbol table
- Literal table