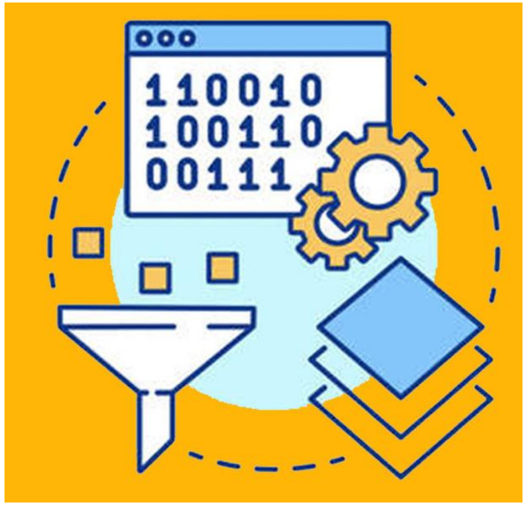
Compiler Design

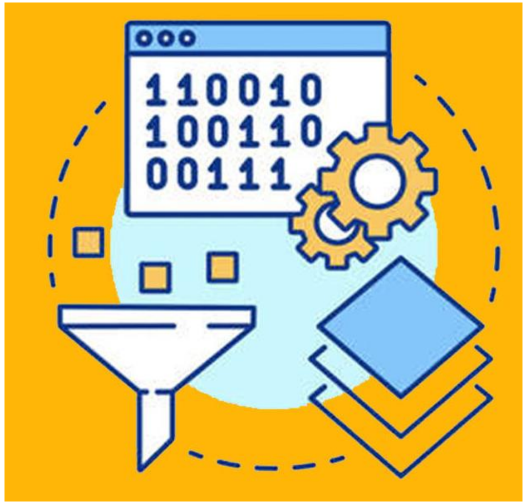


Introduction to Compiler

Definition of Compiler

Phases & Groups of a Compiler

Compiler Construction Tools



Prof. S. D. PADIYA| SSGMCE, Shegaon
padiyasagar@gmail.com

2021-2022 (Spring)

Course Objectives

Throughout the course, you will be expected to demonstrate their understanding of **Compiler Design** by being able to do each of the following:

1. To learn concepts of programming language translation and phases of compiler design.
2. To understand the common forms of parsers.
3. To study concept of syntax directed definition and translation scheme for the representation of language.
4. To illustrate the various optimization techniques for designing various optimizing compilers.

Course Outcomes

On completion of the course, the you will be able to:

1. Describe the fundamentals of compiler and various phases of compilers.
2. Design and implement LL (Left-to-right, Leftmost derivation in reverse) and LR (Left-to-right, Rightmost derivation in reverse) parsers.
3. Solve the various parsing techniques like SLR (Simple LR), CLR (Canonical collection of LR), LALR (Look-Ahead LR).
4. Examine the concept of Syntax-Directed Definition and translation.
5. Assess the concept of Intermediate Code Generation and run-time environment.
6. Explain the concept code generation and code optimization.

Syllabus: Unit 01

Introduction to Compiling:

- Definition of Compiler
- Phases of a Compiler and Grouping of Phases
- Compiler Construction Tools

Lexical Analysis:

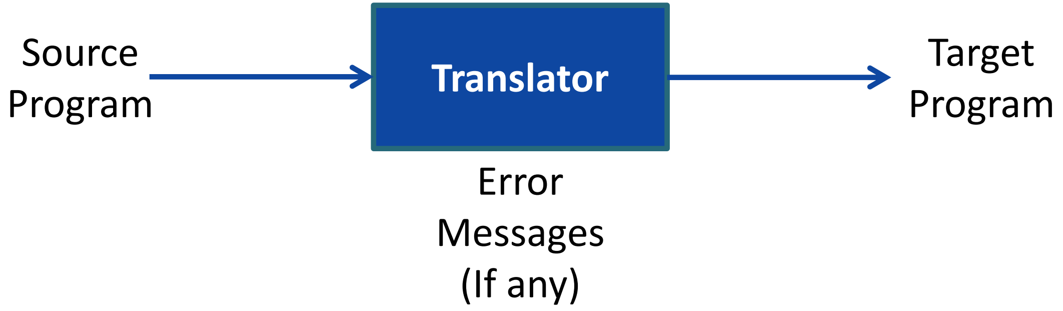
- The role of lexical analyzer
- Input buffering
- Specification of tokens
- Recognition of tokens
- Language for specifying lexical analysis
- LEX and YACC tools
- Finite automata from regular expressions to finite automata and state minimization of DFA.

Translator

A translator is a program that **takes one form of the program as input** and **converts it into another form.**

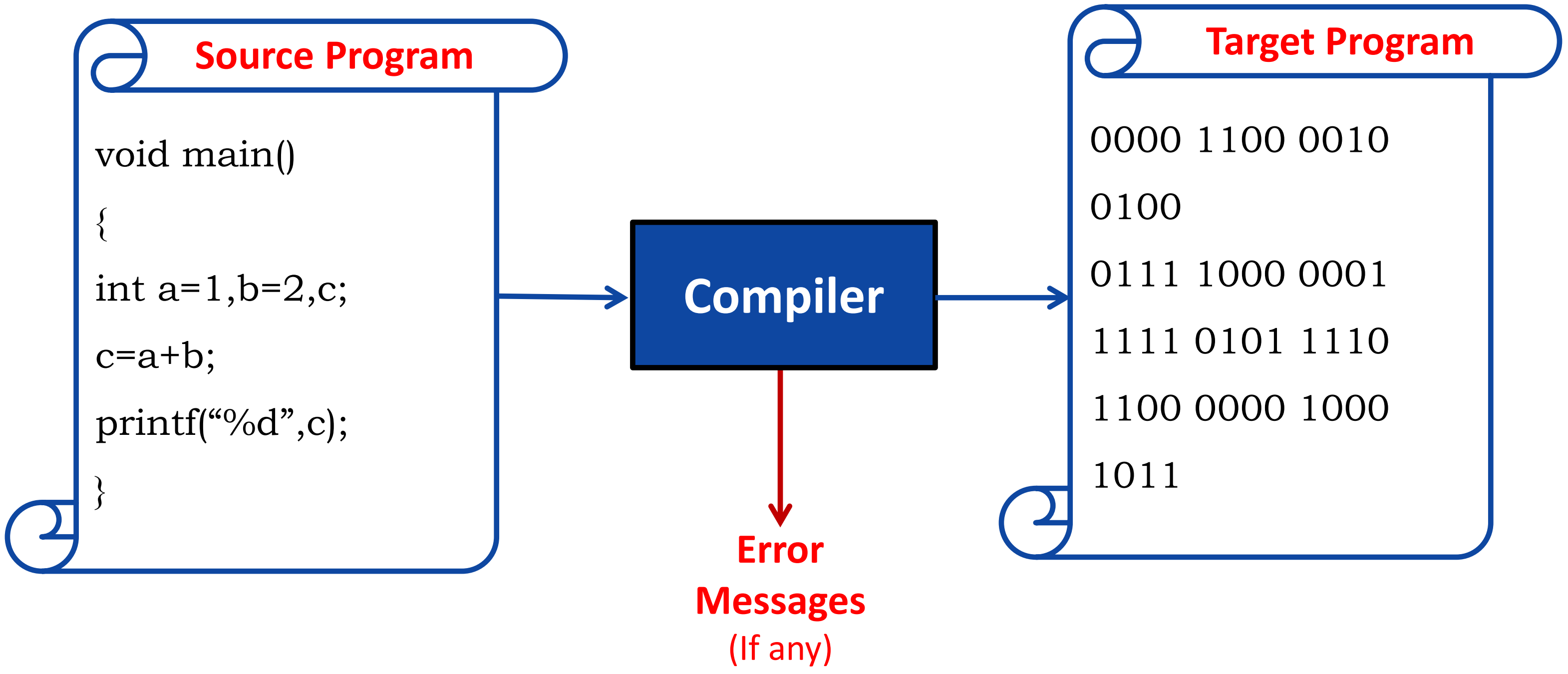
Types of translators are:

1. Compiler
2. Interpreter
3. Assembler



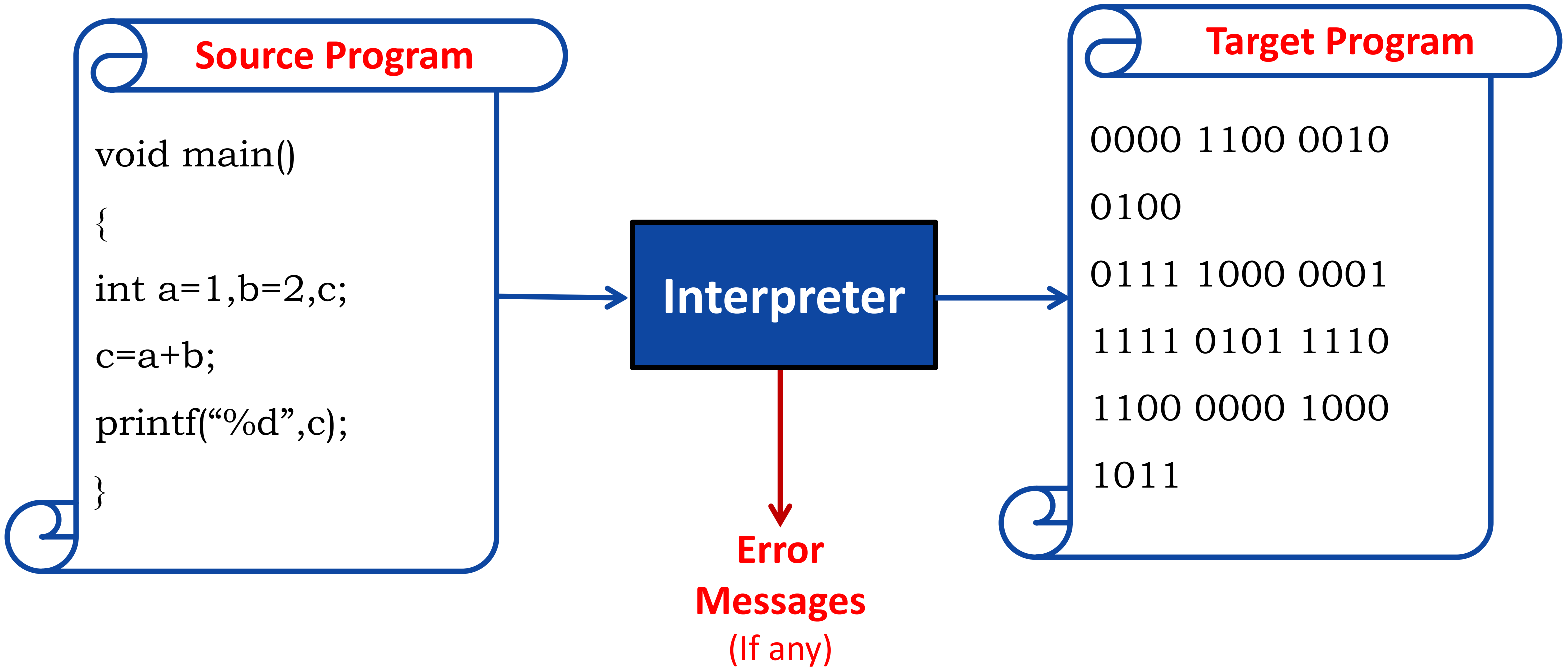
Compiler

A compiler is a program that reads a program written in the source language and translates it into an equivalent program in the target language.



Interpreter

The interpreter is also a program that reads a program written in the source language and translates it into an equivalent program in the target language line by line.

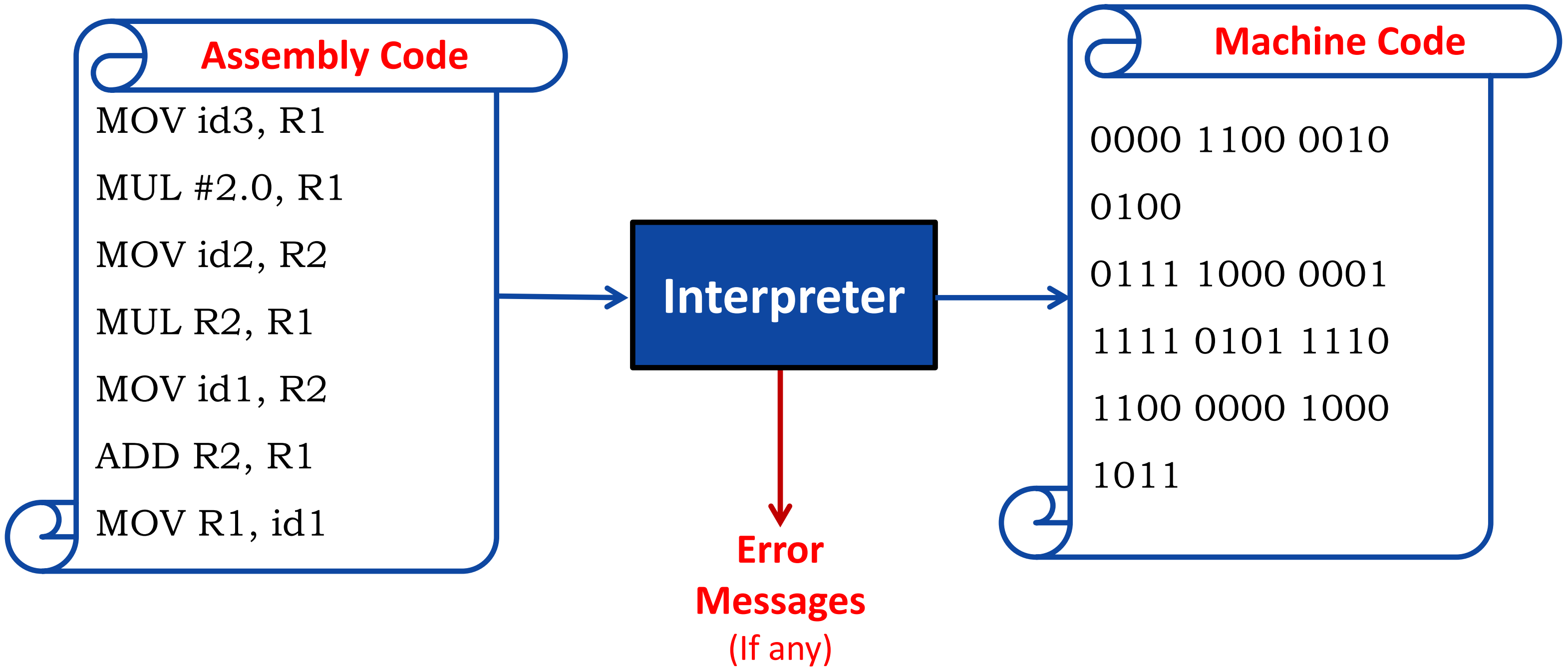


```
void main()
{
int a=1,b=2,c;
c=a+b;
printf("%d",c);
}
```

```
0000 1100 0010
0100
0111 1000 0001
1111 0101 1110
1100 0000 1000
1011
```

Assembler

Assembler is a translator which takes the assembly code as an input and generates the machine code as an output.

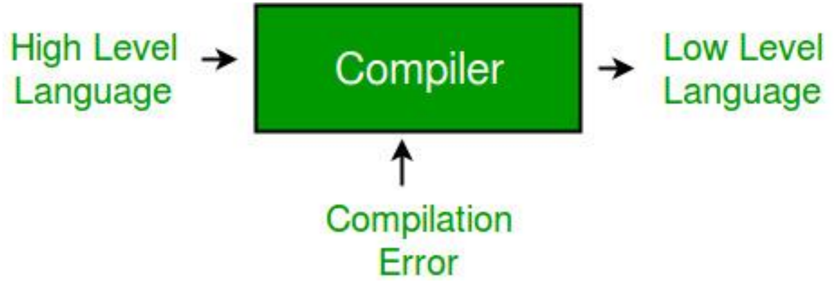


Introduction to Compiler

Introduction to Compiling:

A compiler translates the code written in one language to some other language without changing the meaning of the program. It is also expected that a compiler should make the target code efficient and optimized in terms of time and space.

Compiler design principles provide an in-depth view of translation and optimization process. Compiler design covers basic translation mechanism and error detection & recovery. It includes lexical, syntax, and semantic analysis as front end, and code generation and optimization as back-end.



Introduction to Compiler

The high-level language is converted into a binary language in various phases. A **compiler** is a program that converts high-level language to assembly language. Similarly, an **assembler** is a program that converts the assembly language to machine-level language.

Let us first understand how a program, using a C compiler, is executed on a host machine.

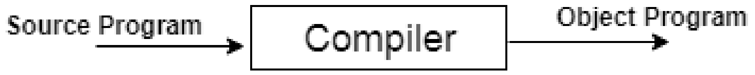
- User writes a program in C language (high-level language).
- The C compiler, compiles the program and translates it to an assembly program (low-level language).
- An assembler then translates the assembly program into machine code (object).
- A linker tool is used to link all the parts of the program together for execution.
- A loader loads all of them into memory and then the program is executed.

Introduction to Compiler

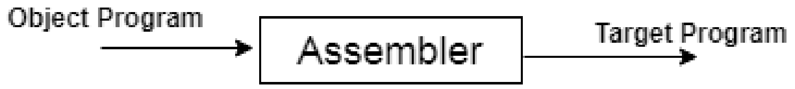
- A compiler is a translator that converts the High-level language into the machine language.
- High-level language is written by a developer and machine language can be understood by the processor.
- Compiler is used to show errors to the programmer.
- The main purpose of compiler is to change the code written in one language without changing the meaning of the program.
- When execute a program which is written in HLL programming language then it executes into two parts.

Introduction to Compiler

In the first part, the source program compiled and translated into the object program (Low Level Language).



In the second part, object program translated into the target program through the assembler.



Introduction to Compiler

Why use a Compiler?

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

Introduction to Compiler

Application of Compiler

- 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.

Introduction to Compiler

Why to Learn Compiler Design?

Computers are a balanced mix of software and hardware. Hardware is just a piece of mechanical device and its functions are being controlled by a compatible software. Hardware understands instructions in the form of **electronic charge**, which is the counterpart of binary language in software programming.

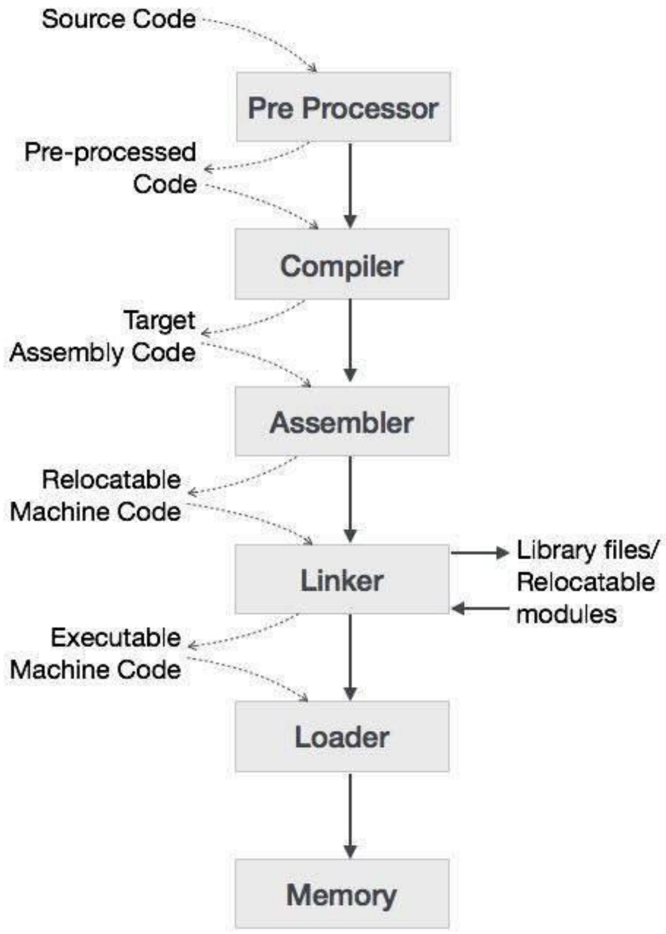
Binary language has only two alphabets, 0 and 1. To instruct, the hardware codes must be written in binary format, which is simply a series of 1s and 0s. It would be a difficult and cumbersome task for computer programmers to write such codes, which is why we have compilers to write such codes.

Introduction to Compiler

Language Processing System

We have learnt that any computer system is made of hardware and software. The hardware understands a language, which humans cannot understand. So we write programs in high-level language, which is easier for us to understand and remember.

These programs are then fed into a series of tools and OS components to get the desired code that can be used by the machine. This is known as Language Processing System.



Introduction to Compiler

Before the concepts of compilers, we should understand a few other tools that work closely with compilers.

1) Preprocessor

A preprocessor is generally considered as a part of a compiler, it is a tool that produces input for compilers. It deals with macro-processing, augmentation, file inclusion, language extension, etc.

2) Interpreter

An interpreter, like a compiler, translates high-level language into low-level machine language. The difference lies in the way they read the source code or input. A compiler reads the whole source code at once, creates tokens, checks semantics, generates intermediate code, executes the whole program and may involve many passes.

Introduction to Compiler

In contrast, an interpreter reads a statement from the input, converts it to an intermediate code, executes it, then takes the next statement in sequence. If an error occurs, an interpreter stops execution and reports it. whereas a compiler reads the whole program even if it encounters several errors.

3) Assembler

An assembler translates assembly language programs into machine code. The output of an assembler is called an object file, which contains a combination of machine instructions as well as the data required to place these instructions in memory.

4) Linker

The linker is a computer program that links and merges various object files together in order to make an executable file.

Introduction to Compiler

All these files might have been compiled by separate assemblers. The major task of a linker is to search and locate referenced modules/routines in a program and to determine the memory location where these codes will be loaded, making the program instruction have absolute references.

Introduction to Compiler: Types

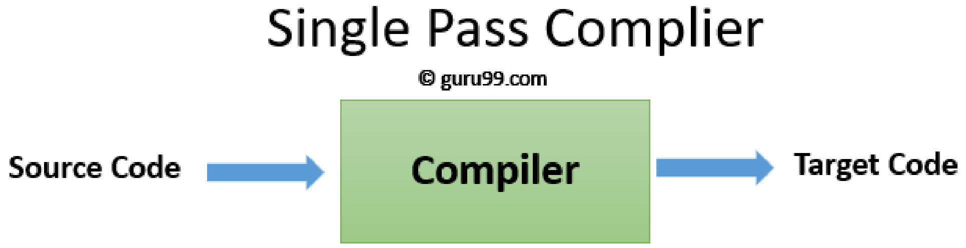
Types of Compiler

Following are the different types of Compiler:

- Single Pass Compilers
- Two Pass Compilers
- Multi Pass Compilers

1) Single Pass Compiler

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



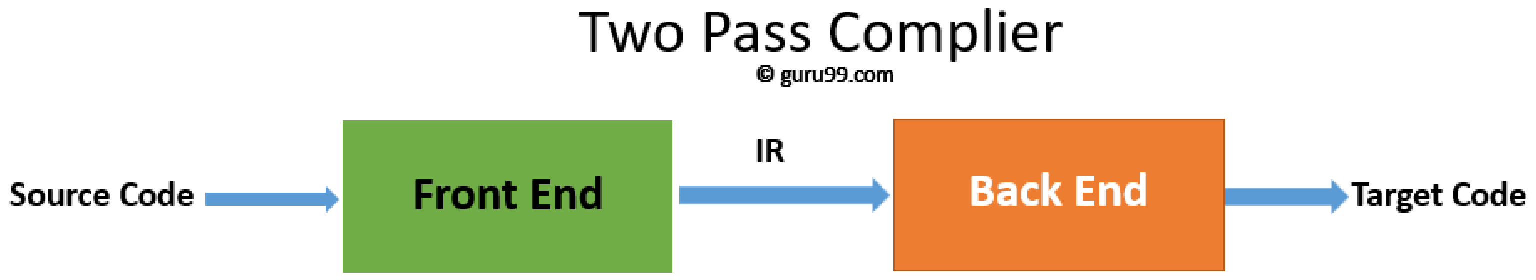
Introduction to Compiler: Types

2) Two-Pass Compiler

The 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

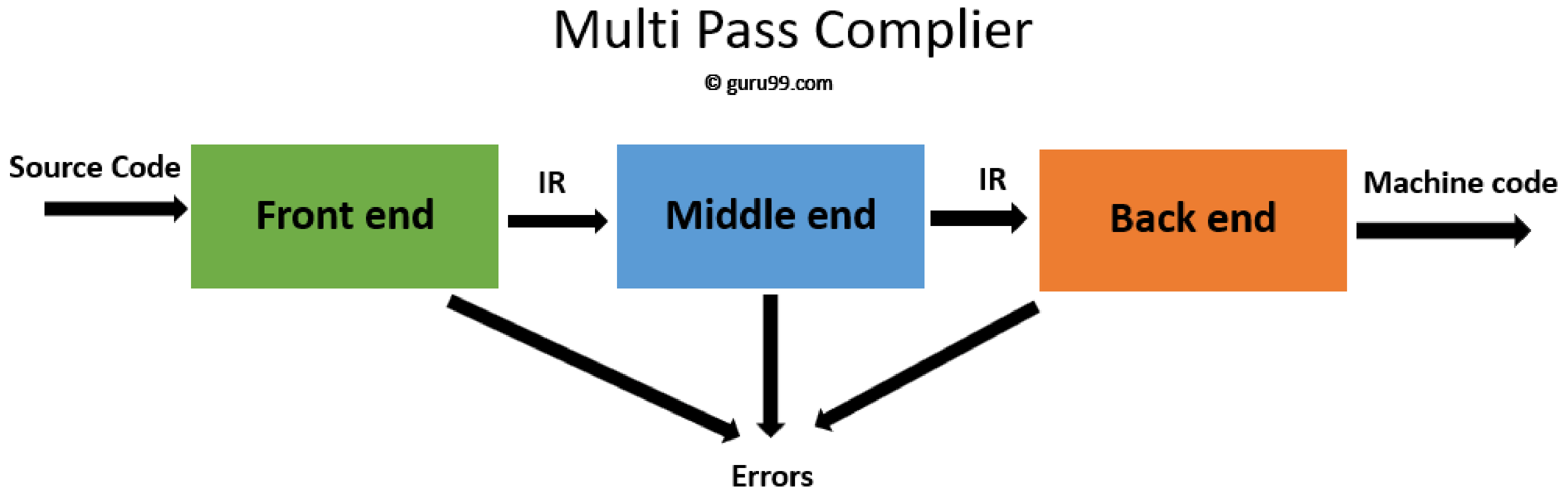
The Two-pass compiler method also simplifies the retargeting process. It also allows multiple fronts ends.



Introduction to Compiler: Types

3) Multi Pass Compiler

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'.



Introduction to Compiler: Tasks

The main tasks performed by the Compiler are:

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

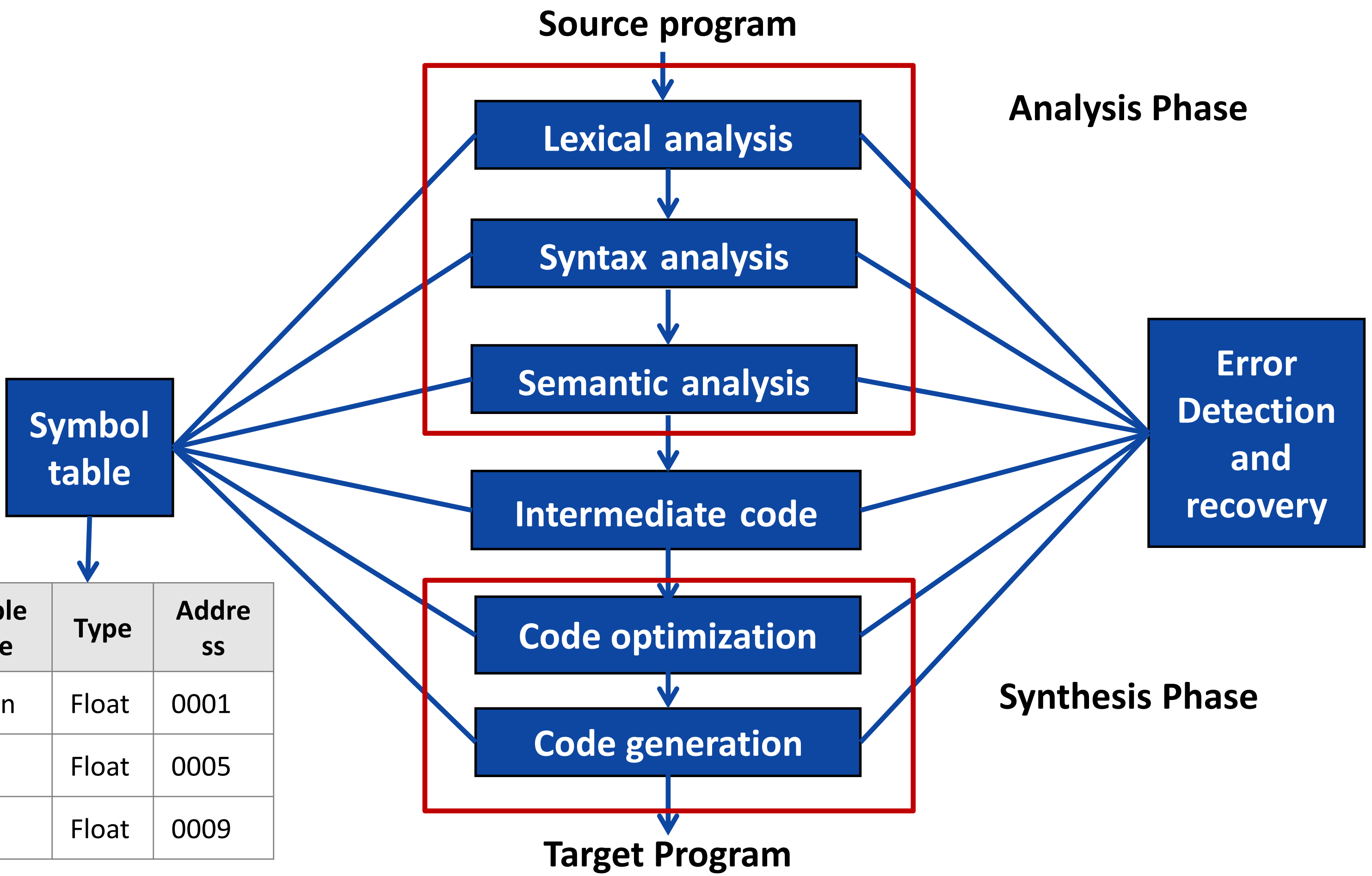
Phrases of Compiler

The compilation process contains the sequence of various phases. Each phase takes the source program in one representation and produces output in another representation. Each phase takes input from its previous stage.

Compiler operates in various phases each phase transforms the source program from one representation to another. Every phase takes inputs from its previous stage and feeds its output to the next phase of the compiler. All these phases convert the source code by dividing it into tokens, creating parse trees, and optimizing the source code by different phases. Each of these phases helps in converting the high-level langue of the machine code.

There are 6 phases in a compiler.

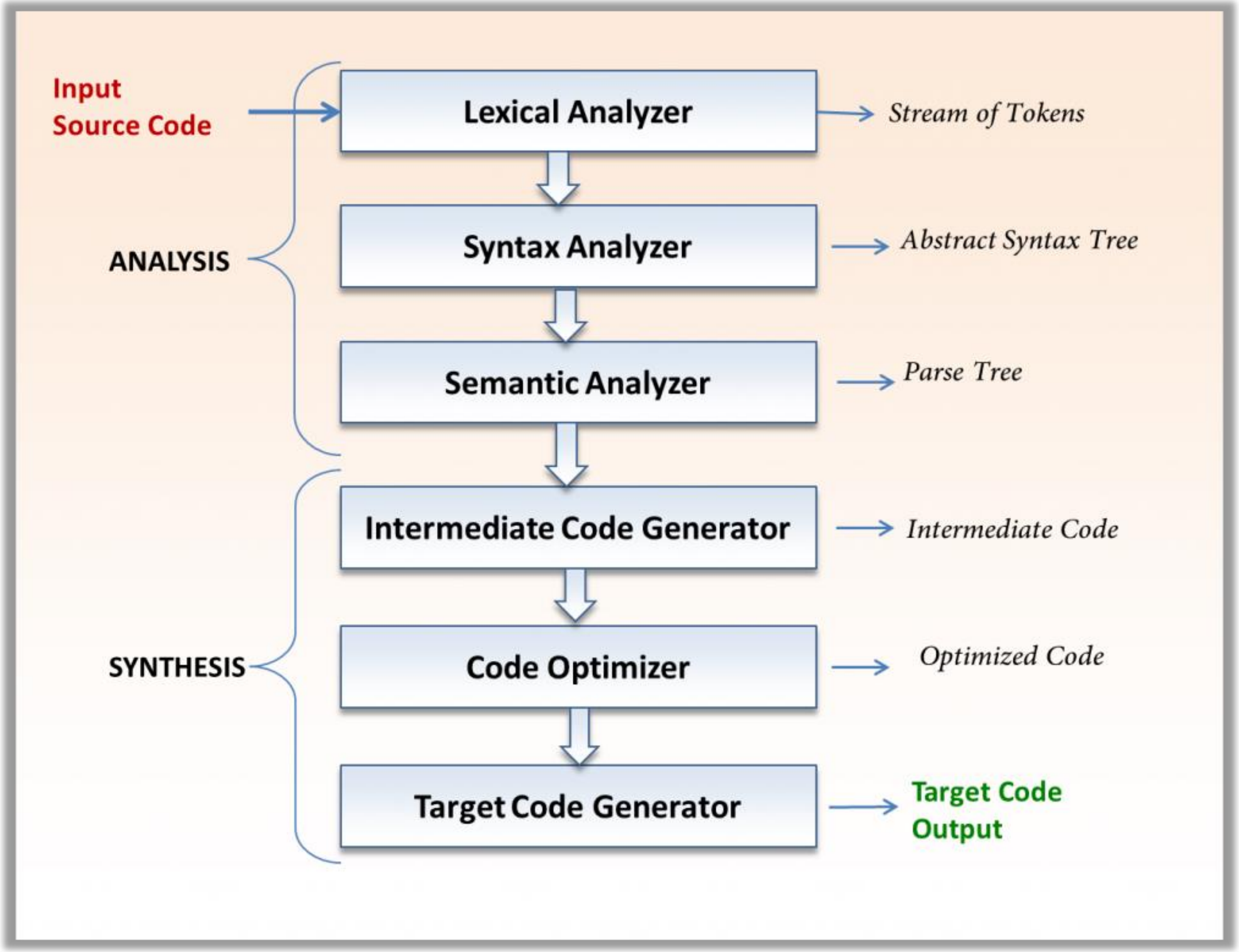
Phrases of Compiler



Variable Name	Type	Addre ss
Position	Float	0001
Initial	Float	0005
Rate	Float	0009

COMPILER DESIGN

Phrases of Compiler



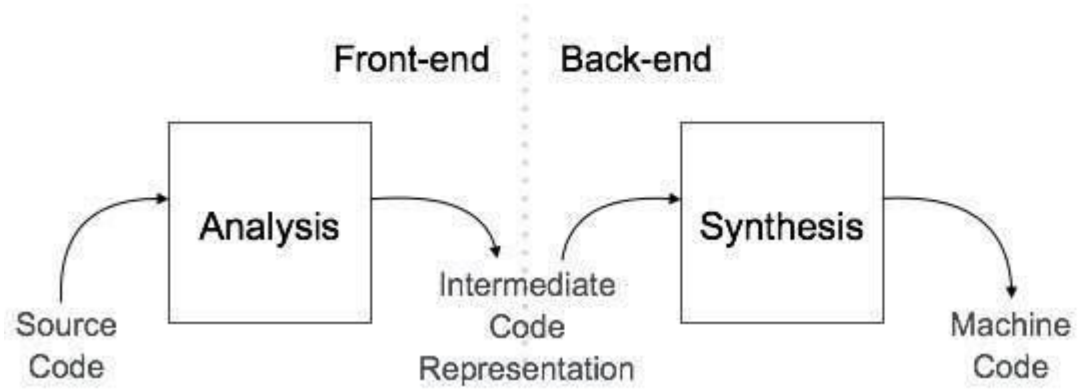
SDPadiya
SSGMCE

Phrases of Compiler

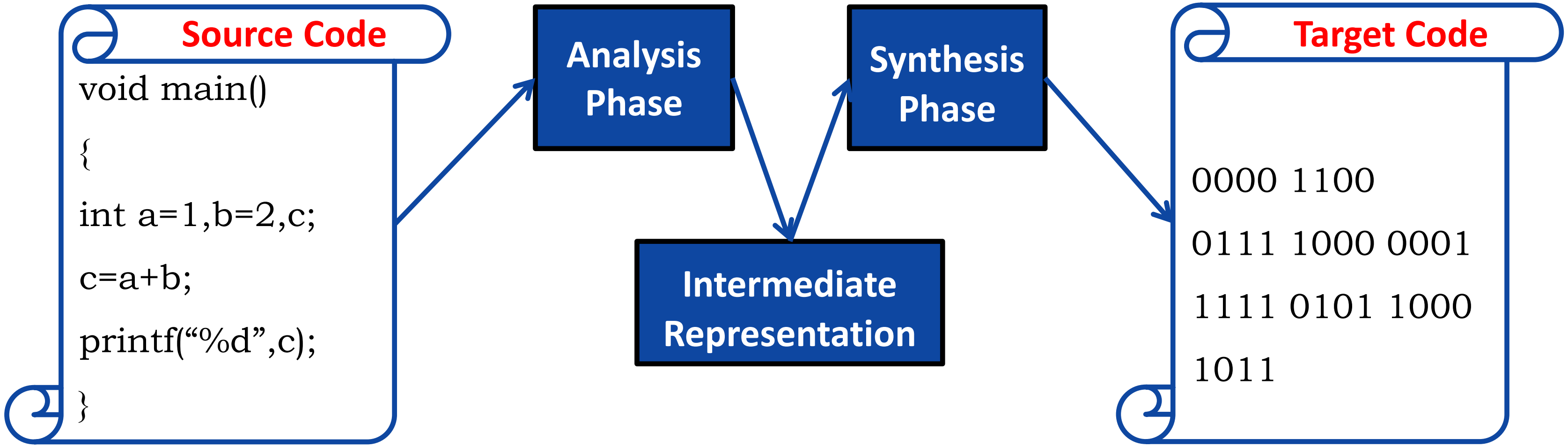
A compiler can be divided (grouped) into two phases based on the way they are compiled.

1) Analysis Phase: Known as the front-end of the compiler, the **analysis** phase of the compiler reads the source program, divides it into core parts and then checks for lexical, grammar and syntax errors. This phase generates an intermediate representation of the source program and symbol table, which should be fed to the Synthesis phase as input.

2) Synthesis Phase: Known as the back-end of the compiler, the **synthesis** phase generates the target program with the help of intermediate source code representation and symbol table.



Phrases of Compiler



```
void main()
{
int a=1,b=2,c;
c=a+b;
printf("%d",c);
}
```

```
0000 1100
0111 1000 0001
1111 0101 1000
1011
```

Phrases of Compiler

Analysis Phase

The analysis part breaks up the source program into constituent pieces and creates an intermediate representation of the source program.

The analysis phase consists of three sub-phases:

1. Lexical analysis
2. Syntax analysis
3. Semantic analysis

Synthesis Phase

The synthesis part constructs the desired target program from the intermediate representation.

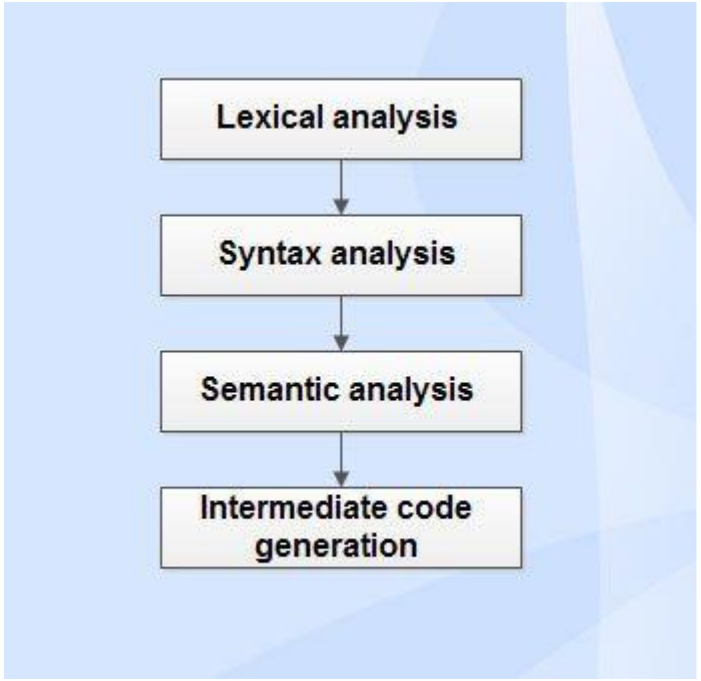
The synthesis phase consists of the following sub-phases:

1. Code optimization
2. Code generation

Phrases of Compiler

1) Analysis Phase / Front End: Front end of a compiler consists of the phases

1. Lexical analysis.
2. Syntax analysis.
3. Semantic analysis.
4. Intermediate code generation.



Phrases of Compiler

1) Analysis Phase / Front End:

A front end comprises phases that are **dependent on the input (source language) and independent on the target machine** (target language).

It includes lexical and syntactic analysis, symbol table management, semantic analysis and the generation of intermediate code.

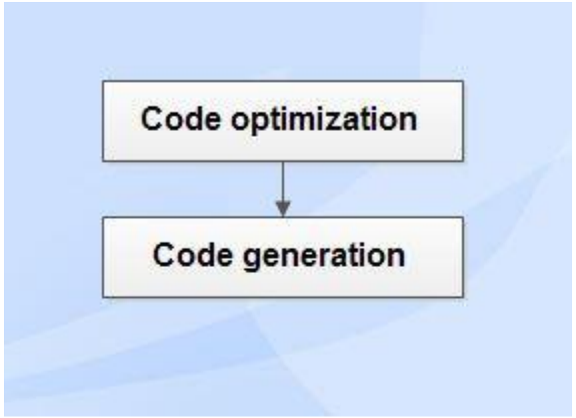
Code optimization can be done by the front end, it also includes error handling at the phases concerned.

Phrases of Compiler

2) Synthesis Phase / Back End

The back end of a compiler contains

1. Code optimization.
2. Code generation.

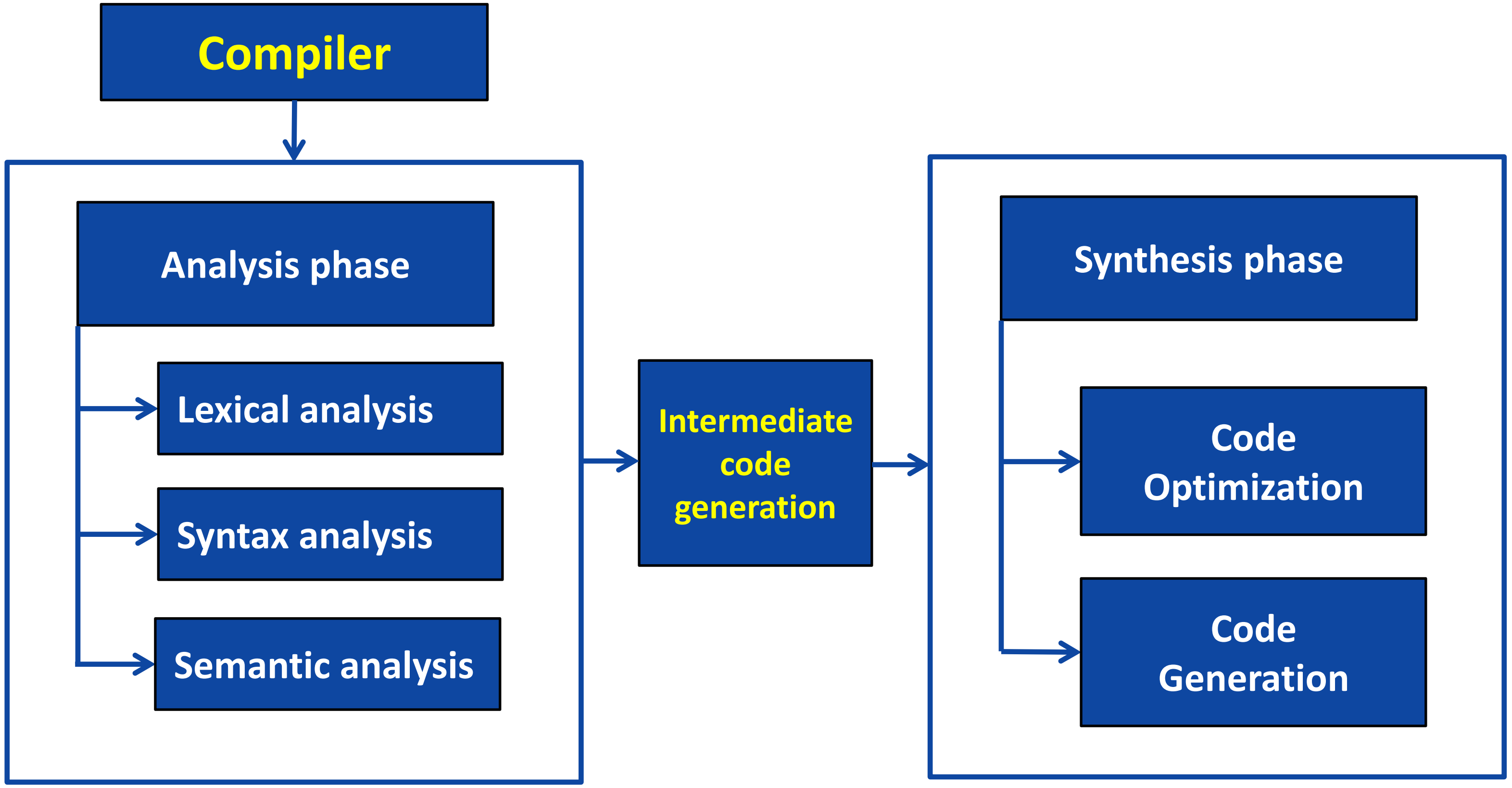


The back end comprises those phases of the compiler that are **dependent on the target machine and independent on the source language**.

This includes code optimization, code generation.

In addition to this, it also encompasses error handling and symbol table management operations

Phrases of Compiler



Phrases of Compiler

Phase 1: Lexical Analysis:

Lexical Analysis is the first phase when the 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 the 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 a token that is not a part of the language.

Phrases of Compiler

Phase 1: Lexical Analysis:

Example:

X = Y + 10

Tokens:

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

$$\begin{tikzcd}
\text{Position = initial + rate*60} \arrow[d] \\
\boxed{\text{Lexical analysis}} \arrow[d] \\
\text{id1 = id2 + id3} * 60
\end{tikzcd}$$

Phrases of Compiler

Phase 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 the source language and the syntax of the language. **Here, is a list of tasks performed :**

- 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

Phrases of Compiler

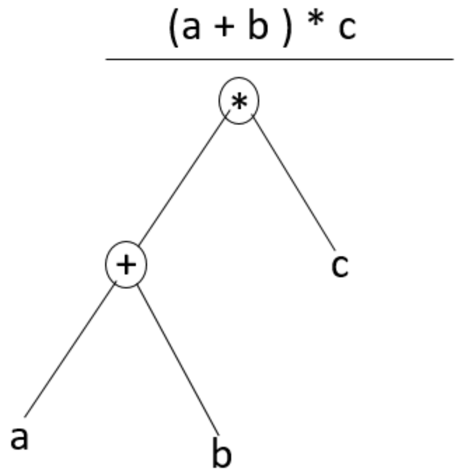
Phase 2: Syntax Analysis:

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



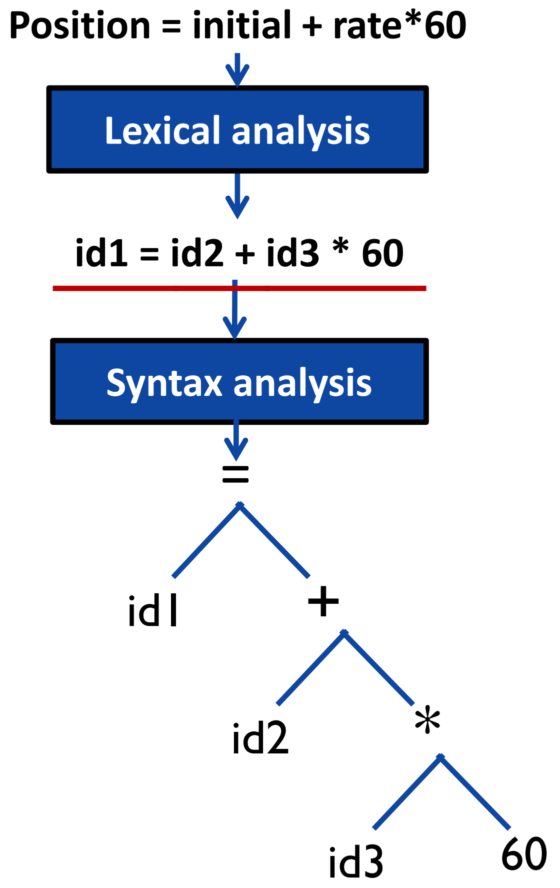
C O M P I L E R D E S I G N

SDPadiya

SSGMCE

Phrases of Compiler

Phase 2: Syntax Analysis:



Phrases of Compiler

Phase 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.

Functions of the Semantic analyses phase are:

- Helps to store type information gathered and save it in symbol table or syntax tree.
- Allows to performs type checking

Phrases of Compiler

Phase 3: Semantic Analysis

- 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

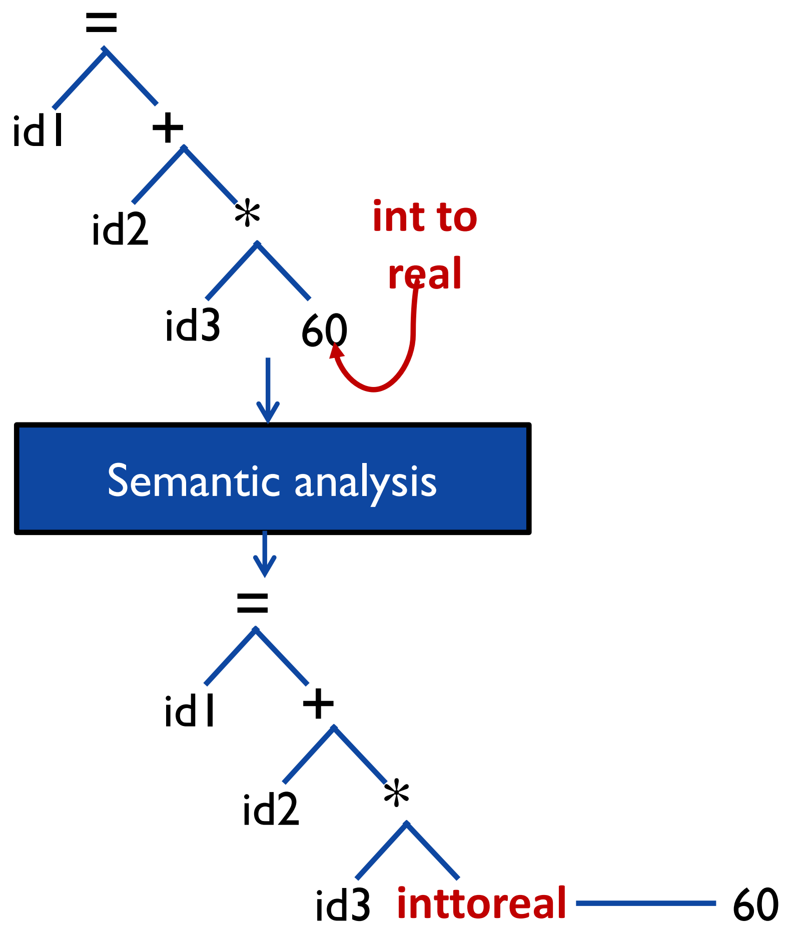
float X = 20.2;

float X*30;

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

Phrases of Compiler

Phase 3: Semantic Analysis



Phrases of Compiler

Phase 4: Intermediate Code Generation

After the semantic analysis phase, the compiler generates intermediate code for the target machine. It represents a program for some abstract machine.

Intermediate code is between the HL and ML 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 a target language.
- Allows you to maintain precedence ordering of the source language.
- It holds the correct number of operands of the instruction.

Phrases of Compiler

Phase 4: Intermediate Code Generation

Example

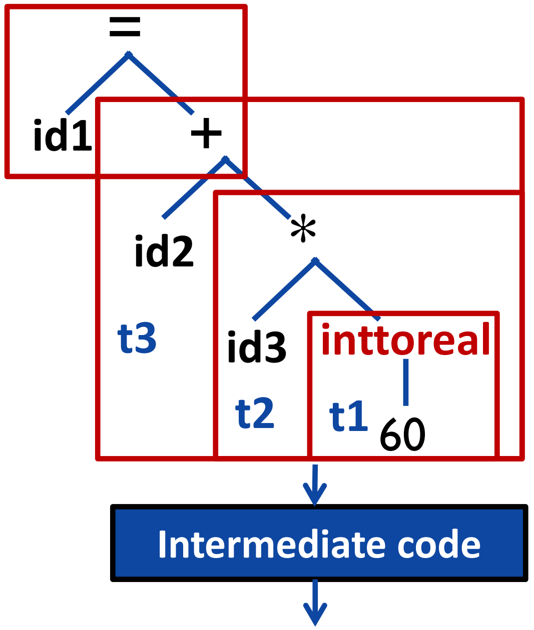
Position = initial + rate*60

Intermediate code with the help of address code method is:

```
t1 := inttoreal (60)
t2 := rate * t1
t3 := initial + t2
Position := t3
```

Phrases of Compiler

Phase 4: Intermediate Code Generation



t1 = int to real(60)

t2 = id3 * t1

t3 = t2 + id2

id1 = t3

Phrases of Compiler

Phase 5: Code Optimization

The next phase is code optimization or Intermediate code. This phase removes unnecessary code lines and arranges the sequence of statements to speed up the execution of the program without wasting resources. The main goal of this 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 an intermediate representation
- Removing unreachable code and getting rid of unused variables
- Removing statements which are not altered from the loop

Phrases of Compiler

Phase 5: Code Optimization

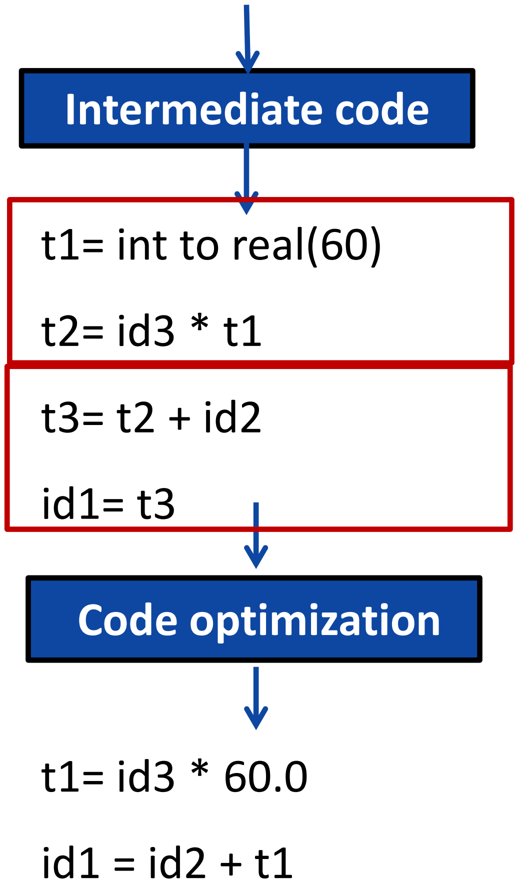
Example:

Consider the following code

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

Can become

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



```
t1= int to real(60)
t2= id3 * t1
t3= t2 + id2
id1= t3
```

```
t1= id3 * 60.0
id1 = id2 + t1
```

Phrases of Compiler

Phase 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 covers the optimized 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.

Phrases of Compiler

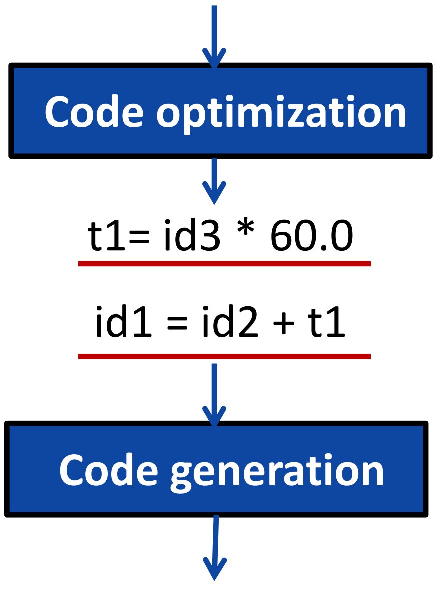
Phase 6: Code Generation

Example:

a = b + 60.0

Would be possibly translated to registers.

```
MOVF a, R1
MULF #60.0, R2
ADDF R1, R2
```

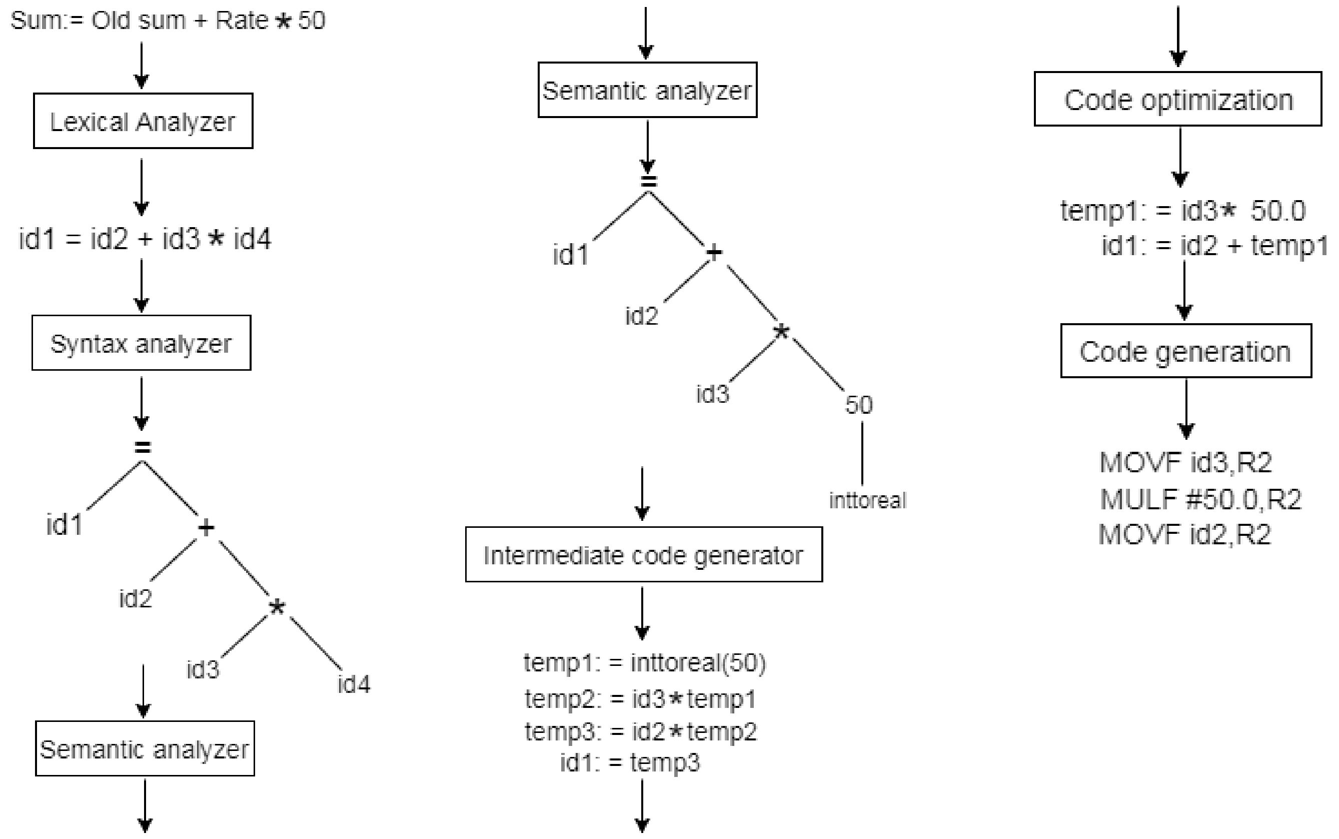


```
t1= id3 * 60.0
id1 = id2 + t1
```

```
MOV id3, R2
MUL #60.0, R2
MOV id2, R1
ADD R2,R1
MOV R1, id1
```

Id3→R2
Id2→R1

Phrases of Compiler



Phrases of Compiler

Error Handling Routine:

In the compiler design process error may occur in all the below-given phases:

1. Lexical analyzer: Wrongly spelt tokens
2. Syntax analyzer: Missing parenthesis
3. Intermediate code generator: Mismatched operands for an operator
4. Code Optimizer: When the statement is not reachable
5. Code Generator: When the memory is full or proper registers are not allocated
6. Symbol Tables: Error of multiple declared identifiers

Phrases of Compiler

Error Handling Routine:

Most common errors are invalid character sequences in scanning, invalid token sequences in type, scope error, and parsing in semantic analysis. The error may be encountered in any of the above phases.

After finding errors, the phase needs to deal with the errors to continue with the compilation process. These errors need to be reported to the error handler which handles the error to perform the compilation process. Generally, the errors are reported in the form of messages.

Compiler Construction Tools

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

These tools use specific language or algorithms 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 the goal of associating one or more translations with each node of the parse tree.

Compiler Construction Tools

- **Parser Generators:** A parser generator takes grammar as input and automatically generates source code that can parse streams of characters with the help of 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.

Thank You

Prof. S. D. Padiya
padiyasagar@gmail.com
SSGMCE, Shegaon