CPSC 323: Compilers and Languages

Chapter 1 – Introduction to Compilers

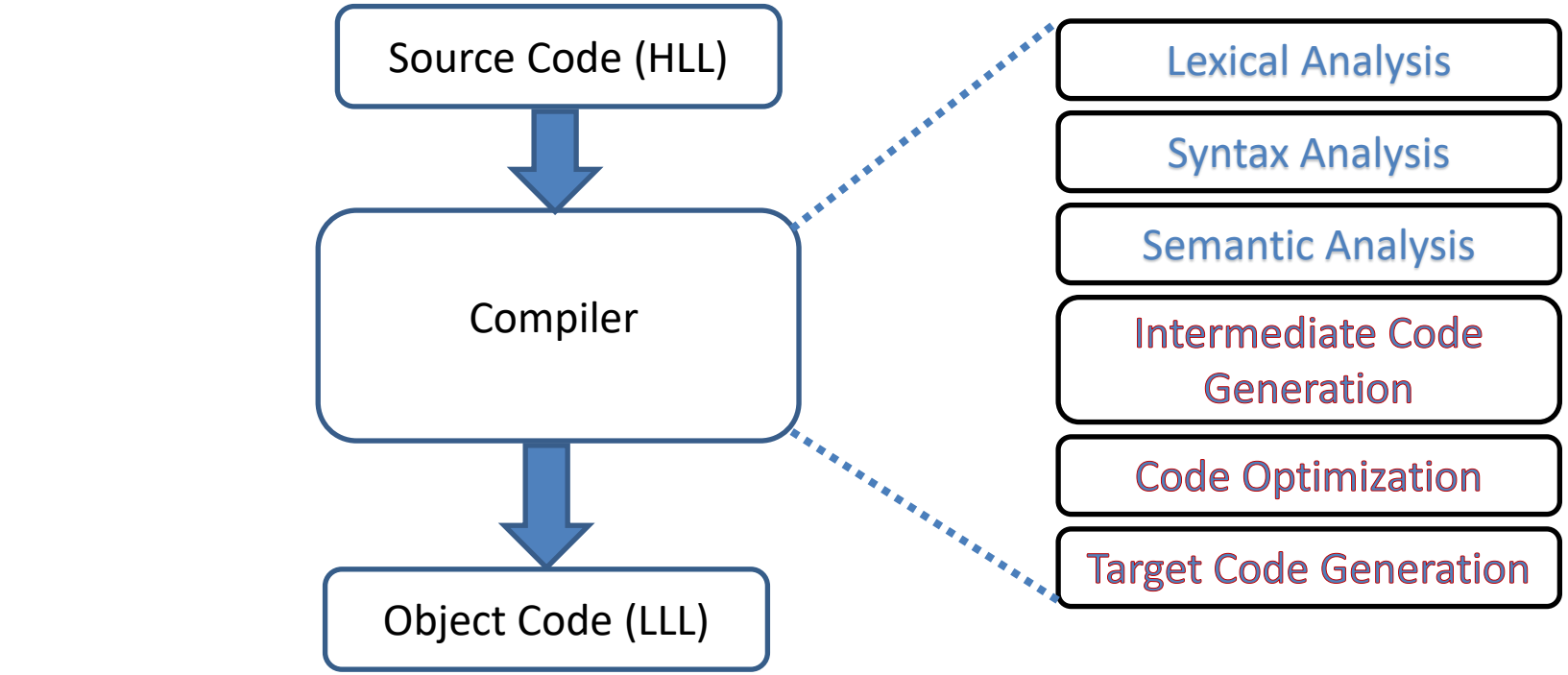
Mr. Param Venkat Vivek Kesireddy
Teaching Associate
Mail: pkesireddy@fullerton.edu

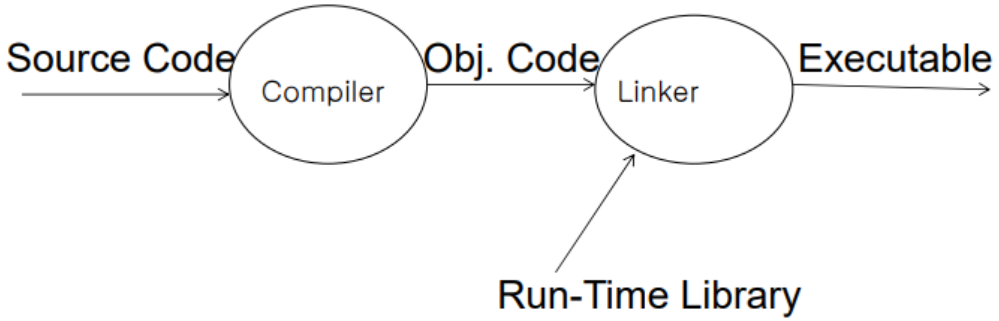
Inputs from Prof Doina Bein & Prof James Choi

Structure of Compiler

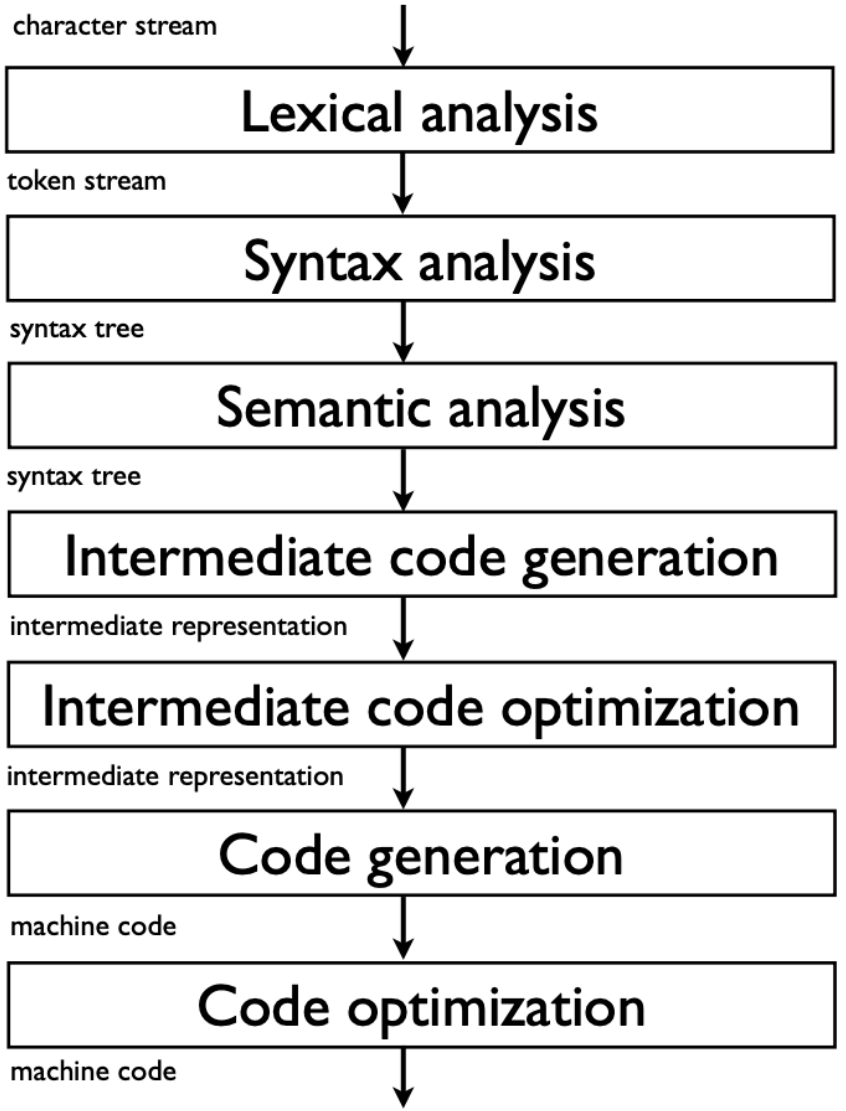
1. Lexical Analysis (Scanning)
2. Syntax Analysis (Parsing)
3. Semantic Analysis
4. Intermediate Code Generation
5. Code Optimization
6. Target Code Generation

Structure of Compiler





Structure of a compiler



- Run-time library: Collection of modules (functions) for computing certain functions such as sqrt(x), sin(x), scanf() etc.
- Linker: provides the run-time library to the object code and generates an executable code.
- Executable (also called the Binary) is the output of a linker after it processes the object code. A machine code file can be immediately executable (i.e., runnable as a program), or it might require linking with other object code files (e.g., libraries) to produce a complete executable program.

- We basically have two phases of compilers, namely the Analysis phase and Synthesis phase. The analysis phase creates an intermediate representation from the given source code. The synthesis phase creates an equivalent target program from the intermediate representation.

- Cross Compiler: a compiler that generates an object code for a machine that is different from the machine on which the compiler runs. E.g., compile a source in C on a PC-platform but the output (object code) runs on a different machine such as Unix, Apple OS etc.

Compiler Vs Interpreter

- A compiler is a software tool that translates the entire source code of a program into machine code or an intermediate representation all at once.
- An interpreter, on the other hand, processes the source code line by line and executes it directly without creating a separate compiled binary.

Lexical Analyzer

Lexemes and Tokens

- A **Lexeme** is a string of characters that is a lowest-level syntactic unit in the programming language. These are the "words" and punctuation of the programming language.
- A **Token** is a syntactic category that forms a class of lexemes. These are the "nouns", "verbs", and other parts of speech for the programming language.
- In a practical programming language, there are a very large number of lexemes, perhaps even an infinite number. In a practical programming language, there are only a small number of tokens.
- A program - Basic task is to scans the source code as a stream of characters and converts it into meaningful lexemes. These lexemes in the form of tokens.

- Lexical analyzer which breaks the source code up into meaningful Units called tokens
- Meaningful units may be different from language to language

Ex. C source code: if (x == y) a = b - 5;

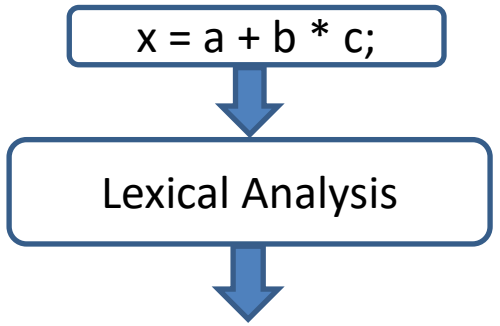
tokens:

1. Keywords: if
2. Identifiers: x, y, a, b
3. Constants: 5
4. Operators: ==, =, -
5. Punctuations: ;
6. Parentheses: (,)

Lexical Analysis frequently does other operations as well:

- removes excessive white spaces (blank, tab etc.)
- overpasses comments
- case conversion if needed
- It is recognition of a regular language, e.g., via a DFA

Lexical Analyzer



Lexemes	Tokens
x	identifier
=	operator
a	identifier
+	operator
b	identifier
*	operator
c	identifier
;	symbol

- Lexeme - A basic lexical unit of a language, consisting of one word or several words, considered as an abstract unit.
- Token - A token is a syntactic category that forms a class of lexemes.

- In the C language, the following 6 types of tokens are available:

  Identifiers
  Keywords
  Constants
  Operators
  Special Characters
  Strings

Consider the following code that is fed to Lexical Analyzer (scanner):

```
#include <stdio.h>
    int maximum(int x, int y) {
        // This will compare 2 numbers
        if (x > y)
            return x;
        else {
            return y;
        }
    }
```

Lexeme	Token
int	Keyword
maximum	Identifier
(	Operator
int	Keyword
x	Identifier
,	Operator
int	Keyword
Y	Identifier
)	Operator
{	Operator
If	Keyword

REGULAR LANGUAGE:

- A regular language is considered as a finite set of strings over some finite set of alphabets.
- Computer languages are considered as finite sets, and mathematically set operations can be performed on them.
- A regular language is a formal language that can be expressed *by means of* ***regular expressions*** *by defining a pattern for finite strings of symbols.*
- The grammar defined by regular expressions is known as **regular grammar.**

For lexical analysis, we care about *regular languages*

- The lexical analyzer needs to scan and identify only *a finite set* of valid tokens that belong to the programming language in hand.
- Tokens can be described by regular expressions.

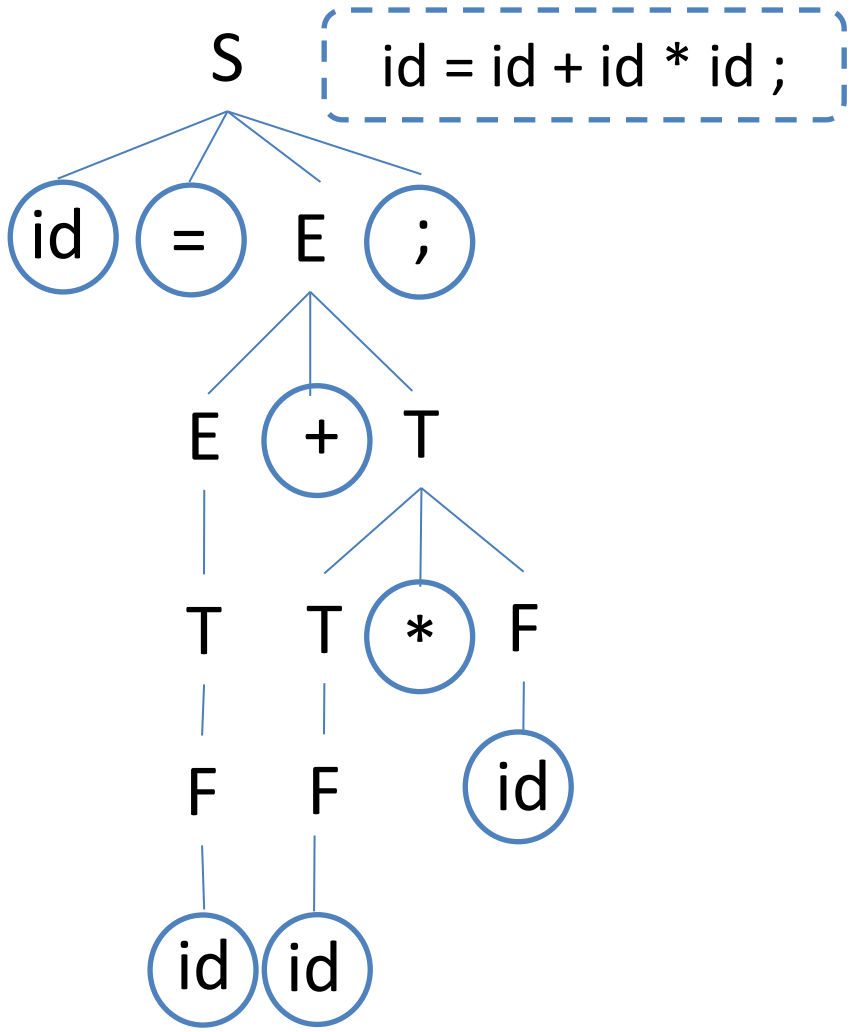
Syntax Analyzer

Lexemes	Tokens
x	identifier
=	operator
a	identifier
+	operator
b	identifier
*	operator
c	identifier
;	symbol

x = a + b * c;

Syntax Analysis

S → id = E;
E → E + T | T
T → T * F | F
F → id



Parse Tree

Other Phases of Compiler

(to be explained in detail in further chapters)

- Semantic Analysis: The third phase of a compiler is semantic analysis. This phase checks whether the code is semantically correct, i.e., whether it conforms to the language's type system and other semantic rules.

- Intermediate Code Generation: The fourth phase of a compiler is intermediate code generation. This phase generates an intermediate representation of the source code that can be easily translated into machine code.

- Optimization: The fifth phase of a compiler is optimization. This phase applies various optimization techniques to the intermediate code to improve the performance of the generated machine code.

- Code Generation: The final phase of a compiler is code generation. This phase takes the optimized intermediate code and generates the actual machine code that can be executed by the target hardware.

- During the compilation Process, we need 2 additional matters to take care of
- Symbol Table Handler: Symbol table is the central depository of information about names (variables, functions, types etc.) created by the source code. This handler inputs, looks up the symbols.
- Error Handler: deals with errors that may occur during the compilation. - Detection, Message and Recovery

- Pass: Pass is reading one version of a program(source) from a file and writing a new version.
- One Pass Compiler: reads the source and writes the object code into a file.
- Multiple Pass Compiler: 2 or more passes are necessary to create the object code.
- Reasons: - Some information is NOT available during initial pass - NOT enough memory to hold all intermediate results.

Writing a Compiler

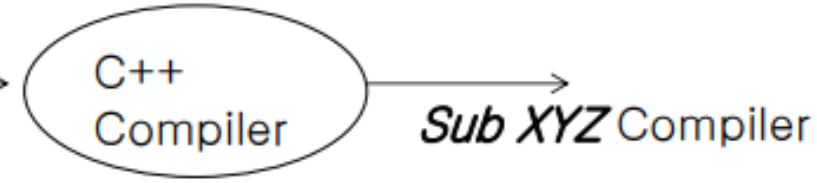
a) Choice of a language:

Any HLL can be used to write a compiler

Q: Can we write a compiler in its own language (ex. XYZ compiler in XYZ language)?

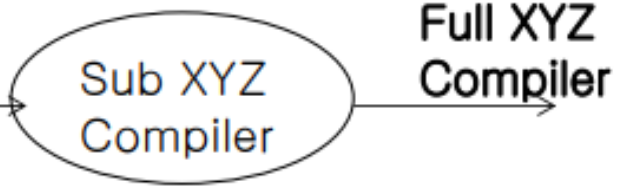
A: YES/no => YES

How? => Bootstrapping:



Compiler for *Sub XYZ* in C++

Full XYZ Compiler in *Sub XYZ*



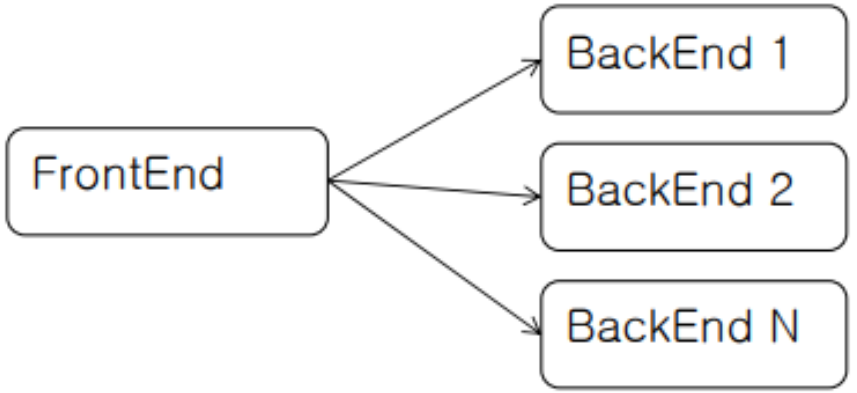
b)Writing Re-targetable Compilers:

Compiler Generation technique that can be used with different machines (Platform)

Q: How do we write a such compiler

<u>Method 1:</u>

Distinguish Front/Back end of the compilation process
Front End: LA, SA, ICG, OPT-IC (Machine Independent)
Back End: OCG, OPT-OC (Machine Dependent)



Method 2) Write a compiler for an imaginary machine (P-Code)