CPSC 323
Compilers and Languages

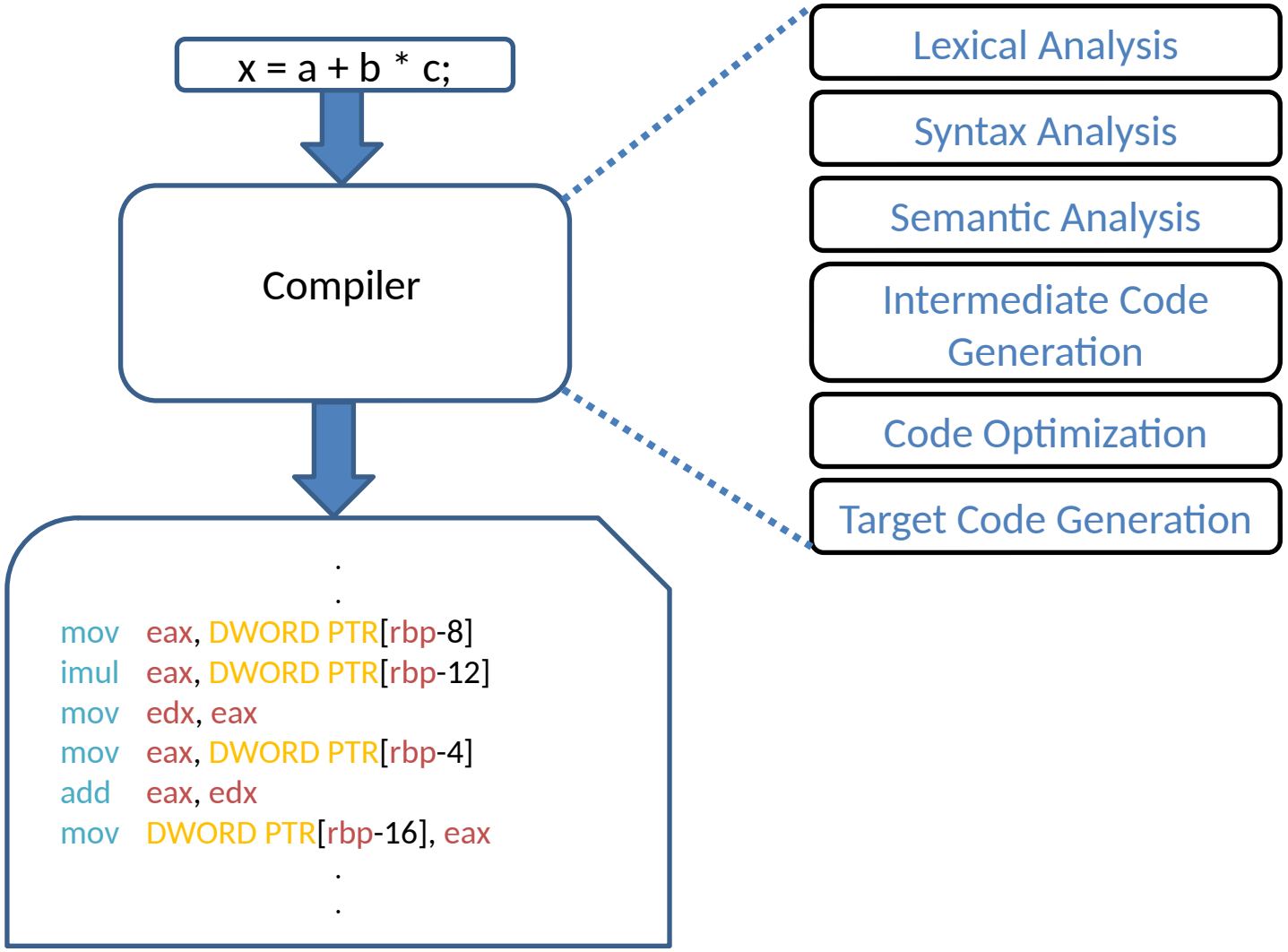
Miss Susmitha Padda
spadda@fullerton.edu

Inputs from Rong Jin and Doina Bein

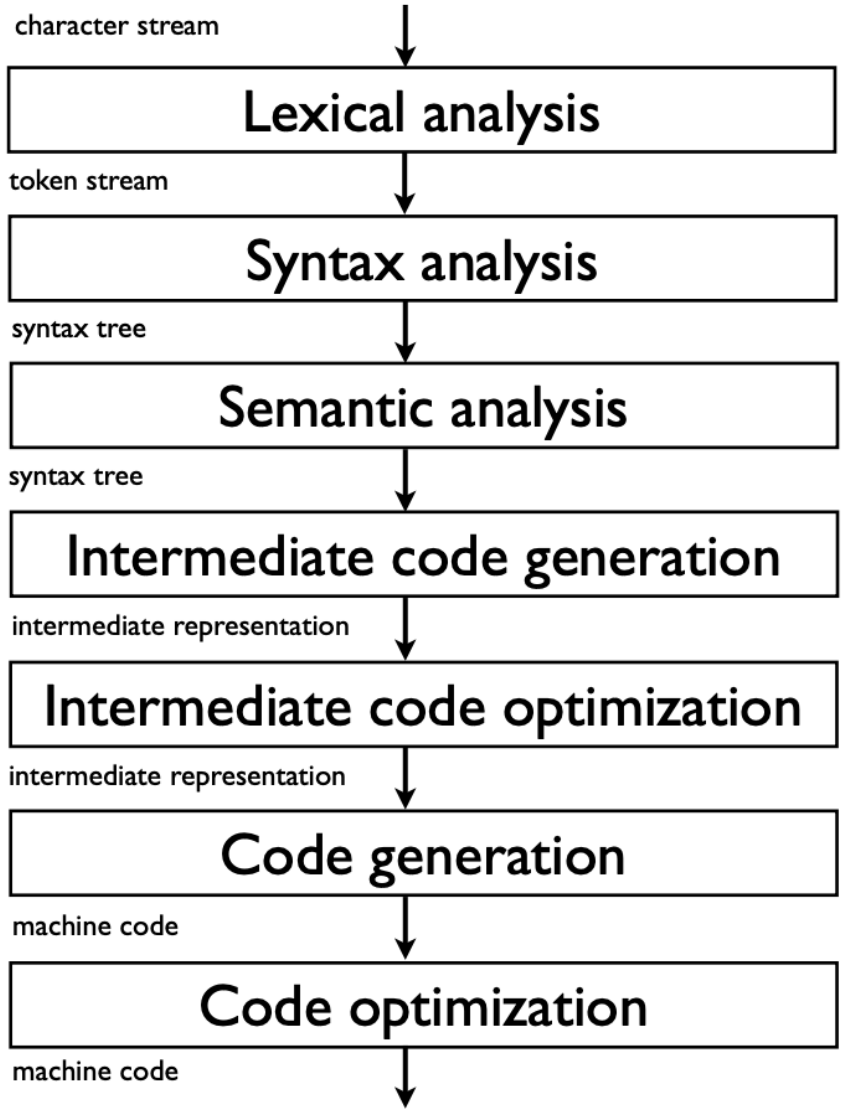
Structure of Compiler

1. Lexical Analysis
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



Lexical Analyzer

Lexemes and Tokens

- A **Lexeme** is a string of characters that is a lowest-level syntatic 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.

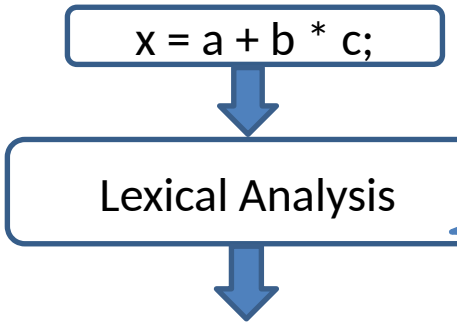
- Lexical analyzer which breaks the source code up into meaningful
  Units called tokens
- Meaningful units may be different from language to language
  Ex. C source code: if (x == y) a = b - 5;
  tokens:

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

Recognizes tokens using Regexs.
E.g. Regex for identifier:
l(l+d)* | _(l+d)*
l: letter
d: digit
_: underscore

Examples of Tokens & Lexemes & Patterns

Token	Sample Lexemes	Informal description of pattern
if	if	if
While	While	while
Relation	<, <=, = , <>, > >=	< or <= or = or <> or > or >=
Id	count, sun, i, j, pi, D2	Letter followed by letters and digits
Num	0, 12, 3.1416, 6.02E23	Any numeric constant

Demo for reference :
https://www.youtube.com/watch?v=VGgIZl5WjH0

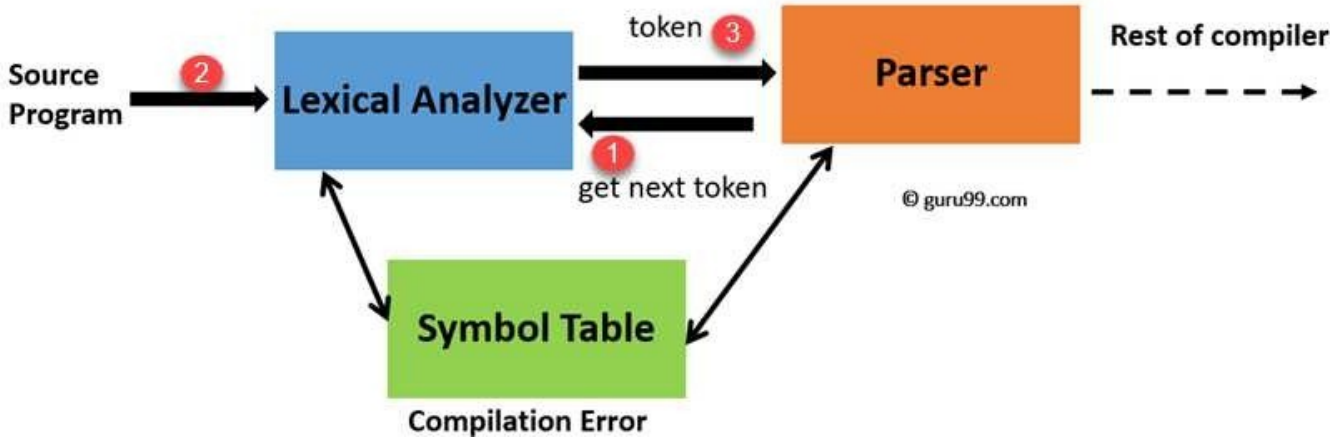
Consider the following code that is fed to Lexical Analyzer (scanner):

```
#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

How are tokens recognized?

1. “Get next token” is a command which is sent from the parser to the lexical analyzer.
2. On receiving this command, the lexical analyzer scans the input until it finds the next tok
3. It returns the token to Parser.



SUMMARY

- **A token** is a generic type, as passed to the parser.
- **A Lexeme** is the actual/specific instance of the token.

SPECIFICATION OF TOKENS :

- **Alphabets** σ
  - Any finite set of symbols {0,1} is a set of binary alphabets $\sigma = \{0,1\}$
  - σ = {0,1,2,3,4,5,6,7,8,9,A,B,C,D,E,F} is a set of Hexadecimal alphabets
  - σ = {a-z, A-Z} is a set of English language alphabets.

- **Strings**
  - Any finite sequence of alphabets (characters) is called **a string**.
  - Length of the string is the total number of occurrence of alphabets. e.g., the length of the string 'tutorialspoint' is 14 and is denoted by |tutorialspoint| = 14.
  - A string having no alphabets, i.e., a string of zero length is known as an **empty string** and is denoted by **ε** (epsilon)

  - *Example:* if $\sigma = \{0, 1\}$, then 10, 1000, 0, 101 and ϵ are strings over σ

REGULAR LANGUAGE:

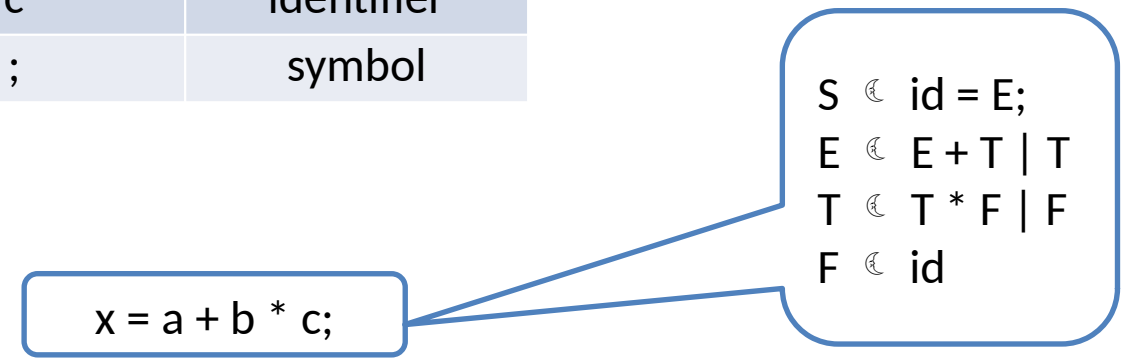
- A regular language is considered as a finite set of strings over some finite set of alphabets.
- Computer languages are considered as finite sets, and mathematically set operations can be performed on them.
- A regular language is a formal language that can be expressed *by means of* ***regular expressions***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

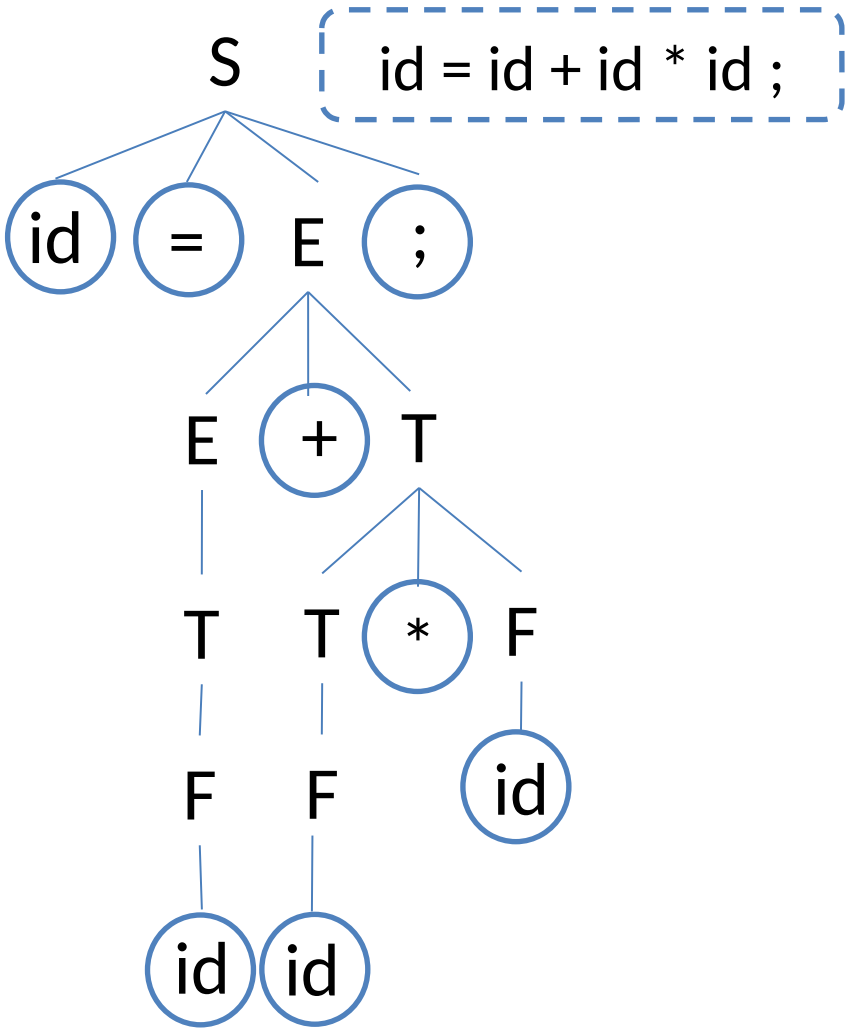
For lexical analysis, we care about *regular languages*

- 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





Parse Tree

Syntax Analyzer

- The Syntax analyzer also known as the parser.
- The Syntax analyzer depends on the type two or context free grammars.
- In order to find out the yield of the parse tree, we will have to traverse it top to bottom left to right.
- In short, taking the stream of tokens, the syntax analyzer analyzes them following specific set of production rules and produces the parse tree. If the yield of the parse tree and the providers stream of tokens are the same, then there is no error, otherwise there is some syntax error in the statement.

Questions ?