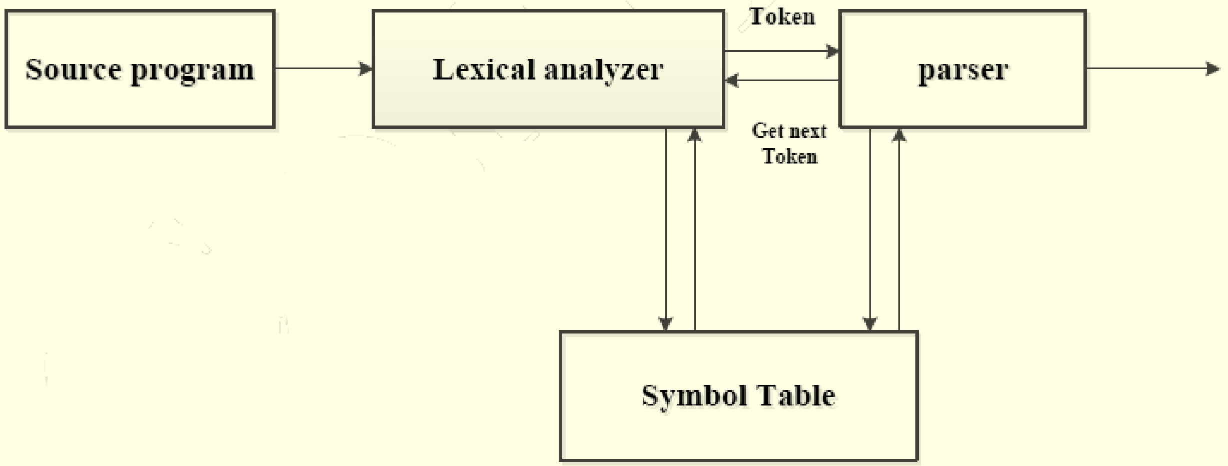
Complier 1

Lecture 3: Compiler structure

BY: Assist Prof. Dr. Ielaf O Abdul Majjed

Lexical Analysis

Lexical analysis is the first phase of a compiler. It takes the modified source code from language preprocessors that are written in the form of sentences. The lexical analyzer breaks these syntaxes into a series of tokens, by removing any whitespace or comments in the source code. The main task of lexical Analyzer is to read the input characters and produce a sequence of tokens such as names, keywords, punctuation marks etc.. For syntax analyzer.



The Lexical analyzer is divided in to two parts:

1. Scanning (Scanner).
2. Lexical analysis.

Scanner: Is responsible for doing a simple task like stripping out from the source program, Comments and whites pace while lexical analysis does the more complex tasks.

Tokens, Patterns, Lexemes

In general, there is a set of strings in the input for which the same token is produced as output. This set of strings is described by a rule called a pattern associated with the token. The pattern is said to match each string in the set. A lexeme is a sequence of characters in the source program that is matched by the pattern for a token.

Token: Token is a sequence of characters that can be treated as a single logical entity. Typical tokens are,

1) Identifiers 2) keywords 3) operators 4) special symbols 5) constants

Pattern: A set of strings in the input for which the same token is produced as output. This set of strings is described by a rule called a pattern associated with the token.

Lexeme: A lexeme is a sequence of characters in the source program that is matched by the pattern for a token.

Example: Const pi = 3.1416   The sub string pi is a lexeme for the token "identifier"

Types of token:-

1. Identifiers (Variables)
2. Constants
3. Operators (Operation) +, -, *, /, ^.
4. Relational <, >, !=, ≤, ≥, =.
5. Special characters ($, ;, {, }, …)
6. Key words (for, int, if, …)

Token	Sample Lexemes	Informal Description of pattern
const	const	Const
If	If	If
Relation	< , > , <= , = , < > , >=	< or > or <= or = or < > or >=
Id	Pi , count , D2, X, Y, Z	Letter followed by letters and digits
Num	3 , 1416 , 0 , 6.02E23	Any numeric constant
Literal	“ core “	Any characters between “ and “ except”

In many programming languages, the following classes cover most or all of the tokens:

1. One token for each keyword. The pattern for a keyword is the same as the keyword itself.
2. Tokens for the operators, either individually or in classes such as the token comparison.
3. One token representing all identifiers.
4. One or more tokens representing constants, such as numbers and literal.
5. Tokens for each punctuation symbol, such as left and right parentheses, comma, and semicolon.

Regular expressions

The set of all integer constants or the set of all variable names are sets of strings, where the individual letters are taken from a particular alphabet. Such a set of strings is called a language. For integers, the alphabet consists of the digits 0-9 and for variable names the alphabet contains both letters and digits (and perhaps a few other characters, such as underscore). Given an alphabet, we will describe sets of strings by regular expressions, an algebraic notation that is compact and easy for humans to use and understand. **The idea is that regular expressions** that describe simple sets of strings can be combined to form regular expressions that describe more complex sets of strings.

When talking about regular expressions, we will use the letters (*r, s and t*) in **italics** to denote unspecified regular expressions.

When letters stand for themselves (i.e., in regular expressions that describe strings that use these letters) we will use typewriter font, e.g., a or b.

Hence, when we say, e.g., “The regular expression s” we mean the regular expression that describes a single one-letter string “s”, but when we say “The regular expression s”, we mean a regular expression of any form which we just happen to call s. We use the notation L(s) to denote the language (i.e., set of strings) described by the regular expression s.

For example, **L(a) is the set {“a”}.**

Table below, shows the constructions used to build regular expressions and the languages they describe:

_ A single letter describes the language that has the one-letter string consisting of that letter as its only element.

Regular Expression	Language (set of strings)	Informal description
a	{“a”}	The set consisting of the one letter string “a”.
ε	{“ ”}	The set containing the empty string.
s|t	L(s) ∪ L(t)	Strings from both languages.
st	{vw | v ∈ L(s), w ∈ L(t)}	Strings constructed by concatenating a string from the first language with a string from the second language. Note: In set-formulas, “|” is not a part of a regular expression, but part of the set-builder notation and reads as “where”.
s*	{“ ”}∪{vw | v ∈ L(s), w ∈ L(s*)}	Each string in the language is a concatenation of any number of strings in the language of s.

- The symbol **ε** (the Greek letter epsilon) describes the language that consists solely of the empty string.
- s|t (pronounced “s or t”) describes the union of the languages described by s and t
- st (pronounced “s t”) describes the concatenation of the languages L(s) and L(t), i.e., the sets of strings obtained by taking a string from L(s) and putting this in front of a string from L(t).

For example, if L(s) is {“a”, “b”} and L(t) is {“c”, “d”},

Then L(st) is the set {“ac”, “ad”, “bc”, “bd”}.

- The language for s* (pronounced “s star”) is described recursively It consists of the empty string plus whatever can be obtained by concatenating a string from L(s) to a string from L(s*). This is equivalent to saying that L(s*) consists of strings that can be obtained by concatenating zero or more (possibly different) strings from L(s).

for example, L(s) is {“a”, “b”}

then L(**s***) is {“ ” , “a”, “b”, “aa”, “ab”, “ba”, “bb”, “aaa”...}, i.e., any string (including the empty) that consists entirely of as* and bs*.

Precedence rules:

When we combine different constructor symbols, e.g., in the regular expression a|ab*, it is not a priori clear how the different sub expressions are grouped. We can use parentheses to make the grouping of symbols explicit such as in (a|(ab))*. Additionally, we use precedence rules, similar to the algebraic convention that 3+4*5 means 3 added to the product of 4 and 5 and not multiplying the sum of 3 and 4 by 5. For regular expressions, we use the following conventions:* binds tighter than concatenation, which binds tighter than alternative (|). The example a|ab* from above, hence, is equivalent to a|(a(b*)). The | operator is associative and commutative (as it corresponds to set union, which has these properties). Concatenation is associative (but obviously not commutative) and distributes over.

Some algebraic properties for regular expression

(r|s)|t = r|s|t = r|(s|t)	associative is |
s|t = t|s	commutative is |
s|s = s	idempotent is |
s? = s| ε = s	definition by
(rs)t = rst = r(st)	associative is concatenation
s ε = s = εs	ε is a neutral element for concatenation
r(s|t) = rs|rt = (rs)|(rt)	concatenation distributes over |
(r|s)t = rt|st = (rt)|(st)	concatenation distributes over |
s*s* = s*	0 or more twice is still 0 or more

Here are a few examples of other typical programming language elements:

❖ **Keywords**

A keyword like if is described by a regular expression that looks exactly like that keyword,
e.g., the regular expression **if** (which is the concatenation of the two regular expressions i and f).

(letter)(letter)* → **[a–z, A–Z][a–z, A–Z]***

❖ **Variable names**

In the programming language C, a variable name consists of letters, digits and the underscore symbol and it must begin with a letter or underscore. This can be described by the regular expression

[a–z, A–Z, _][a–z, A–Z, _, 0–9]*.

❖ **Integers**

An integer constant is an optional sign followed by a non-empty sequence of digits:

[+–]?[0–9]$^{+}$

In some languages, the sign is a separate symbol and not part of the constant itself. This will allow whitespace between the sign and the number, which is not possible with the above

❖ Floats

[+–] ? (([0–9]+ ((.[0–9]*) ? | .[0–9]+) ([eE] [+–] ? [0–9]+)?))

A floating-point constant can have an optional sign.

After this, the mantissa part is described as a sequence of digits followed by a decimal point and then another sequence of digits. Either one (but not both) of the digit sequences can be empty.

❖ String constants

“([a–z, A–Z, 0–9] | \ [a–z, A–Z])*”

A string constant starts with a quotation mark (“) followed by a sequence of symbols and finally another quotation mark (”).