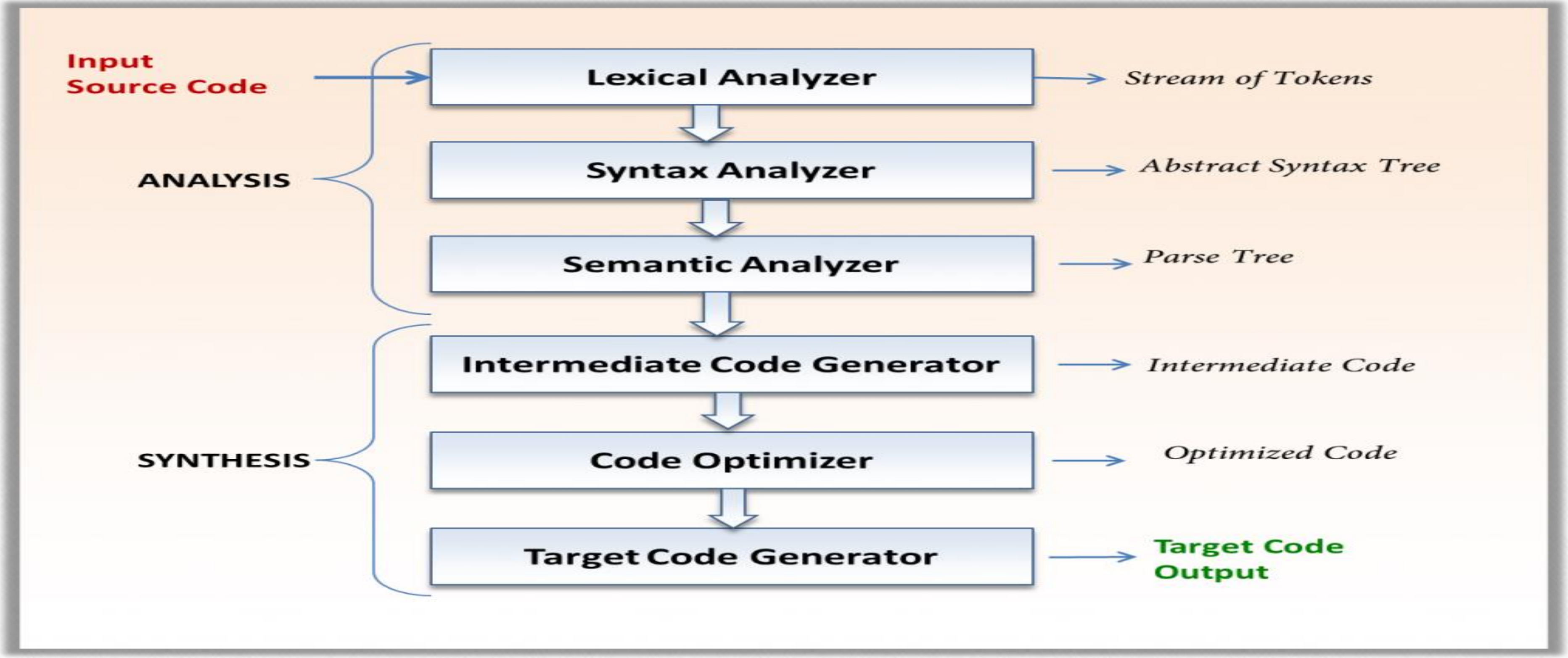
Flex

Presented By:
Dola Das
Dipannita Biswas

Phases of Compiler



Flex

- **Lexical Analysis**: Dividing the input into meaningful units. For a C program the units are variables, constants, keywords, operators, punctuation etc. These units also called as tokens. (we will use **Flex**)
- **Flex: A fast Lexical Analyzer Generator**
- **Parsing**: Involves finding the relationship between input tokens. For a C program, one needs to identify valid expressions, statements, blocks, procedures etc. (Use **Bison**)

Flex

- Flex takes a program written in a combination of Flex and C, and it writes out a file (called lex.yy.c) that holds a definition of function yylex(), with the following prototype.

```
int yylex(void)
```

The file to read

- yylex reads from the file stored in variable yyin:
- FILE* yyin;
- It is up to you to open a file for reading and store it into yyin before you call yylex.
- Each time your program calls yylex, it returns the next token (an integer token code).

The file to read

- When yylex is finished, it call function yywrap(). *If yywrap() returns 1, then yylex returns 0 to its caller*. That means "end of file".
- If yywrap returns 0, then yylex assumes that you have stored a different file into yyin, and it starts reading that file.

Flex Pattern Matching

Metacharacter	Matches
.	any character except newline
\n	newline
*	zero or more copies of the preceding expression
+	one or more copies of the preceding expression
?	zero or one copy of the preceding expression
^	beginning of line
$	end of line
a|b	a or b
(ab)+	one or more copies of ab (grouping)
"a+b"	literal "a+b" (C escapes still work)
[ ]	Character class

Flex RE

RE	Matches
s	string s literally
\c	character c; literally, where c would normally be a lex operator
[s]	character class
[~s]	characters not in character class
[s-t]	range of characters
s?	s occurs zero or one time

Flex RE

RE	Matches
s*	zero or more occurrences of s
s+	one or more occurrences of s
r|s	r or s
(s)	grouping
$	end of line
r{2,5}	anywhere from two to five r's
r{2,}	two or more r's
r{4}	exactly 4 r's

Flex Example

RE	Matches
a(b|c)d	abd or acd
^start	beginning of line with then the literal characters start
END$	the characters END followed by an end-of-line.
(s)	grouping
$	end of line
s/r	s iff followed by r (not recommended) (r is *NOT*consumed)
s{m,n}	m through n occurrences of s

Flex Example

RE	Matches
a*	zero or more a's
.*	zero or more of any character except newline
.+	one or more characters
[a-z]	a lowercase letter
[a-zA-Z]	any alphabetic letter
[^a-zA-Z]	any non-alphabetic character
a.b	a followed by any character followed by b
rs|tu	rs or tu

Flex Example

- Difference among [a-z], [-az] and [-a-z]
- Are those same: A+ and AA*

Flex Function Example

- **yywrap()** - wraps the above rule section
- **yyin** - takes the file pointer which contains the input
- **yylex()** - this is the main flex function which runs the Rule Section
- **yytext** - is the text in the buffer

yywrap()

- Function yywrap is called by lex when input is exhausted. When the end of the file is reached the return value of yywrap() is checked.
- If it is non-zero, scanning terminates and if it is 0 scanning continues with next input file.

Flex Example

RE	Matches
[:alpha:]	[a-zA-Z]
[:digit:]	[0-9]
[:alnum:]	[a-zA-Z0-9]
[:lower:]	[a-z]
[:upper:]	[A-Z]
[:punct:]	All punctuation marks
[:space:]	space

Flex Example

- The regular expressions listed above are grouped according to precedence, from highest precedence at the top to lowest at the bottom. Those grouped together have equal precedence.
- For example,

  foo|bar* matches ????

- (foo)|(bar)*, fo(o|b)ar*, (foo|bar)*, (foo)|((ba)r*)

Flex Example

- Since the ’*’ operator has higher precedence than concatenation, and concatenation higher than alternation (’|’). This pattern therefore matches either the string "foo" or the string "ba" followed by zero-or-more r’s.
- To match "foo" or zero-or-more "bar"’s, use: **foo|(bar)***
- To match zero-or-more "foo"’s-or-"bar"’s: **(foo|bar)***

Flex – Function

- The **yyless(n)** function, accepts n characters of the token and then they will be re-scanned for finding the next match. It basically keeps reducing n characters and returns the string for re-scanning .
- The **yymore()** function will output yytext, when the action part of any rule which has yymore() is finished. It basically outputs the matched input only after the rule has been executed.

Flex - Function

- The **yy_flush_buffer()** function flushes out the first two characters of the token by setting it to NULL(‘\0’) character.
- The **yyterminate()** function ends the execution of the program as soon as it is called.
- The **unput()** function reads the characters and basically replaces those characters.

Flex - Function

- The **input()** function reads the characters and makes them unavailable to the scanner. It is basically like truncating those characters. In the program below, if we find a string starting with comment '/*' the input() function will read characters till we find '*/' and will not show these characters/.

How the input is matched

- If it finds more than one match, it takes the one matching the most text.
- If it finds two or more matches of the same length, the rule listed first in the flex input file is chosen.
- Once the match is determined, the text corresponding to the match (called the token) is made available in the global character pointer **yytext**, and its length in the global integer **yyleng**.

How the input is matched

- If no match is found, then the default rule is executed: the next character in the input is considered matched and copied to the standard output. Thus, the simplest legal flex input is:

```
%%
```

- which generates a scanner that simply copies its input (one character at a time) to its output.

How the input is matched

- Note that **yytext** can be defined in two different ways: either as a **character pointer** or as a **character array**.
- You can control which definition flex uses by including one of the special directives '%pointer' or '%array' in the first (definitions) section of your flex input.
- The default is '%pointer',

Actions

- Each pattern in a rule has a corresponding action, which can be any arbitrary C statement.
- The pattern ends at the first non-escaped whitespace character; the remainder of the line is its action.
- If the action is empty, then when the pattern is matched the input token is simply discarded.

Actions

- If the action contains a ’{’, then the action spans till the balancing ’}’ is found, and the action may cross multiple lines.
- Actions can include arbitrary C code, including return statements to return a value to whatever routine called ‘yylex()’.
- Each time ‘yylex()’ is called it continues processing tokens from where it last left off until it either reaches the end of the file or executes a return.

Actions

- There are a number of special directives which can be included within an action:
  - **ECHO** copies yytext to the scanner’s output.
  - **BEGIN** followed by the name of a start condition places the scanner in the corresponding start condition.
  - **REJECT** directs the scanner to proceed on to the "second best" rule which matched the input (or a prefix of the input).

ECHO & REJECT

- ECHO statement copies the token matched to the output.
- Whenever REJECT statement is executed, the last letter will be treated with the matched token and will continue with the prefixed input for the next action or rule.

Another application of yymore()

- ‘**yymore()**’ tells the scanner that the next time it matches a rule, the corresponding token should be appended onto the current value of yytext rather than replacing it. For example, given the input "mega-kludge" the following will write "mega-mega-kludge" to the output:

```
%%
mega- ECHO; yymore();
kludge ECHO;
```

- First "mega-" is matched and echoed to the output. Then "kludge" is matched, but the previous "mega-" is still hanging around at the beginning of yytext so the ‘ECHO’ for the "kludge" rule will actually write "mega-kludge".

The generated scanner

- The output of flex is the file ‘lex.yy.c’, which contains the scanning routine ‘yylex()’, a number of tables used by it for matching tokens, and a number of auxiliary routines and macros.
- By default, ‘yylex()’ is declared as follows:

```
int yylex()
{
... various definitions and the actions in here ...
}
```

The generated scanner

- Whenever ‘yylex()’ is called, it scans tokens from the global input file **yyin** (which defaults to stdin). It continues until it either reaches an EOF (at which point it returns the value 0) or one of its actions executes a return statement.
- If the scanner reaches an EOF, subsequent calls are undefined unless either yyin is pointed at a new input file (in which case scanning continues from that file), or ‘yyrestart()’ is called. ‘yyrestart()’ takes one argument, a ‘FILE *’ pointer (which can be nil, if you’ve set up YY_INPUT to scan from a source other than yyin), and initializes yyin for scanning from that file.

The generated scanner

- If ‘yylex()’ stops scanning due to executing a return statement in one of the actions, the scanner may then be called again and it will resume scanning where it left off.