

CPSC 323 Compilers and Languages

Instructor: Susmitha Padda

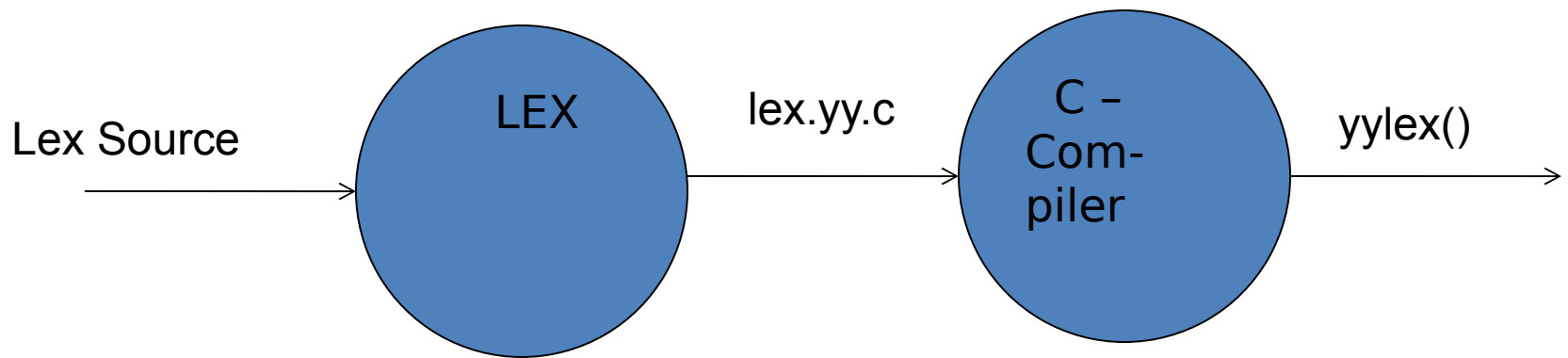
Lex & Yacc

1. Lex

2. YACC

1. Lex

- **lex is a program (generator) that generates lexical analyzers, (widely used on Unix).**
- **Window version is called Flex**
- **It reads the input stream (specifying the lexical analyzer) and outputs source code implementing the lexical analyzer in the C programming language.**



The Lex source (3 sections)

%%

Token Definitions section

%%

Tokens & action section

%%

User Defined section (subroutines)

1.1 Token Definition section

Elementary Expressions

- Use of REs

- $\Sigma = \{\text{all alphabets, digits, special chars, etc}\}$

Excepts some special chars such as,

\$, ^, [,] - ? Etc -> in that case use “ ” sign.

- concatenation
- union $a | b | c | \dots | z$ or $[a..z]$, $[a-d0-9]$
- Kleene Closure $[a-z]^*$
- + $[a-z0-9]^+$

Lex

- Pattern Matching examples.**

Expression	Matches
abc	abc
abc*	ab abc abcc abccc ...
abc+	abc abcc abccc ...
a(bc)+	abc abcbcb abcbcbcb ...
a(bc)?	a abc
[abc]	one of: a, b, c
[a-z]	any letter, a-z
[a\ -z]	one of: a, -, z
[-az]	one of: -, a, z
[A-Za-z0-9]+	one or more alphanumeric characters
[\t\n]+	whitespace
[^ab]	anything except: a, b
[a b]	one of: a, , b
a b	one of: a, b

Example of Token Definition

%%

letter [a-zA-Z]

digit [0-9]

ident {letter}({letter} | {digit}) *

Real {digit}+ "." {digit}+

op "+" | "-" | "*" | "/"

ws [\n \t] +

1.2 Tokens and action section

This section describes, what to do with it once a token is recognized (actions).

e.g.,

{newline}	linecount++
{integer}	{ printf (“ found integer”);
	return INTEGER
	}

1.3 user defined section

section where a user can define own functions that can be used I previous sections.

Lex example:

letter [a-zA-Z]

digit [0-9]

ident {letter}({letter} | {digit}) *

op "+", "-", "*", "/", "(", ")", ",",

ws [\n t]+

%%

{ws} ;

{ident} { yyval = install (yytext);
return ID; }

{op} return yytext[0];

%%

main()

{ }

Int install (char *ident) {

/* install the identifier into the symbol table */

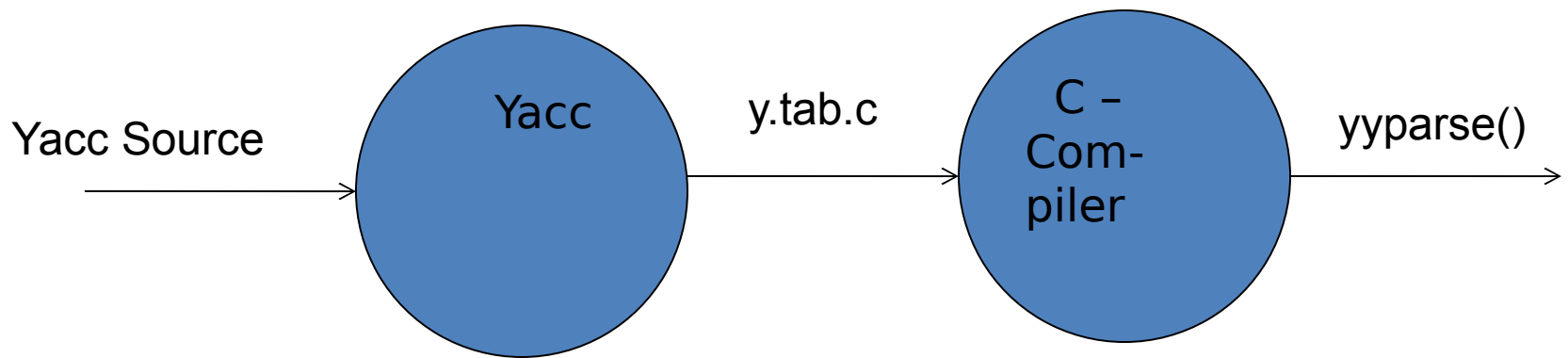
}

Lex predefined variables. – stats with yy....

Name	Function
<code>int yylex(void)</code>	call to invoke lexer, returns token
<code>char *yytext</code>	pointer to matched string
<code>yylen</code>	length of matched string
<code>yyval</code>	value associated with token
<code>int yywrap(void)</code>	wrapup, return 1 if done, 0 if not done
<code>FILE *yyout</code>	output file
<code>FILE *yyin</code>	input file

2. Yacc

- Yacc reads the grammar and generate C code for a parser
- Grammars written in Backus Naur Form (BNF) .
- BNF grammar used to express *context-free languages* .
- This known as *bottom-up or shift-reduce parsing* .
(*LALR parser*)



Yacc source
(consists of 3 sections)

Definition & Declaration section

%%

Syntax rules & Action section

%%

User defined section

2.1 definition & declaration section

Define or declare variables, tokens etc

e.g.,

%token ID

%token INTEGER

%left '+' '-' /* operator associativity */

%start expr /* the starting symbol */

2.2 syntax rules & action sections

Rules are defined similar to the BNF notation

2.3 user-defined section

Define own functions to be used in the Yacc source

Yacc Example

```
%token ID
%start expr
%%

expr : expr '+' term    { $$ = $1 + $3 }
    | expr '-' term    { $$ = $1 - $3 }
    | term              { $$ = $1 }
    ;

term : term '*' factor  { $$ = $1 * $3 }
    | term '/' factor  { $$ = $1 / $3 }
    | factor;          { $$ = $1 }
    ;

Factor : '(' expr ')'   { $$ = $2 }
    | ID               { $$ = $1 }
    ;

%%

main () { ..... }
```

