Lex & Yacc

References

***Tom Niemann. “A Compact Guide to Lex & Yacc ”. Portland, Oregon. 18 April 2010 <http://epaperpress.com>**

***Levine, John R., Tony Mason and Doug Brown [1992]. Lex & Yacc. O’Reilly & Associates, Inc. Sebastopol, California.**

Lexical Analysis

- What do we want to do? Example:

```
if (i == j)
     z = 0;
else
     z = 1;
```

- The input is just a sequence of characters:

```
if (i == j)\n\tz = 0;\nelse\n\tz = 1
```

- **Goal:** Partition input strings into substrings
  - And classify them according to their role

Program Elements

Lexical Analysis:

Lexical analyzer: scans the input stream and converts sequences of characters into tokens.

Token: a classification of groups of characters.

Examples:

Lexeme	Token
`Sum`	`ID`
`for`	`FOR`
`=`	`ASSIGN_OP`
`==`	`EQUAL_OP`
`57`	`INTEGER_CONST`
`“Abcd”`	`STRING_CONST`
`*`	`MULT_OP`
`,`	`COMMA`
`:`	`SEMICOLUMN`
`(`	`LEFT_PAREN`

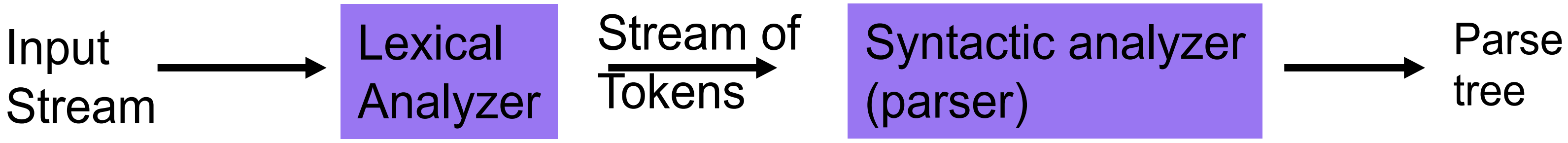
Lex is a tool for writing lexical analyzers.

Syntactic Analysis (Parsing):

Parser: reads tokens and assembles them into language constructs using the grammar rules of the language.

Yacc (Yet Another Compiler Compiler) is a tool for constructing parsers.

Lexical and syntactic analysis



Input Stream → Lexical Analyzer → Stream of Tokens → Syntactic analyzer (parser) → Parse tree

- **Lexical analyzer:** scans the input stream and converts sequences of characters into tokens.

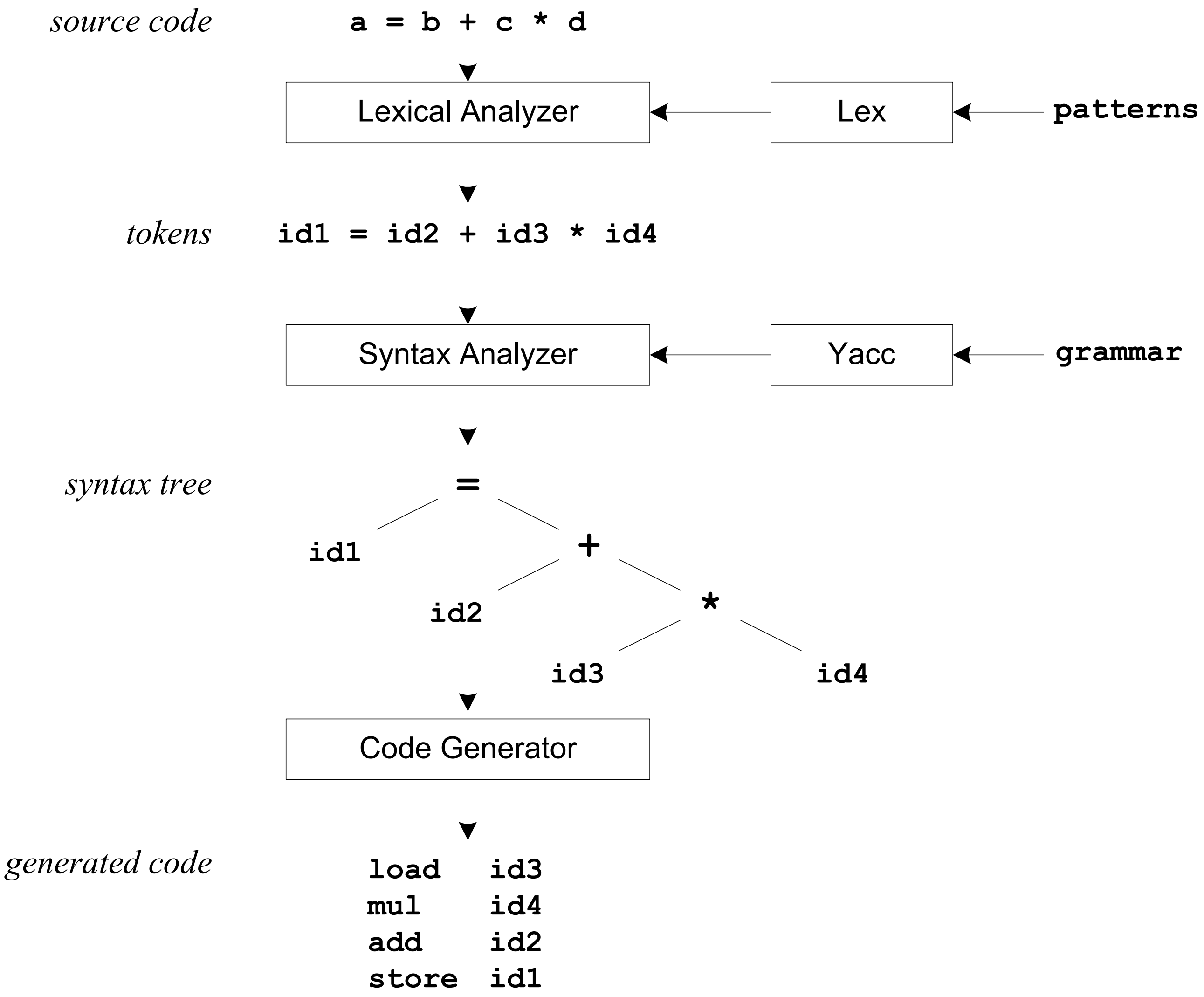
  (char list) → (token list)

- **`Lex`** is a tool for writing lexical analyzers.

- **Syntactic Analysis:** reads tokens and assembles them into language constructs using the grammar rules of the language.

- **`Yacc`** is a tool for constructing parsers.

Code Translation



```
load   id3
mul    id4
add    id2
store  id1
```

Figure 1: Compilation Sequence

Lex − A Lexical Analyzer Generator

M. E. Lesk and E. Schmidt

Bell Laboratories
Murray Hill, New Jersey 07974

ABSTRACT

Lex helps write programs whose control flow is directed by instances of regular expressions in the input stream. It is well suited for editor-script type transformations and for segmenting input in preparation for a parsing routine.

Lex source is a table of regular expressions and corresponding program fragments. The table is translated to a program which reads an input stream, copying it to an output stream and partitioning the input into strings which match the given expressions. As each such string is recognized the corresponding program fragment is executed. The recognition of the expressions is performed by a deterministic finite automaton generated by Lex. The program fragments written by the user are executed in the order in which the corresponding regular expressions occur in the input stream.

The lexical analysis programs written with Lex accept ambiguous specifications and choose the longest match possible at each input point. If necessary, substantial lookahead is performed on the input, but the input stream will be backed up to the end of the current partition, so that the user has general freedom to manipulate it.

Lex can generate analyzers in either C or Ratfor, a language which can be translated automatically to portable Fortran. It is available on the PDP-11 UNIX, Honeywell GCOS, and IBM OS systems. This manual, however, will only discuss generating analyzers in C on the UNIX system, which is the only supported form of Lex under UNIX Version 7. Lex is designed to simplify interfacing with Yacc, for those with access to this compiler-compiler system.

Example

- Recall:

```
if (i == j)\n\tz = 0;\nelse\n\tz = 1;
```

- Token-lexeme pairs returned by the lexer:
  - <Keyword, “**if**”>
  - <Whitespace, “ ”>
  - <OpenPar, “ **(**”>
  - <Identifier, “**i**”>
  - <Whitespace, “ ”>
  - <Relation, “**==**”>
  - <Whitespace, “ ”>
  - ...

Implementation of A Lexical Analyzer

The lexer usually discards **uninteresting** tokens that don't contribute to parsing.

Examples: Whitespaces, Comments

– Exception: which language cares about whitespaces?

The goal is to partition the string. That is implemented by reading left-to-right, recognizing one token at a time.

Lexical structure described can be specified using ***regular expressions*.**

Lex and Yacc

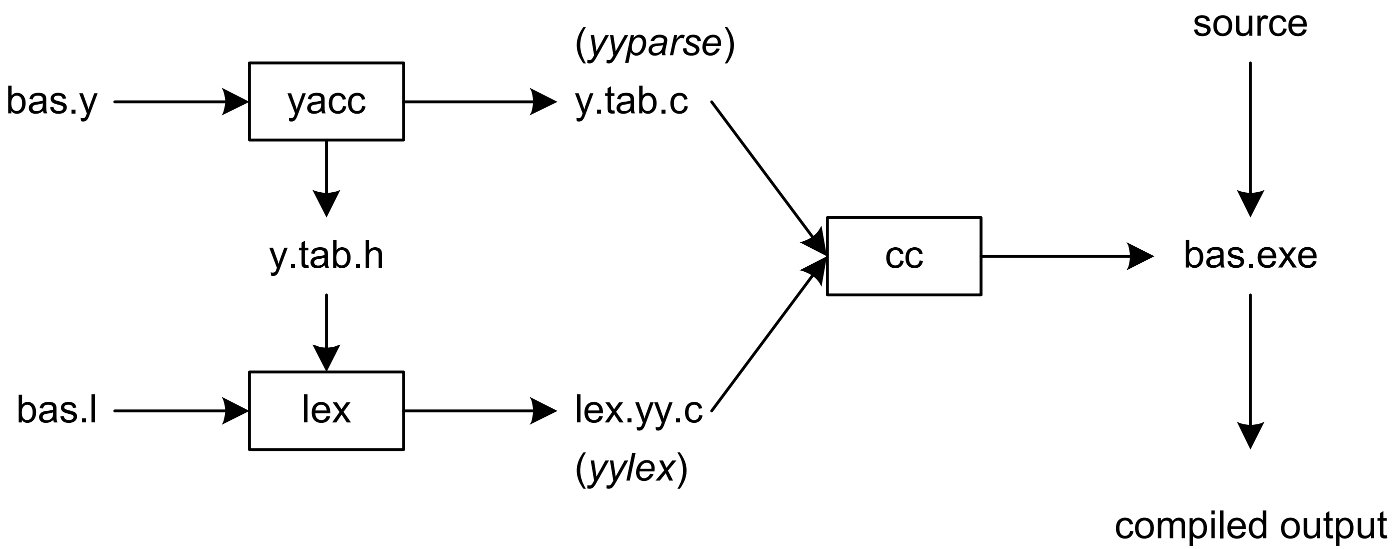
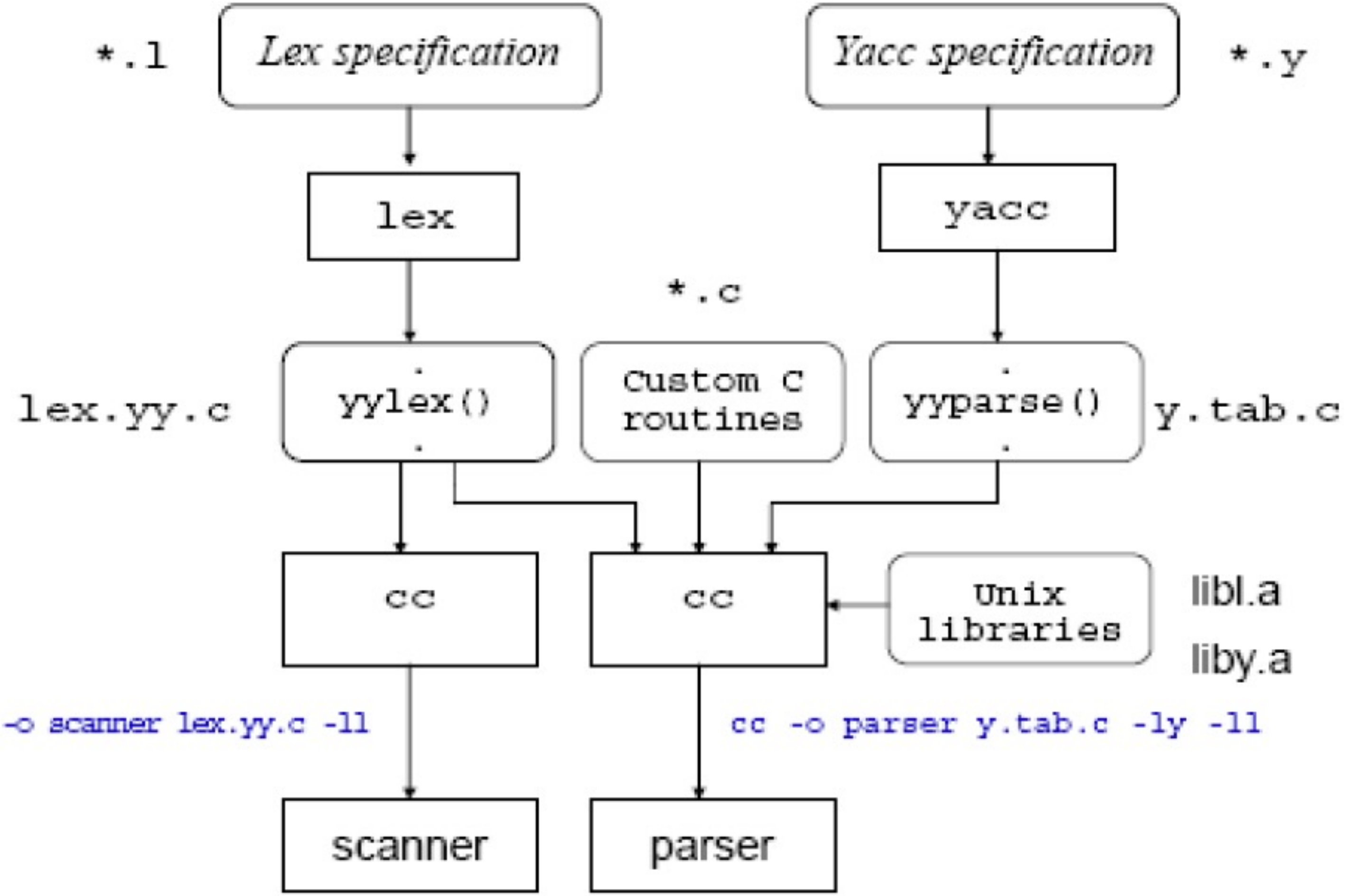


Figure 2: Building a Compiler with Lex/Yacc

Using `lex` and `yacc` tools



```
cc -o scanner lex.yy.c -ll
```

```
cc -o parser y.tab.c -ly -ll
```

Using **lex**

- Contents of a **lex** program:

```
Declarations
%%
Translation rules
%%
Auxiliary functions
```

- The declarations section can contain declarations of variables, manifest constants, and regular definitions. The declarations section can be empty.

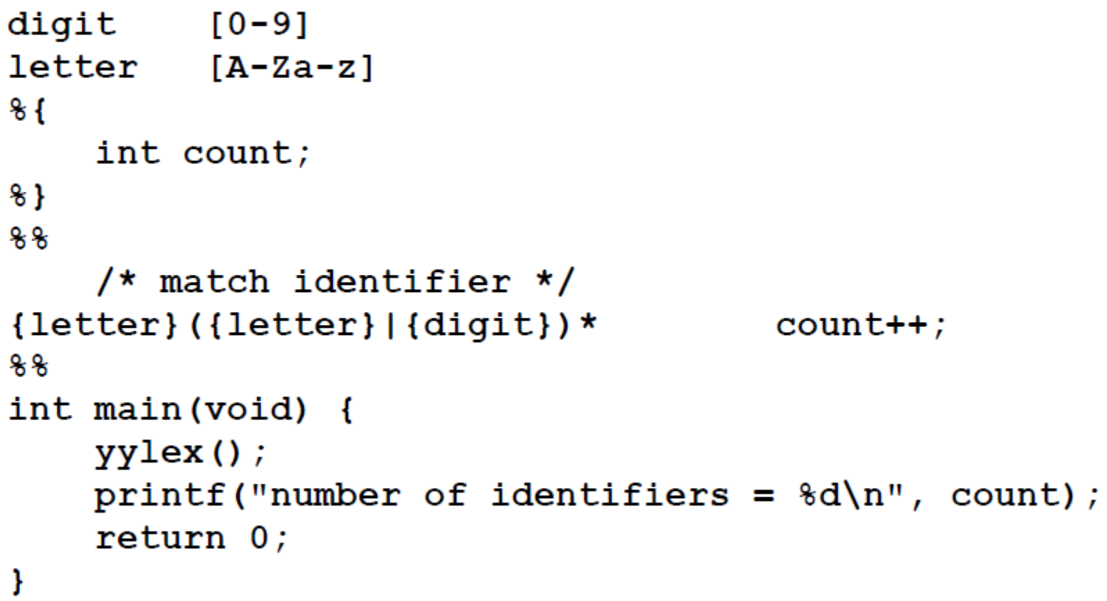
- The translation rules are each of the form

  ```
  pattern {action}
  ```

  - Each pattern is a regular expression which may use regular definitions defined in the declarations section.
  - Each action is a fragment of C-code.

- The auxiliary functions section starting with the second `%%` is optional. Everything in this section is copied directly to the file **`lex.yy.c`** and can be used in the actions of the translation rules.

Lex



```
digit    [0-9]
letter   [A-Za-z]
%{
    int count;
%}
%%
    /* match identifier */
{letter}({letter}|{digit})*        count++;
%%
int main(void) {
    yylex();
    printf("number of identifiers = %d\n", count);
    return 0;
}
```

- Whitespace must separate the defining term and the associated expression.
- Code in the definitions section is simply copied as-is to the top of the generated C file and must be bracketed with “%{“ and “%}” markers.
- substitutions in the rules section are surrounded by braces ({letter}) to distinguish them from literals.

Running `lex`

On Unix system

```
$ lex mylex.l
```

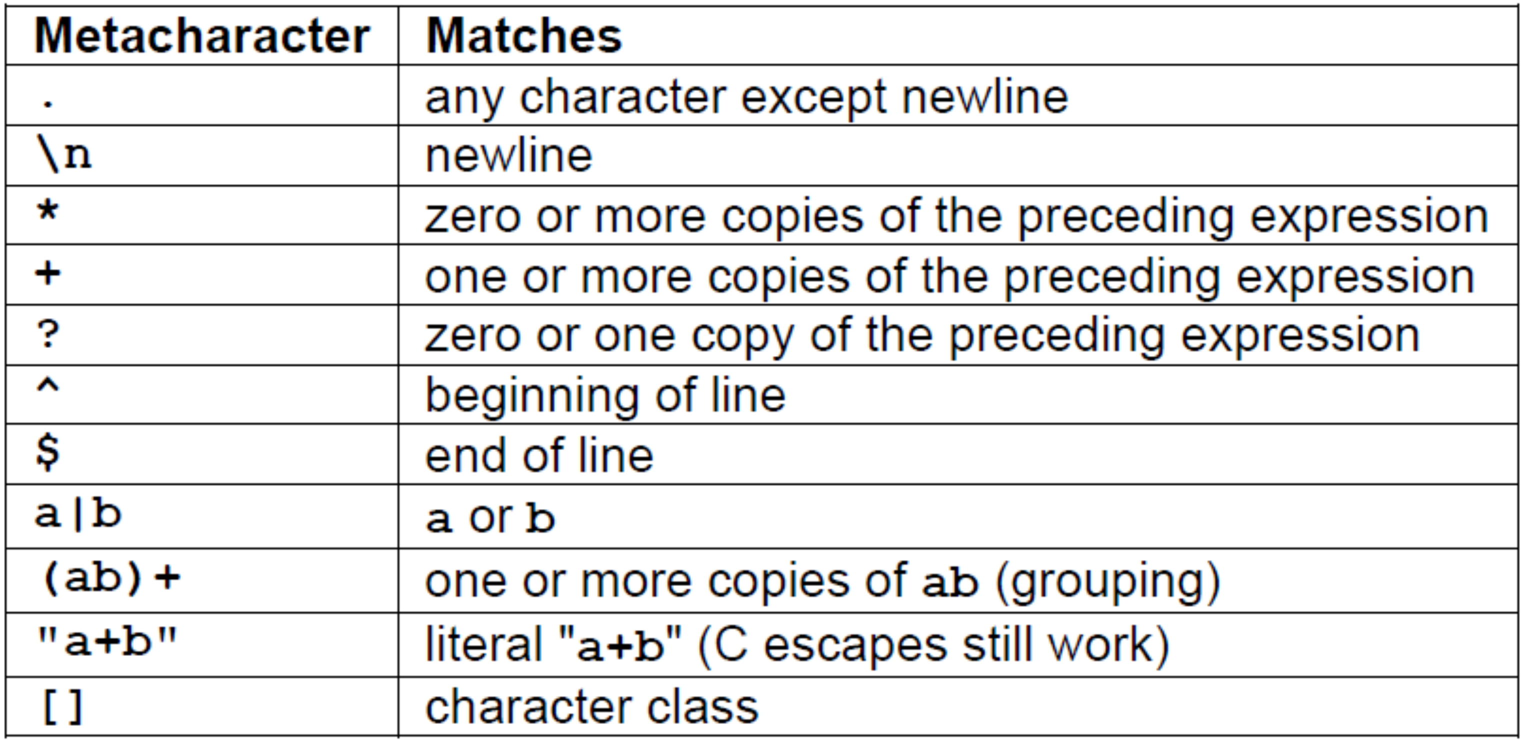
it will create `lex.yy.c`

then type

```
$ gcc -o mylex lex.yy.c -lfl
```

The open-source version of `lex` is called "`flex`"

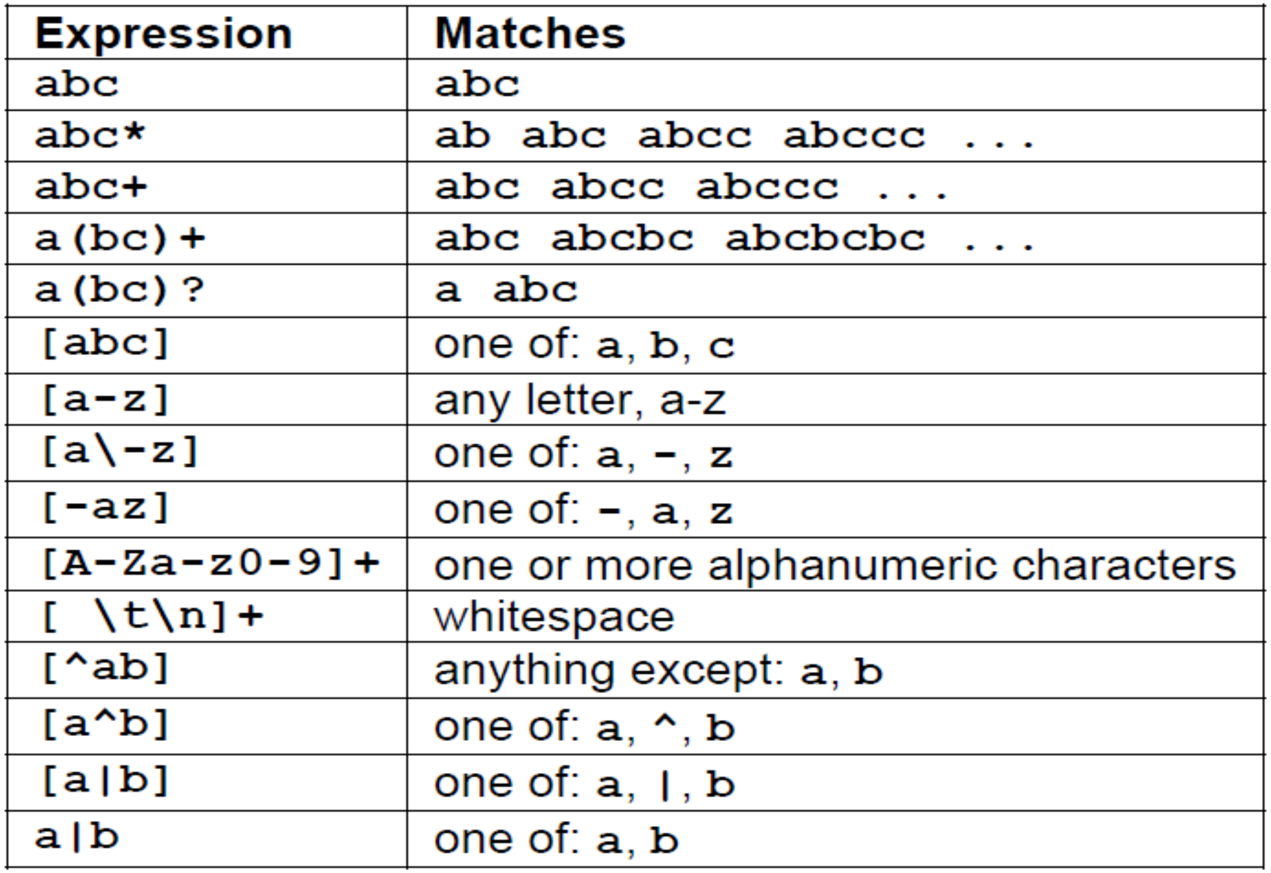
Lex



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

Pattern Matching Primitives

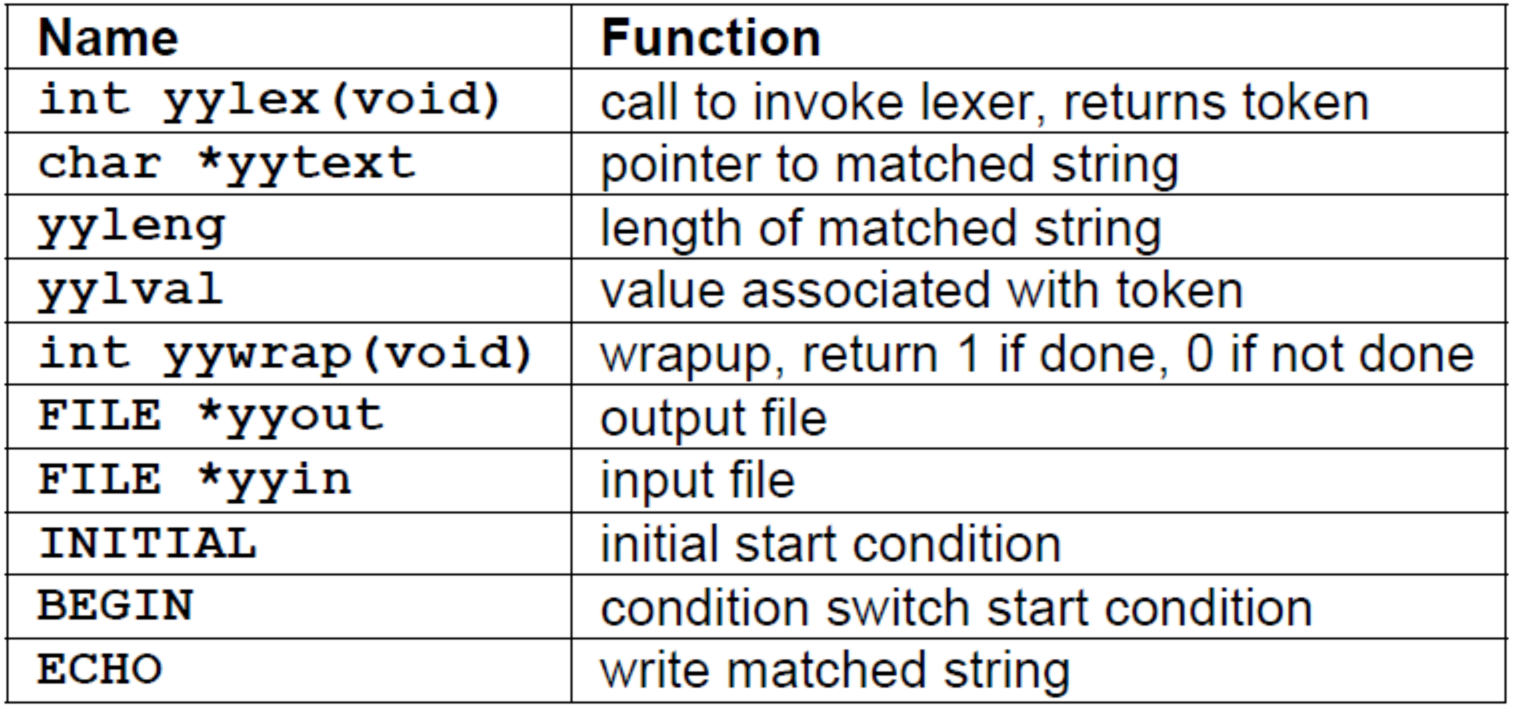
Lex



Expression	Matches
abc	abc
abc*	ab abc abcc abccc ...
abc+	abc abcc abccc ...
a(bc)+	abc abcbc abcbcbc ...
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
a|b	one of: a, b

- Pattern Matching examples.

Lex



Name	Function
`int yylex(void)`	call to invoke lexer, returns token
`char *yytext`	pointer to matched string
`yyleng`	length of matched string
`yylval`	value associated with token
`int yywrap(void)`	wrapup, return 1 if done, 0 if not done
`FILE *yyout`	output file
`FILE *yyin`	input file
`INITIAL`	initial start condition
`BEGIN`	condition switch start condition
`ECHO`	write matched string

Lex predefined variables.