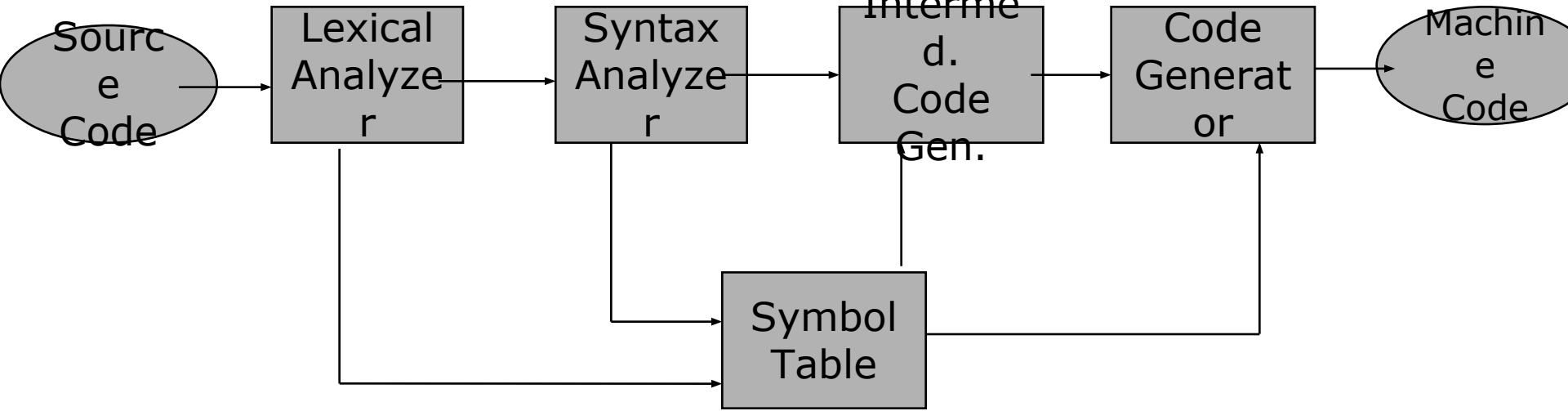
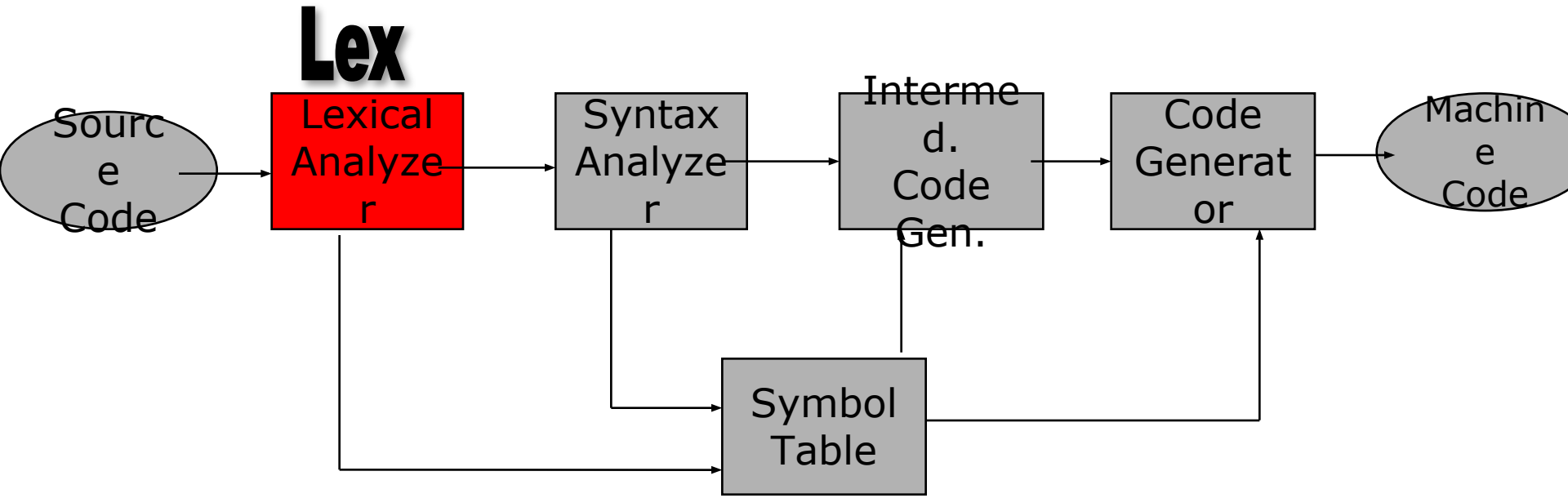
Lex

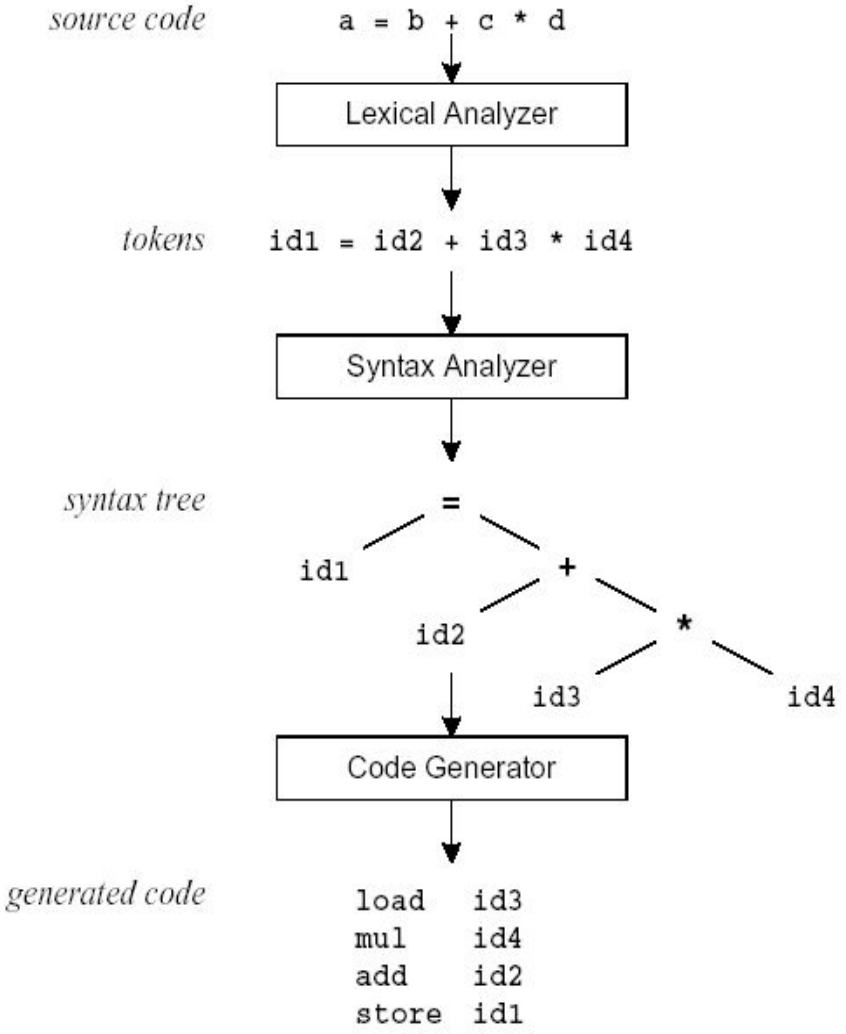
Introduction



Introduction



Introduction



Introduction

- Lex is a program (generator) that generates lexical analyzers, (widely used on Unix).
- It is mostly used with Yacc parser generator.
- Written by Eric Schmidt and Mike Lesk.
- It reads the input stream (specifying the lexical analyzer) and outputs source code implementing the lexical analyzer in the C programming language.
- Lex will read patterns (regular expressions); then produces C code for a lexical analyzer that scans for identifiers.

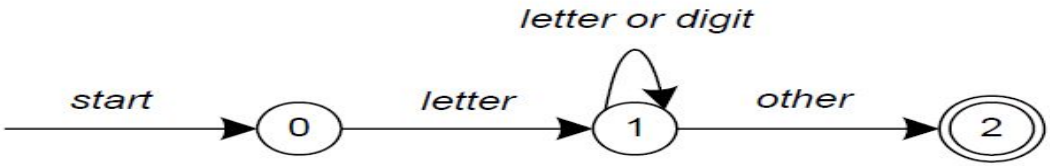
Introduction

- A simple pattern: letter(letter|digit)*

- Regular expressions are translated by lex to a computer program that mimics an FSA.

- This pattern matches a string of characters that begins with a single letter followed by zero or more letters or digits.

Introduction




```
start:  goto state0

state0: read c
        if c = letter goto state1
        goto state0

state1: read c
        if c = letter goto state1
        if c = digit goto state1
        goto state2

state2: accept string
```

Introduction

- A ***lexical analyzer*** *(scanner)* is to break up an input stream into *tokens*

```
t = u * v + x;
```

ID ASSIGN ID MULT ID PLUS ID SEMI

- Lex is an utility to help you rapidly generate your scanners

FLEX

- Flex is a tool for generating scanners.
- Flex source is a table of regular expressions and corresponding program fragments.
- Generates *lex.yy.c* which defines a routine *yylex()*

FLEX: Template

- The flex input file consists of three sections, separated by a line with just %% in it:

```
definitions
%%
rules
%%
user code
```

LEX: Definitions

- The definitions section contains declarations of simple name definitions to simplify the scanner specification.
- Name definitions have the form:

```
name definition
```

- Example:

```
DIGIT    [0-9]
ID       [a-z][a-z0-9]*
```

LEX: Rules

- The rules section of the flex input contains a series of rules of the form:

```
pattern action
```

- Example:

```
{ID} printf( "An identifier: %s\n", yytext );
rule printf( "A rule is: %s\n", yytext );
{ID} ;
```

- The ***yytext*** and ***yylength*** variable.
- If action is empty, the matched token is discarded

LEX: User Code

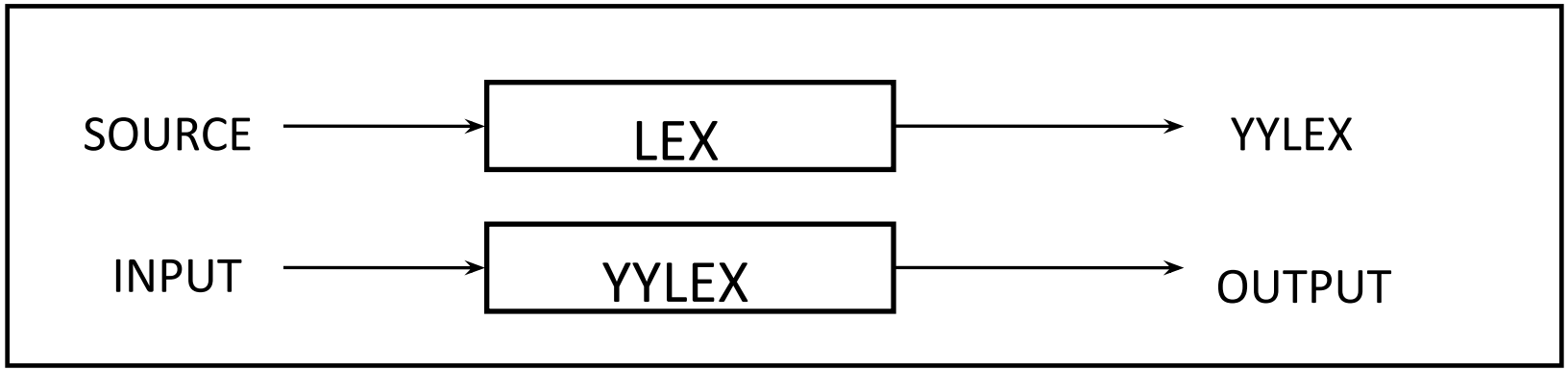
- The user code section is simply copied to *lex.yy.c* verbatim.
- The presence of this section is optional; if it is missing, the second `%%` in the input file may be skipped.
- In the definitions and rules sections, any indented text or text enclosed in `%{` and `%}` is copied verbatim to the output (with the `%{ }`'s removed).

LEX: BASIC PRINCIPLES

- When the generated scanner is run, it analyses its input string looking for strings which match any of its patterns.
- If the current input can be matched by several expressions, then the ambiguity is resolved by the following rules.
    - The longest match is preferred.
    - The rule given first is preferred.
- Once the match is determined,
    - the text corresponding to it is available in the global character pointer yytext its length is yyleng and the current line number is yylineno,
    - and the action corresponding to the matched pattern is then executed,
    - and then the remaining input is scanned for another match.

LEX

- User's expressions and actions are converted into output general-purpose language
  - yylex : generated program
  - Recognize expressions in input and perform specified actions for each



LEX: Installation

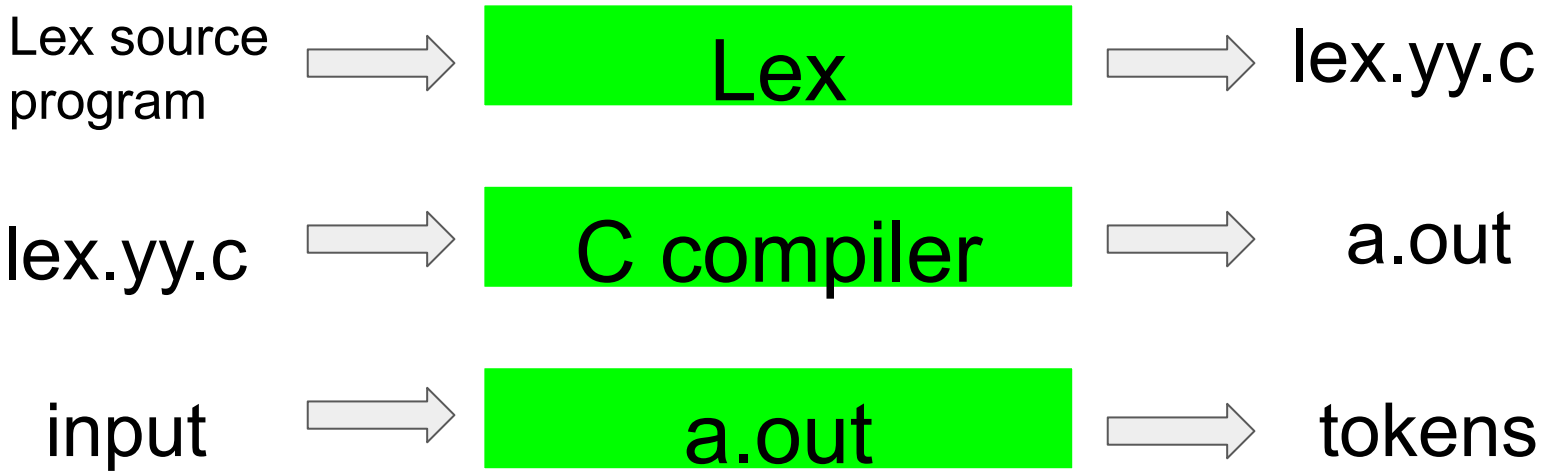
Install FLEX

Ubuntu: sudo apt-get install flex

Windows:
https://thesvgway.wordpress.com/2013/10/09/how-to-compile-run-lex-yacc-programs-on-windows/

Mac: brew install flex
http://macappstore.org/flex/
https://github.com/westes/flex

LEX


Lex source program
Lex
lex.yy.c
lex.yy.c
C compiler
a.out
input
a.out
tokens

LEX

- Run Lex:

```
lex scanner.l
```

- Will create ***lex.yy.c*** for the lexical analyzer.
- Compile lex.yy.c:

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

- Run lexical analyzer:

```
./a.out
```

LEX

```
%%
[0-9]* ECHO;
%%

---

%%
. printf("%s", yytext);
%%
```

LEX

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

Predefined Variables

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

- Blank, tab and newline ignoring

  [\t \n]+ ;

- **yytext**
  - [A-Z]* printf(“%s”,yytext);
- **ECHO**
  - [A-Z]* ECHO;

LEX

- **yyleng**
  - Count of the number of characters matched
  - Count number of words and characters in input
    - **[0-9]* printf("Number of digits = %d",yyleng);**

LEX

- yymore()
  - Next time a rule is matched, the token should be appended to current value of yytext

    iiitssricity ECHO; yymore();
- yyless(n)
  - Returns all but the first ‘n’ characters of the current token back to input stream

    iiitssricity ECHO; yyless(2);

Regular Expression

x	match the character 'x'
.	any character (byte) except newline
[xyz]	a "character class"; in this case, the pattern matches either an 'x', a 'y', or a 'z'
[abj-oZ]	a "character class" with a range in it; matches an 'a', a 'b', any letter from 'j' through 'o', or a 'Z'
[^A-Z]	a "negated character class", i.e., any character but those in the class. In this case, any character EXCEPT an uppercase letter.
[^A-Z\n]	any character EXCEPT an uppercase letter or a newline

Regular Expression

r*	zero or more r's, where r is any regular expression
r+	one or more r's
r?	zero or one r's (that is, "an optional r")
r{2,5}	anywhere from two to five r's
r{2,}	two or more r's
r{4}	exactly 4 r's
{name}	the expansion of the "name" definition (see above)
"[xyz]\"foo“	the literal string: [xyz]"foo
\X	if X is an 'a', 'b', 'f', 'n', 'r', 't', or 'v', then the ANSI-C interpretation of \x. Otherwise, a literal 'X' (used to escape operators such as '*')

Regular Expression

\0	a NUL character (ASCII code 0)
\123	the character with octal value 123
\x2a	the character with hexadecimal value 2a
(r)	match an r; parentheses are used to override precedence (see below)
rs	the regular expression r followed by the regular expression s; called "concatenation"
r|s	either an r or an s
^r	an r, but only at the beginning of a line (i.e., which just starting to scan, or right after a newline has been scanned).
r$	an r, but only at the end of a line (i.e., just before a newline). Equivalent to "r/\n".

Important Links

- Lex & Yacc, 2nd Edition. by John Levine, Doug Brown, Tony Mason. Released October 1992. Publisher(s): O'Reilly Media
- https://www.isi.edu/~pedro/Teaching/CSCI565-Fall15/Materials/LexAndYaccTutorial.pdf
- http://dinosaur.compilertools.net/flex/
- http://dinosaur.compilertools.net/lex/index.html#:~:text=Lex%20helps%20write%20programs%20whose,expressions%20and%20corresponding%20program%20fragments.
-