UNIVERSITY OF NICOSIA
ΠΑΝΕΠΙΣΤΗΜΙΟ ΛΕΥΚΩΣΙΑΣ

COMP-421 Compiler Design

Presented by
Dr Ioanna Dionysiou

Lecture Outline

- Lexical Analyzer
  - Lex
- Lex Examples

Figures and part of the lecture notes taken from
"A compact guide to lex&yacc", epaperpress.com

What is Lex?

- Lex is a tool for building lexical analyzers or lexers
- A lexer takes an arbitrary input stream and tokenizes it
  - Divides it into lexical tokens
    - Tokenized output can be furthered processed, usually by yacc
    - It can be the end product

Lex Specification

- Lex specification
  - Create a set of patterns
    - which lex matches against the input
  - When one of the patterns matches
    - the lex program invokes C code
      - you provide this code that does something with the matched text
  - Lex itself does not produce an executable program
    - it translates the lex specification into a file containing a C routine called yylex()
    - your program calls yylex() to run the lexer
    - using C compiler, you compile the file that lex produced

Lex in Compilation Sequence

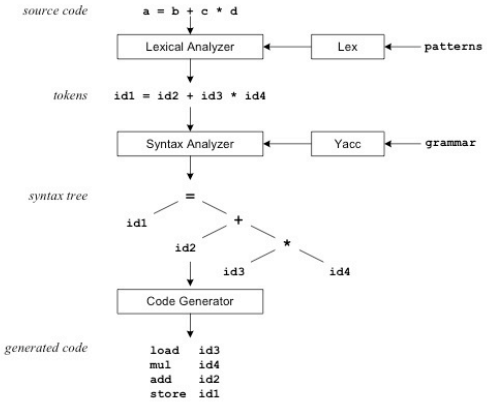

Figure 1: Compilation Sequence

Lex and Yacc

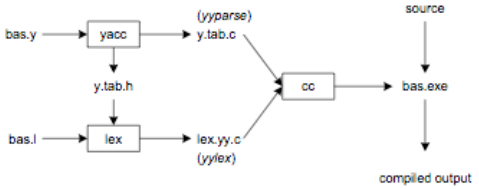

Figure 2: Building a Compiler with Lex/Yacc

```
yacc -d bas.y                  # create y.tab.h, y.tab.c
lex bas.l                      # create lex.yy.c
cc lex.yy.c y.tab.c -obas.exe  # compile/link
```


Lex Pattern Matching

- Lex is using a rich regular expression language
  - Any regular expression can be expressed as a FSA
  - Lex is using regular expressions for pattern matching
    - There are limitations though
    - Lex only has states and transitions between states
      - Lex cannot be used to recognize nested structures such as parentheses
      - Nested structures are handled by incorporating a stack
      - Yacc augments the DFA with a stack and can process those easily

Lex Pattern Matching

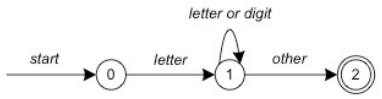

Figure 3: Finite State Automaton

```
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
```

Pattern Matching Primitives

Pattern	Matches
.	Any single character except newline
*	Zero or more occurrences of the preceding expression
+	One or more occurrences of the preceding expression
?	Zero or one occurrence of the preceding expression
"…"	Literal within the quotation marks, C escape sequences are recognized (\n, \t, …)
[ ]	Character class (matches any character within brackets). If the first character is ^ it changes the meaning to match any character except the ones within the brackets A dash - indicates a character range A dash - or bracket as the first character lets you include dashes and brackets in character classes C escape sequences with \ are recognized

Pattern Matching Primitives

Pattern	Matches
^	Beginning of line as the first character (also used for negation within square brackets
\	Used to escape sequences
|	Either the preceding expression or the following
/	Matches the preceding expression but only if followed by the following regular expression
()	Sequences of characters

Pattern Matching Examples

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`

Table 2: Pattern Matching Examples

Pattern Matching Examples

"if"

[\n\t]

.*

#.*

\/\/.*

Pattern Matching Rule

- If two patterns match the same string, the longest match wins!
  - In case both matches are the same length, then the first pattern listed is used

Lex Specification Sections

- A lex specification consists of three sections:
  - Definition section
  - Rules section
  - User subroutines section

```
... definitions ...
%%
... rules ...
%%
... subroutines ...
```

- The parts are separated by lines consisting of %%
- The first two parts are required, although a part may be empty
- The third part and the preceding %% may be omitted

Definition Section

- It can include
  - literal block
  - definitions

Definition Section - Literal Block

- The literal block in the definition section is C code bracketed by the lines %{ and %}

```
%{
C code declarations
%}
```

- C code is copied verbatim to the generated C source file near the beginning, before the beginning of yylex()
  - It usually contains
    - declarations of variables and functions used by code in the rules section,
    - #include lines for header files

Definition Section - Definitions

- Definitions (or substitutions)
  - allow you to name a regular expression (or part of it) and refer to it by name in the rules section
  - It is useful to break up complex expressions
- Definition Syntax
  - NAME expression
    - where name can contain letters, digits, underscores, and must not start with a digit. Some implementations allow hyphen as well
    - e.g. DIGIT [0-9]

Rules Section

- It contains pattern lines and C code
  - A line that starts with
    - a whitespace or material enclosed in %{ and %} is C code
  - A line that starts with
    - anything else is a pattern line

```
... definitions ...
%%
... rules ...
%%
... subroutines ...
```


C code lines

- They are copied verbatim to the generated C file
- Lines at the beginning of the section are placed near the beginning of the generated yylex() function
  - Should be
    - declarations of variables used by code associated with the patterns
    - initialization code for the scanner
  - C code lines anywhere else are copied to an unspecified place in the generated file
    - should contain only comments

Pattern lines

- They contain a pattern followed by some whitespace and C code to execute when the input matches the pattern
  - If C code is more than one statement or spans multiple lines, it must be enclosed in braces {}

Pattern lines

- It may include references to definitions with the name in braces

{DIGIT}+

Pattern Matching while Scanning input…

- Lexer runs…
  - matches the input against the patterns in the rules section.
    - Every time it finds a match (token) it executes the C code associated with that pattern
      - If a pattern is followed by | (instead of C code), the pattern uses the same C code as the next pattern in the file
    - When an input character matches no pattern, the lexer acts as though it matched a pattern whose code is "ECHO"
      - ECHO is a macro that writes code matched by the pattern
        - » writes a copy of the token to the output

User Subroutines Section

- The contents of this section are copied verbatim by lex to the C file
  - This section typically includes routines called from the rules
  - If you redefine input(), unput(), output() or yywrap(), then the new versions might be here

```
... definitions ...
%%
... rules ...
%%
... subroutines ...
```


Lex 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

Table 3: Lex Predefined Variables

Lex Example

Empty definition section

```
%%
    /* match everything except newline */
.   ECHO;
    /* match newline */
\n  ECHO;

%%

int yywrap(void) {
    return 1;
}

int main(void) {
    yylex();
    return 0;
}
```

Two patterns specified in rules section

Lex Example

```
%%
    /* match everything except newline */
.   ECHO;
    /* match newline */
\n  ECHO;

%%

int yywrap(void) {
    return 1;
}

int main(void) {
    yylex();
    return 0;
}
```

Defaults
yyin is stdin
yyout is stdout

You may change these to read from and write to a file respectively

Lex Example

```
%%
    /* match everything except newline */
.   ECHO;
    /* match newline */
\n  ECHO;

%%

int yywrap(void) {
    return 1;
}

int main(void) {
    yylex();
    return 0;
}
```

yywrap() is called when lex encounters an end of file.

It returns either 1 (done) or 0 (more processing is required)

Lex example 2

```
%{
    int yylineno;
%}
%%
^(.*)\n    printf("%4d\t%s", ++yylineno, yytext);
%%
int main(int argc, char *argv[]) {
    yyin = fopen(argv[1], "r");
    yylex();
    fclose(yyin);
}
```

Definition Section : 1 declaration
Rules Section: 1 pattern
Subroutine Section : main routine

Lex Example 3

```
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;
}
```

Definition Section : 2 definitions, 1 declaration
Rules Section: 1 pattern
Subroutine Section : main routine

Lex Example 4

```
%{
    int nchar, nword, nline;
%}
%%
\n          { nline++; nchar++; }
[^ \t\n]+   { nword++, nchar += yyleng; }
.           { nchar++; }
%%
int main(void) {
    yylex();
    printf("%d\t%d\t%d\n", nchar, nword, nline);
    return 0;
}
```

Definition Section : 3 declarations
Rules Section: 2 patterns
Subroutine Section : main routine