Compiler Course Program Assignment

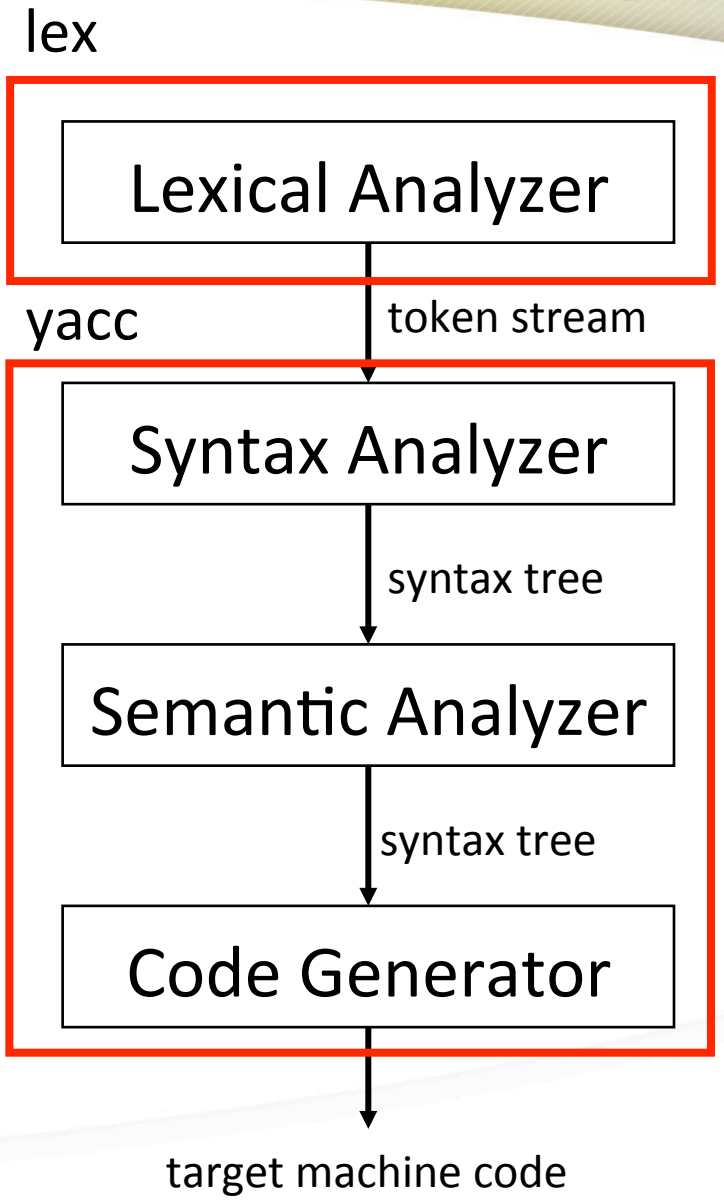
Speaker：Liaw, Jiann-Fuh

Program Assignment

- Goal: Build a compiler for C
- Deadline: 6/16 23:40
- Demo: 6/17
- Upload your project to moodle
  - A zipped file(.rar, .zip, .7z, ...) contain your source code and readme
    - do not hand in corpse(e.g. can not be compiled or be ran)!
  - Filename: StudentID
    - e.g. F74012345.rar
- Post your problems on moodle or ask TAs
- Environment: Ubuntu 14.04 with gcc 4.8.4

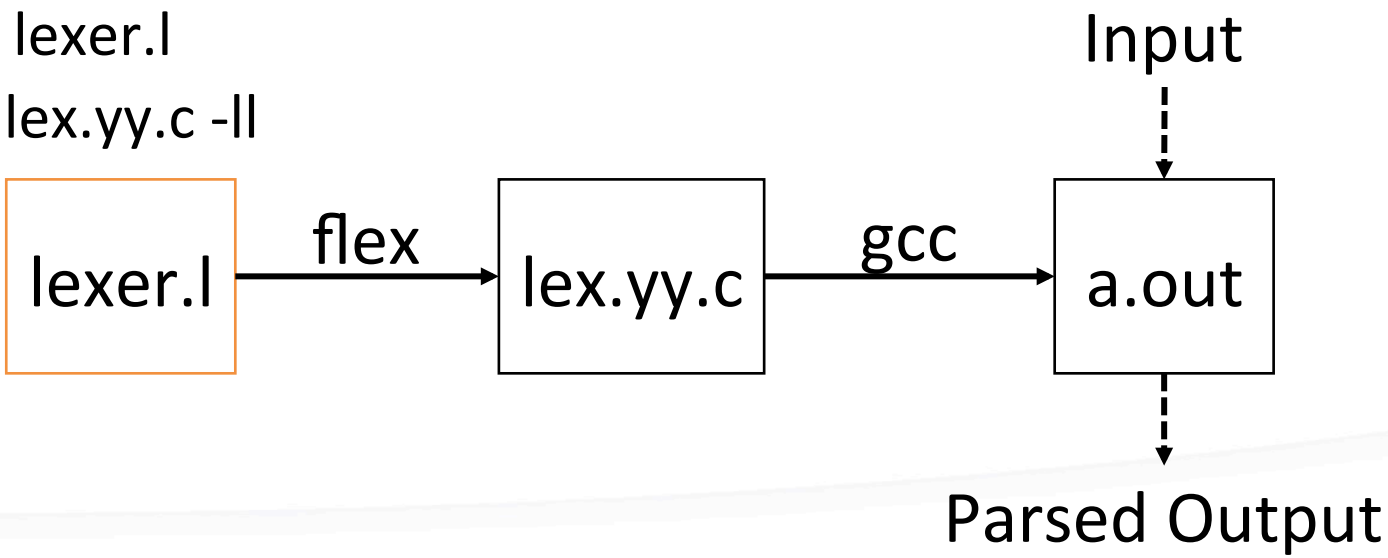
Compile Process

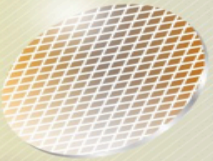
- Aid tools:
  - lex(flex 2.5.35)
  - yacc(bison 3.0.2)
- Lex is a program generator designed for lexical processing of character input streams
- Yacc provides a general tool for imposing structure on the input to a computer program



Introduction to Flex

- Flex allows you to implement a lexical analyzer by writing rules
- Flex compiles your rule file(e.g. lexer.l) to C source code implementing a finite automaton
  - you don't need to understand the C source file
- Usage
  - $ flex lexer.l
  - $ gcc lex.yy.c -ll





Rule Files

- Declarations and User subroutines sections are optional
  - write declarations and helper function in C
- Definitions section is also optional
  - useful to name regular expression
  - e.g. `DIGIT [0-9]`

Rule File Structure

```
%{
Declarations
%}

Definitions
%%
Rules
%%
User subroutines
```

Lex Regular Expression

x	the character “x”
“x”	an “x”, even if x is an operator
\x	an “x”, even if x is an operator
[xy]	the character x or y
[x-z]	the characters x, y or z
[^x]	any character but x
.	any character but newline
^x	an x at the beginning of a line
<y>x	an x When Lex is in start condition y

x$	an x at the end of a line
x?	an optional x
x*	0,1,2, ... instances of x
x+	1,2,3, ... instances of x
x|y	an x or an y
(x)	an x
x/y	an x but only if followed by y
{xx}	the translation of xx from the definitions section
x{m,n}	m through n occurrences of x

Operators
" \ [ ] ^ - ? . * + | ( ) $ / { } % < >

Lex Actions

- Rule specifies an action to perform if input matches the regular expression or definition
- The action is specified by writing regular C code
- The default action is copying input to output
- e.g.

```
[0-9] {
    printf("%s", yytext);
}
[ \t\n] ;
. ;
```

Rule File Structure

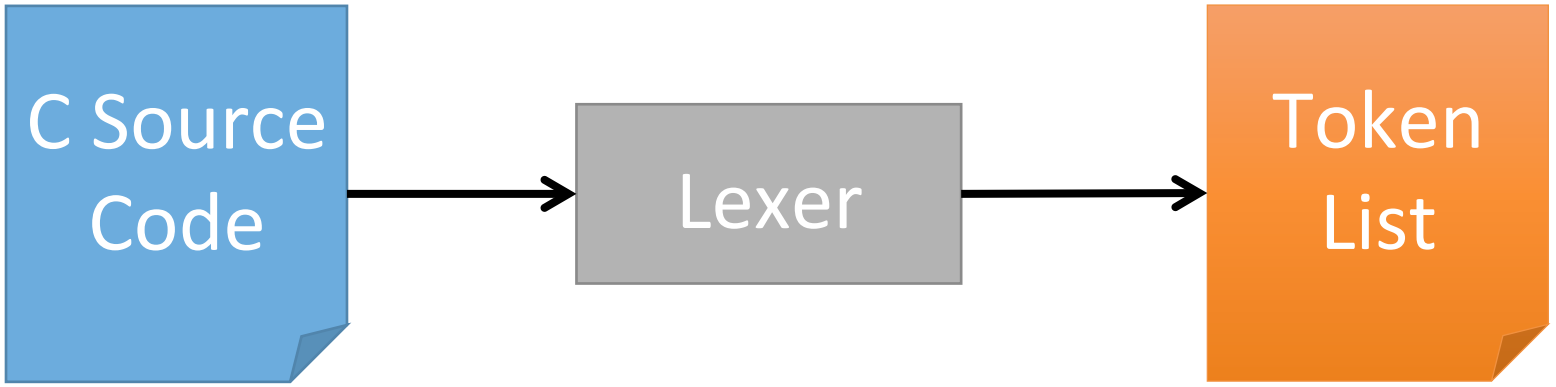
```
%{
Declarations
%}

Definitions
%%
Rules
%%
User subroutines
```

Lex Predefined Variables

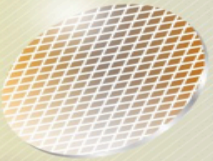
Name	Function
`int yylex(void)`	call to invoke lexer, returns token
`int yywrap(void)`	wrapup, return 1 if done, 0 if not done
`char *yytext`	pointer to matched string
`yyleng`	length of matched string
`yylineno`	current line number
`yylval`	value associated with token
`FILE *yyout`	output file
`FILE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

Lexer Practice



Token

- Keywords
  - int char float double return if else while break for print
- Operators
  - = ! + - * / == != < > <= >= && ||
- Special Symbols
  - [ ] ( ) { } ; ,
- Identifier
  - [a-zA-Z_][a-zA-Z0-9_]*
- Number
  - [1-9][0-9]+
- Char
  - ‘[.|\n|\t|]’
  - e.g. ‘a’, ‘\n’, ‘ ’
- Comment(no need to print)
  - //
- Error
  - other characters
  - e.g. `, ~



Token List Format

- Line xx
  - xx – Line number
- <yyy> zzz
  - yyy – Token Category(Keyword、Operator、Special Symbol、Identifier、Char、Number、Error)
  - zzz – Token
- e.g.

Line 1

<Keyword>	int
<Identifier>	main
<Special Symbol>	(
<Special Symbol>	)

Lexer Output Example

test.c

```
1  int main() {
2          int i1 = 3;
3          if() ~
4  }
```

lexical analyze →

```
Line 1
    <Keyword>           int
    <Identifier>        main
    <Special Symbol>    (
    <Special Symbol>    )
    <Special Symbol>    {
Line 2
    <Keyword>           int
    <Identifier>        i1
    <Operator>          =
    <Number>            3
    <Special Symbol>    ;
Line 3
    <Keyword>           if
    <Special Symbol>    (
    <Special Symbol>    )
    <Error>             ~
Line 4
    <Special Symbol>    }
Line 5
```