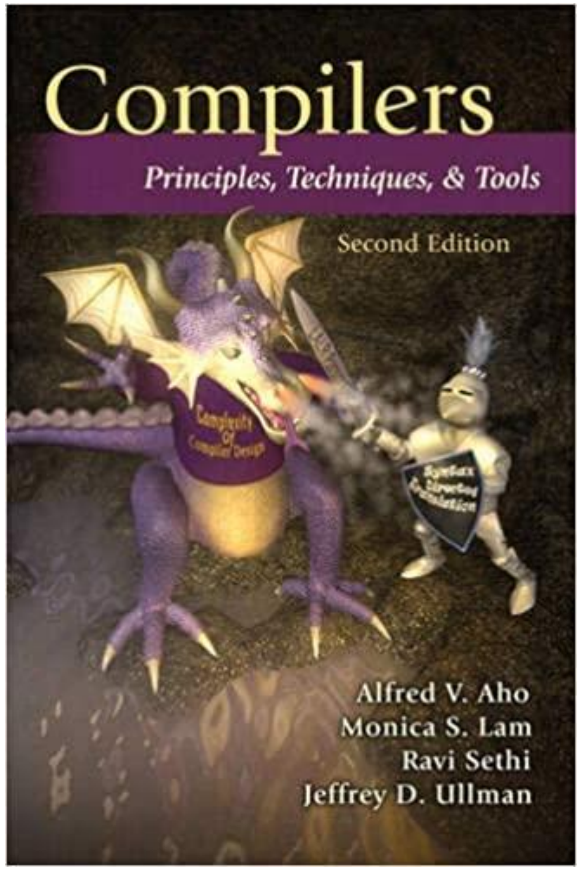
CS340400 Compiler Design Homework 1

Deadline
4/12 (Wed) 23:59

Alfred Vaino Aho and Jeffrey David Ullman

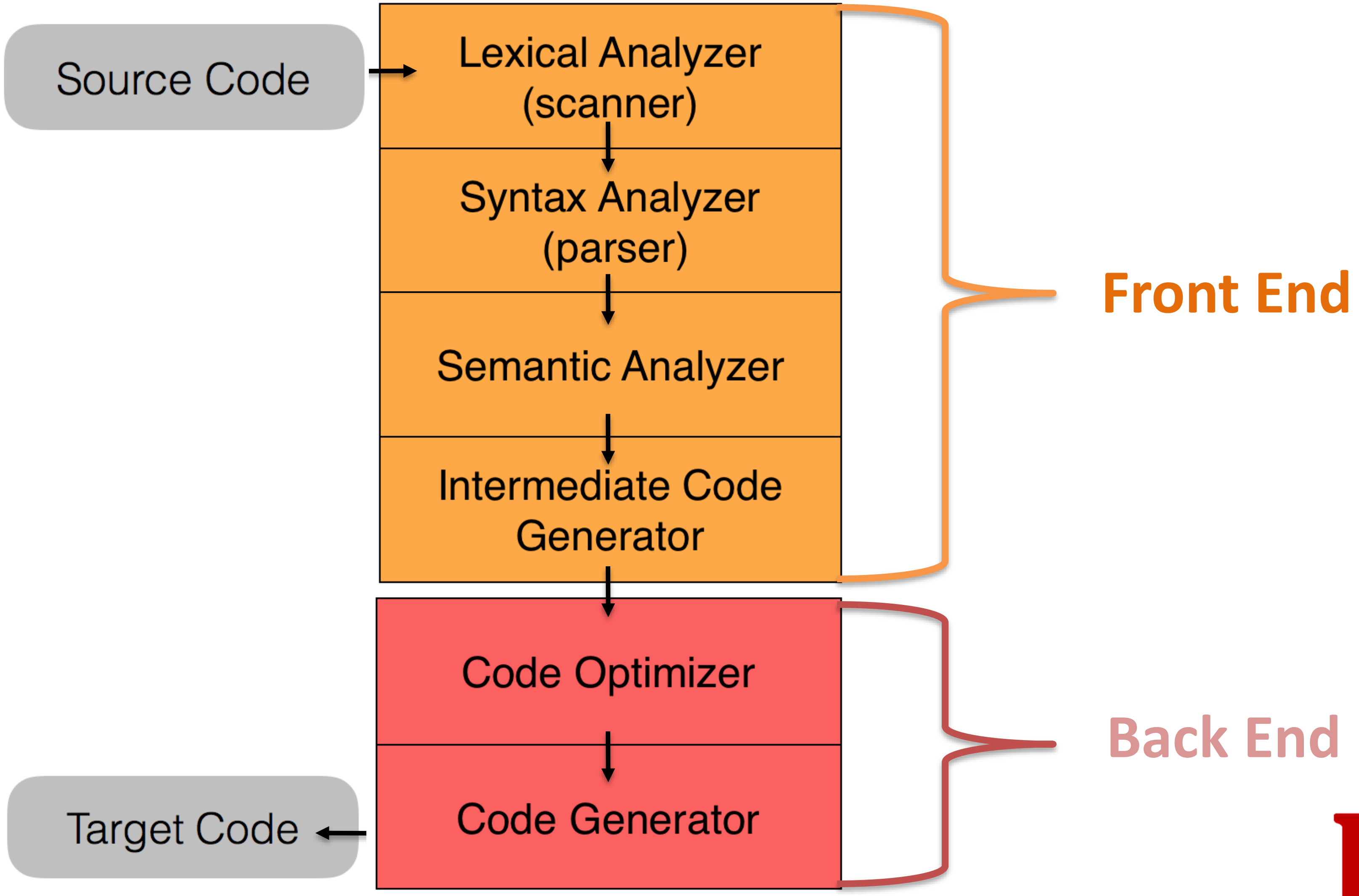
Turing Award Winner of 2020

for fundamental algorithms and theory underlying
programming language implementation

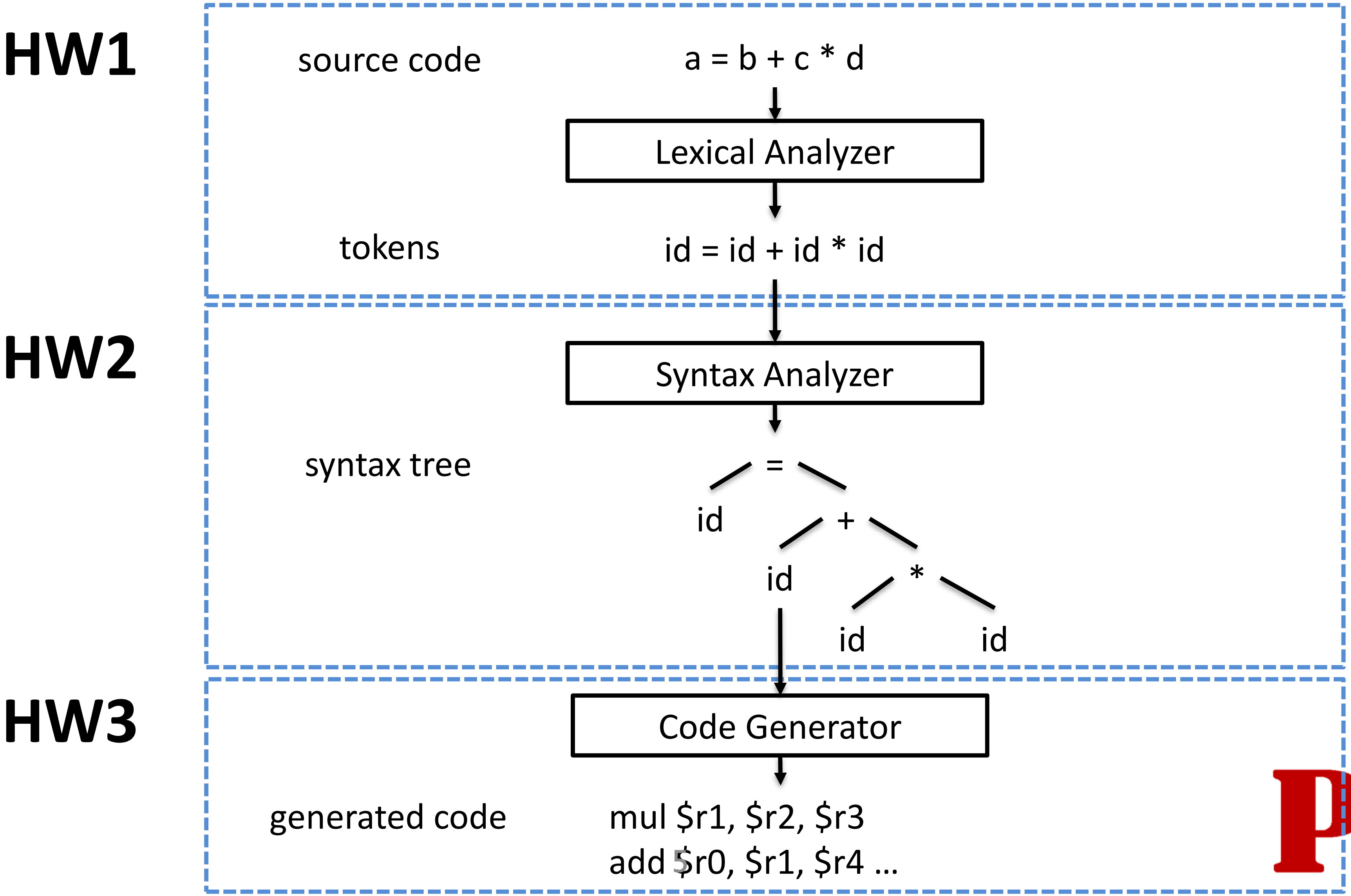


Overview

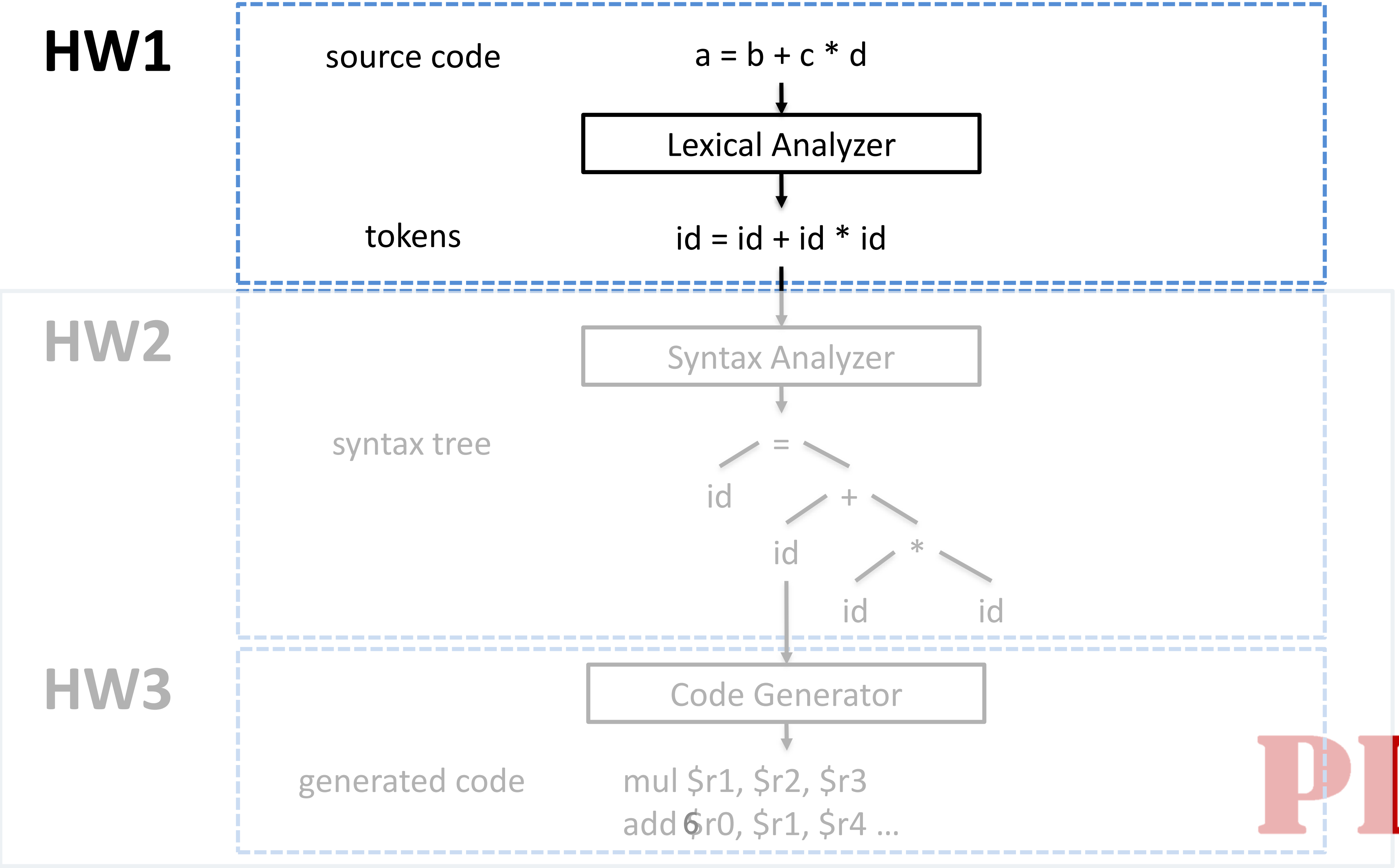
The Structure of a Modern Compiler



What are we Going to do?



The Structure of Compiler



Lex
- lexical analyser generator

What is Lex?

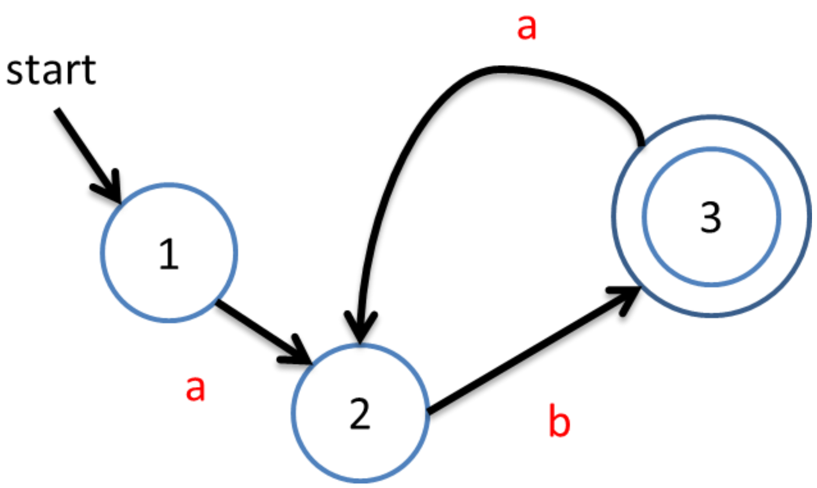
- **Lex** is a **program generator** designed for lexical processing of character input streams. It accepts a high-level, problem oriented specification for character string matching, and produces a program in a general purpose language which recognizes **regular expressions**.
- The regular expressions are specified by the user in the source specifications given to **Lex**.
- **Lex** generates a **deterministic finite automaton (DFA)** from the regular expressions in the source.
- The **Lex** written code recognizes these expressions in an input stream and partitions the input stream into strings matching the expressions.

From http://dinosaur.compilertools.net/lex/

(ab)+

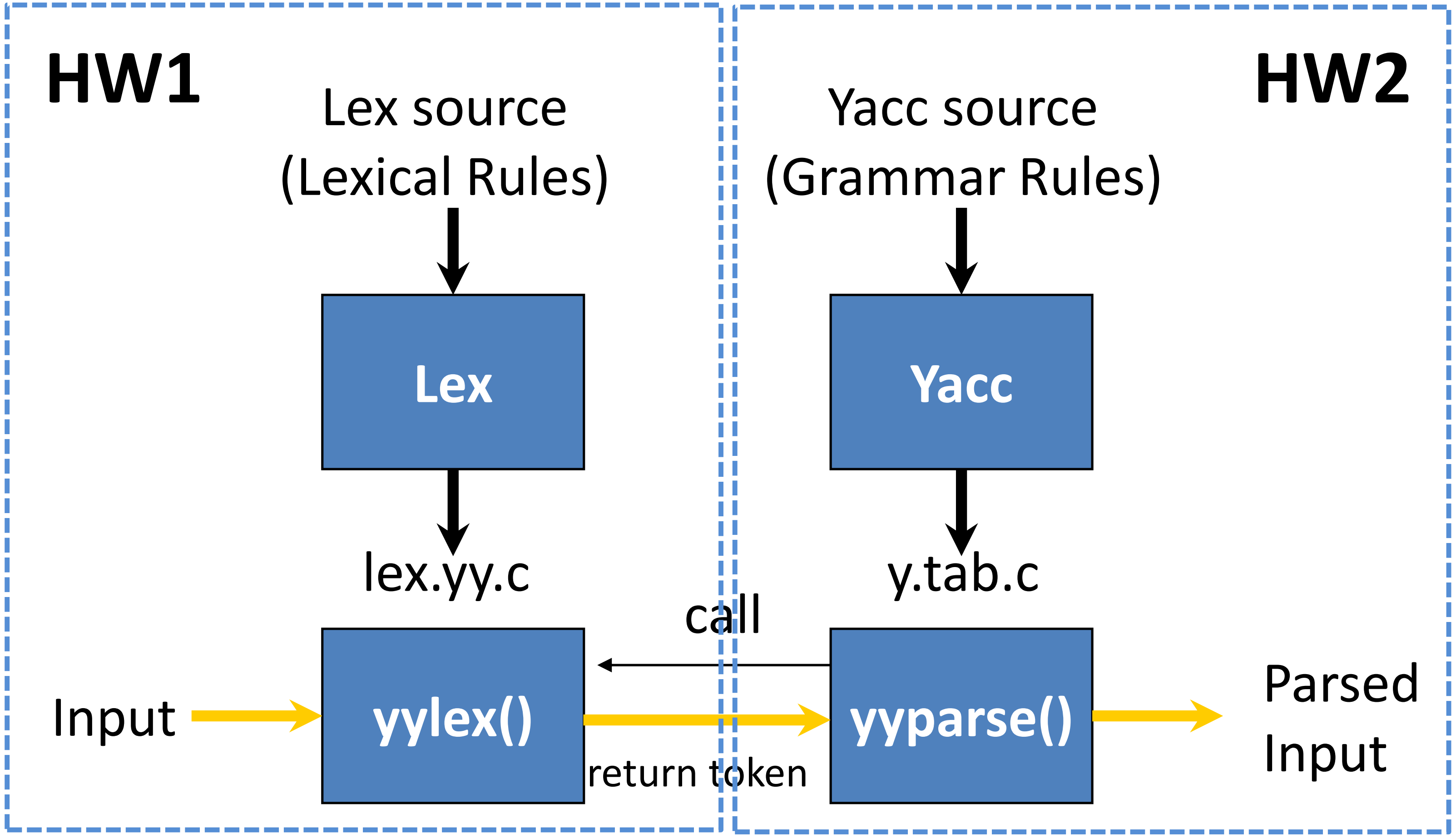
LEX

Input Stream



If (matches the DFA expression)
Do Something…….

Lex with Yacc



How to Write Lex?

Definition Section → **(required)**
%%
Rules Section → **(required)**
%%
C Code Section → **(optional)**

How to Write Lex?

Definition Section
%%
Rules Section
%%
C Code Section

```
%{
#include <stdio.h>

int         lineCount=0;
%}
```

The Definition Section will be copied to the top of generated C program. Include header files, declare variables.

How to Write Lex?

Definition
Section
%%
Rules Section
%%
C Code Section

```
\n        { lineCount++;
            printf("line:%d\n", lineCount); }
```

The Rules Section is for writing regular expression to recognize tokens. When **pattern** is matched, then execute **action**

[Regular expression rule]   { The things you want to do; }

NTHU PL LAB

How to Write Lex?

```
Definition
Section
%%
Rules Section
%%
C Code Section
```

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

int yywrap() {
  return 1;
}

// Other function you defined.
```

The C Code Section will be copied to the bottom of generated C program.

How to Write Lex?

A completed Lex program

```
Definition
Section
%%
Rules Section
%%
C Code Section
```

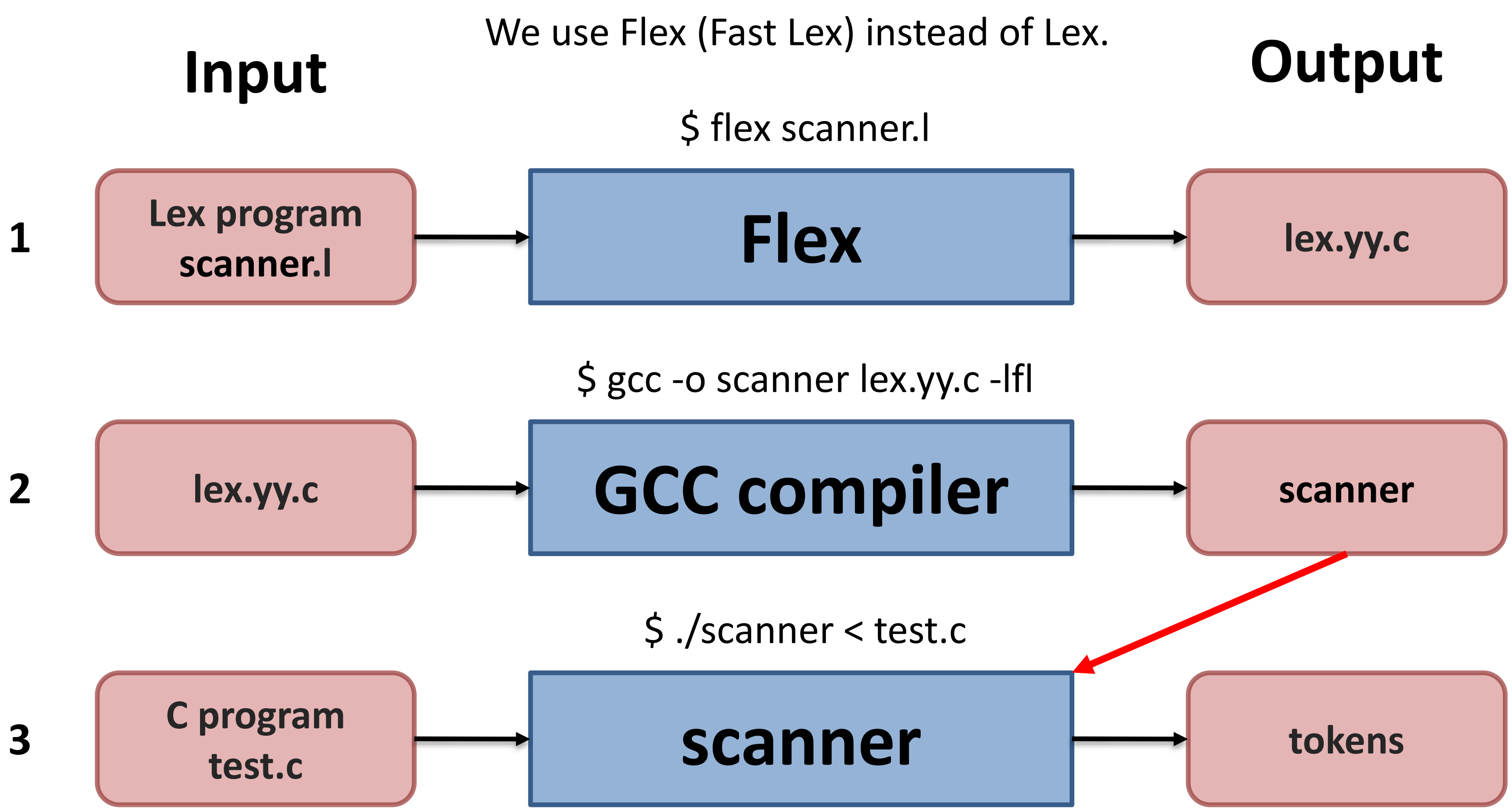
```
%{
#include <stdio.h>

int          lineCount=0;
%}

%%
\n           { lineCount++;
              printf("line:%d\n", lineCount); }

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

Compilation Flow



Flex: the Fast Lexical Analyser Generator

```
3 Introduction
**************

'flex' is a tool for generating "scanners".  A scanner is a program
which recognizes lexical patterns in text.  The 'flex' program reads the
given input files, or its standard input if no file names are given, for
a description of a scanner to generate.  The description is in the form
of pairs of regular expressions and C code, called "rules".  'flex'
generates as output a C source file, 'lex.yy.c' by default, which
defines a routine 'yylex()'.  This file can be compiled and linked with
the flex runtime library to produce an executable.  When the executable
is run, it analyzes its input for occurrences of the regular
expressions.  Whenever it finds one, it executes the corresponding C
code.
```

```
The 'flex' input file consists of three sections, separated by a line
containing only '%%'.

        definitions
        %%
        rules
        %%
        user code
```

Link with library **libfl.a**

From: http://oz.nthu.edu.tw/~d947207/chap10_lex.ppt

Flex Example:
Count Number of Lines and Number of Characters

count_line.l

```
%{

#include <stdio.h>
int num_lines = 0, num_chars = 0;

%}

%%

\n   { ++num_lines ; ++ num_chars ; }
.    { ++num_chars ; }

%%

int main(int argc, char* argv[])
{
  yylex() ;
  printf("# of lines = %d, # of chars = %d\n",
      num_lines, num_chars );
  return 0 ;
}
```

```
[imsl@linux count_line]$ ./a.out
This is a book
byebye
# of lines = 2, # of chars = 22
[imsl@linux count_line]$
```

Press Ctrl+D

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
T	h	i	s		i	s		a		b	o	o	k	\n
b	y	e	b	y	e	\n								

```
[imsl@linux count_line]$
[imsl@linux count_line]$ flex count_line.txt
[imsl@linux count_line]$ ls
count_line.txt  lex.yy.c
[imsl@linux count_line]$ gcc lex.yy.c -lfl
[imsl@linux count_line]$ ls
a.out  count_line.txt  lex.yy.c
[imsl@linux count_line]$
```

Generate source C-code **lex.yy.c**

Library libfl.a

NTHU PL LAB

From: http://oz.nthu.edu.tw/~d947207/chap10_lex.ppt

Grammar of Input file of Flex

Lex copy data enclosed by %{ and %} into C source file

pattern action

\n { ++num_lines ; ++ num_chars ; }

. { ++ num_chars ; }

wild card character, represent any character expect line feed **\n**

User code

```
%{

#include <stdio.h>
int num_lines = 0, num_chars = 0;

%}

%%

\n  { ++num_lines ; ++ num_chars ; }
.   { ++num_chars ; }

%%

int main(int argc, char* argv[])
{
   yylex() ;
   printf("# of lines = %d, # of chars = %d\n",
       num_lines, num_chars );
   return 0 ;
}
```

grammar of input file

definition section

%%

rule section

%%

user code

pattern action

When **pattern** is matched, then execute **action**

NTHU PL LAB

Can we Compile lex.yy.c without –lfl ?

We want to use **lex.yy.c** on different platforms (Linux and windows), to avoid specific library is lesson one.

```
[imsl@linux count_line]$ gcc lex.yy.c
/tmp/ccgm0gZ8.o(.text+0x30d): In function `yylex':
: undefined reference to `yywrap'
/tmp/ccgm0gZ8.o(.text+0xa4f): In function `input':
: undefined reference to `yywrap'
collect2: ld returned 1 exit status
[imsl@linux count_line]$
```

Library **libfl.a** contains function *yywrap()*

-lfl means "include library **libfl.a**", this library locates in **/usr/lib**

```
[imsl@linux lib]$ pwd
/usr/lib
[imsl@linux lib]$ ls libf*
libfam.a      libfam.so.0.0.0      libfontconfig.so.1.0  libform.so.5.3     libfreetype.so.6
libfam.la     libfl.a              libform.a             libfreetype.a      libfreetype.so.6.3.2
libfam.so     libfontconfig.so     libform.so            libfreetype.la
libfam.so.0   libfontconfig.so.1   libform.so.5          libfreetype.so
[imsl@linux lib]$ ar -t libfl.a
libmain.o
libyywrap.o
[imsl@linux lib]$
```

contains function *main()* and *yywrap()*

From: http://oz.nthu.edu.tw/~d947207/chap10_lex.ppt

PL NTHU LAB

From: http://oz.nthu.edu.tw/~d947207/chap10_lex.ppt

Can we Compile lex.yy.c without –lfl ?

count_line.l

```
%{

#include <stdio.h>
int num_lines = 0, num_chars = 0;

%}

%%
\n      { ++num_lines ; ++ num_chars ; }
.       { ++num_chars ; }

%%

int main(int argc, char* argv[])
{
  yylex() ;
  printf("# of lines = %d, # of chars = %d\n",
      num_lines, num_chars );
  return 0 ;
}

/*  when yylex() read a EOF, then it call yywrap().
 *  Return value of yywrap() is either 0 or 1.
 *  if return value is 1, then it means NO any input,
 *     program is end ( yylex() return 0 )
 *  if return value is 0, then tells yylex() that
 *     new file is ready, it can go on to process new token.
 *
 *  Hence if we have multiple files to be parsed, then we can use yywrap() to
 *  open file one by one
 */

int yywrap()
{
   return 1 ;  /* eof */
}
```

Implement function **yywrap** explicitly

How to Process a file?

count_line.l

```
%%
\n      {
            ++num_lines ;
            ++ num_chars ;
        }
.       {
            ++num_chars ;
        }

%%

int main(int argc, char* argv[])
{
  ++argv ;
  --argc ; /* skip over program name*/

  if ( 0 < argc ){
    yyin = fopen( argv[0], "r") ;
  }else{
    yyin = stdin ;
  }
  yylex() ;
  printf("# of lines = %d, # of chars = %d\n",
        num_lines, num_chars );
  return 0 ;
}

/*  when yylex() read a EOF, then it call yywrap().
 *  Return value of yywrap() is either 0 or 1.
 *  if return value is 1, then it means NO any input,
```

lex.yy.c

```
/* Translate the current start state into a value that can b
 * to BEGIN to return to the state.  The YYSTATE alias is fo
 * compatibility.
 */
#define YY_START ((yy_start - 1) / 2)
#define YYSTATE YY_START

/* Action number for EOF rule of a given start state. */
#define YY_STATE_EOF(state) (YY_END_OF_BUFFER + state + 1)

/* Special action meaning "start processing a new file". */
#define YY_NEW_FILE yyrestart( yyin )

#define YY_END_OF_BUFFER_CHAR 0

/* Size of default input buffer. */
#define YY_BUF_SIZE 16384

typedef struct yy_buffer_state *YY_BUFFER_STATE;

extern int yyleng;
extern FILE *yyin, *yyout;

#define EOB_ACT_CONTINUE_SCAN 0
#define EOB_ACT_END_OF_FILE 1
#define EOB_ACT_LAST_MATCH 2
```

yyin is a file pointer in **lex**, function yylex() read characters from **yyin**

From: http://oz.nthu.edu.tw/~d947207/chap10_lex.ppt

Lex Predefined Variables

Name	Function
char *yytext	Pointer to matched string.
int yyleng	Length of matched string.
int yylex(void)	Function call to invoke lexer and return token.
int yywrap(void)	Return 1 if no more files to be read.
char *yymore(void)	Return the next token.
int yyless(int n)	Retain the first n characters in yytext and (sort of) return the rest back to the input stream.
FILE *yyin	Input stream pointer.
FILE *yyout	Output stream pointer.
ECHO	Print out the yytext.
BEGIN	Condition switch.
REJECT	Go to the next alternative rule.

man flex or **info flex** to get more info

Regular Expressions

.	Any character excepts ‘\n’.	. = {a, b, c, d, ……}
*	Zero or more.	ab* = {a, ab, abb, abbb, ……}
+	One or more.	ab+ = {ab, abb, abbb, ……}
?	Zero or one.	a? = {ε, a}
|	Or.	a|b = {a, b}
[]	Any character of the character set.	[abc] = {a, b, c}
()	To group characters.	(ab)* = {ε, ab, abab, ……}
\	For escape character.	* = {*}, \\ = {\}
"…"	Literally.	"a*" = {a*}
{n,N}	Repeat n to N times.	a{1,3} = {a, aa, aaa}
[^…]	Not these characters. (Opposite of [])	[^abc] = {d, e, f, ……}
^/$	Head/End of line.	^a = a... // line starts with a
/	Followed by specific character.	a/b = {ab} // but only returns a

Regular Expressions

Input String

she

Rule Section

```
%%

she      { printf("she\t"); }
[sS]he   { printf("another she\t"); }
he       { printf("he\t"); }
s        { printf("s\t"); }

%%
```

The output result is "she":

1. Choose the longest matching pattern. If there's multiple matches of the same length, then
2. Choose the first matching pattern from top to bottom

More Elegant Way to Write Regular Expressions

```
%{
#include <stdio.h>

int         lineCount=0;
%}

ch          [a-z]

%%
\n          { lineCount++;
            printf("line:%d\n", lineCount) ;}

{ch}+       { ECHO; }
```

More Elegant Way to Write Regular Expressions

```
%{
#include <stdio.h>

int         lineCount=0;
%}

%%
\n          { lineCount++;
            printf("line:%d\n", lineCount); }

[[:alpha:]]+        { ECHO; }
```

Regular Expressions

Regular Expression	Meaning
[a-zA-Z]	Any character of a ~ z and A ~ Z.
[0-9]	Any character of 0 ~ 9.
[:lower:]	[[:lower:]] = [a-z]
[:upper:]	[[:upper:]] = [A-Z]
[:alpha:]	[[:alpha:]] = [a-zA-Z]
[:digit:]	[[:digit:]] = [0-9]
[:alnum:]	[[:alnum:]] = [a-zA-Z0-9]

Start Condition

- What if you encounter the string like this?

Input String

```
/* int count
   is for counting line number */
```

Input String

```
printf( “int is 32-bit” );
```

Start Condition

```
%{
…
%}

%x    COMMENT

%%
```

- Declare at Definition Section
- %s STATE_NAME – inclusive
  - If the start condition is *inclusive*, then rules with no start conditions at all will also be active.
- %x STATE_NAME – exclusive
  - If it is *exclusive*, then only rules qualified with the start condition will be active.

Start Condition

Input String

/*int

```
%{
…
%}

%x    COMMENT
/* Exclusive */

%%

"/*"             { BEGIN COMMENT; }
int              { printf("normal\n");}
<COMMENT>int     { printf("special\n");
                   BEGIN 0; }

%%
```

Start Condition

Input String

/*int

```
%{
...
%}

%s    COMMENT
/* Inclusive */

%%

"/*"              { BEGIN COMMENT; }
int               { printf("normal\n");}
<COMMENT>int      { printf("special\n");
                    BEGIN 0; }

%%
```

Versions of Lex

- AT&T: lex
  http://www.combo.org/lex_yacc_page/lex.html
- GNU: flex
  http://www.gnu.org/manual/flex-2.5.4/flex.html
- a Win32 version of flex
  http://www.monmouth.com/~wstreett/lex-yacc/lex-yacc.html

  or Cygwin
  http://sources.redhat.com/cygwin/
- Lex on different machines is not created equal.

Homework1 - Requirements

Subset of C Language

- Character Set of Testcases
  - ASCII characters
    - Only those in the right image
  - '\n' and '\t'
- Maximum Length of each Line: 299

二進位	十進位	十六進位	圖形	二進位	十進位	十六進位	圖形	二進位	十進位	十六進位	圖形
0010 0000	32	20	(space)	0100 0000	64	40	@	0110 0000	96	60	`
0010 0001	33	21	!	0100 0001	65	41	A	0110 0001	97	61	a
0010 0010	34	22	"	0100 0010	66	42	B	0110 0010	98	62	b
0010 0011	35	23	#	0100 0011	67	43	C	0110 0011	99	63	c
0010 0100	36	24	$	0100 0100	68	44	D	0110 0100	100	64	d
0010 0101	37	25	%	0100 0101	69	45	E	0110 0101	101	65	e
0010 0110	38	26	&	0100 0110	70	46	F	0110 0110	102	66	f
0010 0111	39	27	'	0100 0111	71	47	G	0110 0111	103	67	g
0010 1000	40	28	(	0100 1000	72	48	H	0110 1000	104	68	h
0010 1001	41	29	)	0100 1001	73	49	I	0110 1001	105	69	i
0010 1010	42	2A	*	0100 1010	74	4A	J	0110 1010	106	6A	j
0010 1011	43	2B	+	0100 1011	75	4B	K	0110 1011	107	6B	k
0010 1100	44	2C	,	0100 1100	76	4C	L	0110 1100	108	6C	l
0010 1101	45	2D	-	0100 1101	77	4D	M	0110 1101	109	6D	m
0010 1110	46	2E	.	0100 1110	78	4E	N	0110 1110	110	6E	n
0010 1111	47	2F	/	0100 1111	79	4F	O	0110 1111	111	6F	o
0011 0000	48	30	0	0101 0000	80	50	P	0111 0000	112	70	p
0011 0001	49	31	1	0101 0001	81	51	Q	0111 0001	113	71	q
0011 0010	50	32	2	0101 0010	82	52	R	0111 0010	114	72	r
0011 0011	51	33	3	0101 0011	83	53	S	0111 0011	115	73	s
0011 0100	52	34	4	0101 0100	84	54	T	0111 0100	116	74	t
0011 0101	53	35	5	0101 0101	85	55	U	0111 0101	117	75	u
0011 0110	54	36	6	0101 0110	86	56	V	0111 0110	118	76	v
0011 0111	55	37	7	0101 0111	87	57	W	0111 0111	119	77	w
0011 1000	56	38	8	0101 1000	88	58	X	0111 1000	120	78	x
0011 1001	57	39	9	0101 1001	89	59	Y	0111 1001	121	79	y
0011 1010	58	3A	:	0101 1010	90	5A	Z	0111 1010	122	7A	z
0011 1011	59	3B	;	0101 1011	91	5B	[	0111 1011	123	7B	{
0011 1100	60	3C	<	0101 1100	92	5C	\	0111 1100	124	7C	|
0011 1101	61	3D	=	0101 1101	93	5D	]	0111 1101	125	7D	}
0011 1110	62	3E	>	0101 1110	94	5E	^	0111 1110	126	7E	~
0011 1111	63	3F	?	0101 1111	95	5F	_				

Subset of C Language

- Implement: Keywords
  - for, do, while, break, continue, if, else, return, struct, switch, case, default
  - void, int, double, float, char
  - const, signed, unsigned, short, long
- Implement: Macros
  - NULL, __COUNTER__, __LINE__, INT_MAX, INT_MIN, CHAR_MAX, CHAR_MIN, MAX, MIN

Subset of C Language

- Implement: Identifiers (case-sensitive)
  - Follow the standard C variable naming rule: A variable name can only have letters (both uppercase and lowercase letters), digits and underscore. The first letter of a variable should be either a letter or an underscore.
- Implement: Operators
  - `+ - * / % ++ -- < <= > >= == != = && || ! & |`
- Implement: Punctuation characters
  - `: ; , . [ ] ( ) { }`

Subset of C Language

- Implement:
  - Integer constants (e.g. 0, -0, 1, 123, 45, -2131)
    - There can be a deliberate '+' preceding positive numbers
      - Note: "- 0" is "-" (op) and "0" (integer), while "-0" is an integer
  - Simple form floating point constants (e.g. 0.0, 0.1234, 123.456, -0.0, -0.1234)
    - There can be a deliberate '+' preceding positive numbers
      - Note: "- 0." is "-" (op) and "0." (double), while "-0." is a double
    - The number, if it's 0, before or after the decimal point can be missing

Subset of C Language

- Implement:
  - String constants (e.g. "This is a string")
    - Take particular note of escaped sequences (<https://en.wikipedia.org/wiki/Escape_sequences_in_C#Table_of_escape_sequences>)
  - Character constants (e.g. 'a', 'b', '\t', '\0')
    - Also take particular note of escaped sequences (<https://en.wikipedia.org/wiki/Escape_sequences_in_C#Table_of_escape_sequences>)
  - C Comments
    - both the // … and /* ... */ syntax

Subset of C Language

- Implement: Pragma directives
  - #pragma source on
  - #pragma source off
  - #pragma token on
  - #pragma token off
  - Note
    - Pragmas could have spaces and/or '\t' on the same line and between words in them, but not between the hash ('#') and "pragma"
    - Pragmas should be on their own line

Subset of C Language

- Note
  - Always parse with the rule that matches the longest input
    - E.g. "ab123" is an Identifier ("ab123", length 5), not Identifier ("ab", length 2) and Integer ("123", length 3)

Output Format

- Token type
  - Keyword (**key**): Refer to slide page 35
  - Macro (**macro**): Refer to slide page 35
  - Identifier (**id**): Refer to slide page 36
  - Operator (**op**): Refer to slide page 36
  - Punctuation Character (**punc**): Refer to slide page 36
  - Integer (**integer**): Refer to slide page 37
  - Floating Point (**float**): Refer to slide page 37
  - Char (**char**): Refer to slide page 38
  - String (**string**): Refer to slide page 38

One must print the token types with the type names designated in the parentheses once a token of these types is encountered.

Output Format

- One must print the result in this format
  - For each line of input
    - If the extracted token is a **pragma directive** or part of a **comment**, print nothing except the line information (see below)
    - Otherwise, print "#" first, then the token type and token content (`#${token_type}:${token_content}`)
      - Quotes of strings and characters should be retained
    - Finally, print the line number and content at the end of each input line (`${line_number}:${line_content}`)
      - #token_type1:token_content1
      - #token_type2:token_content2
      - ...
      - line_number:line_content

Output Format Examples: Testcase0 Line1

char a = 'i';

Output Format Examples: Result

#key:char

#id:a

#op:=

#char:'i'

#punc:;

1:char a = 'i';

Please use diff command to check your output against the golden scanner!

Pragma Directives: Source

default:	
#pragma source on char a = 'i';	#pragma source off char a = 'i';
1:**#pragma source on** #key:char #id:a #op:= #char:'i' #punc:; 2:char a = 'i';	#key:char #id:a #op:= #char:'i' #punc:;

Pragma Directives: Token

default

```
#pragma token on
char a = 'i';
```

```
1:#pragma token on
#key:char
#id:a
#op:=
#char:'i'
#punc:;
2:char a = 'i'    ;
```

```
#pragma token off
char a = 'i';
```

```
1:#pragma token off
2:char a = 'i';
```

Grading Policies

- For all homeworks
  - Any warning during compilation: -20 points penalty
  - Late Submission: -10 points penalty/24H
    - Applied at the last step after all other penalties
  - Executable, but not complying to specifications: 20% off your original score if you apply for a manual review (Reviews are not guaranteed to be accepted.)
    - Including build script failures
  - Non-executable: A flat 30 point if you turn in your source code and a report
    - Refer to later slides for report requirements
  - **Plagiarism/Cheating: A flat 0 point even if you do everything you can do**

Grading Policies

- If your scanner behaves correctly on the following type:
  - Keywords && Identifiers: +10
  - Macros: +10
  - Operators && Punctuations: +15
  - Integers && Floating Points && Characters: +15
  - Strings: +15
  - Comments: +15
  - Pragma Directives: +10
  - Comprehensive Testcases: +10
- A sample testcase is released for reference
  - Use `./scanner < sample_testcase.txt` to test your scanner

Report

- For students who **cannot finish** his homework, he can turn in a report answering ALL of the following topics:
  - Describe Lex, including its features and possible purposes
  - Describe the problems you faced when implementing your scanner
  - Describe your guess/understanding on the nature of those problem, and what you did to solve them
- For those who successfully passes at least 1 testcase, no report is required

Submission

- HW1 Lex Implementation: Submit on Server
  - Create a `hw1` directory in your home, and provide in it:
    - A lex source code file named `scanner.l`. E.g. `/home/104062634/hw1/scanner.l`
    - A makefile to compile your code. It should only refer to resources in your `hw1` directory using relative paths. E.g. `/home/104062634/hw1/makefile`
  - The compiled output must be named `scanner` with the executable bit set
  - Make sure your program works correctly in the server environment
- Report + Project Source Code: eeclass (if you can't finish your implementation)
  - Upload your report (PDF) and source code (ZIP) to eeclass

How to Connect to our Server?

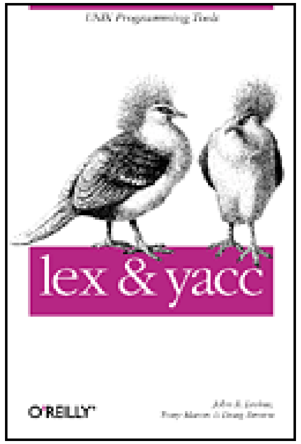
- SSH Protocol
  - IP: 140.114.88.201
  - Port: 8787
  - Username: (Will be released)
  - Default Password: (Will be released)
    - One can change password by entering `passwd`
- Clients
  - Windows: PuTTY, MobaXterm, …
  - Linux, Mac OS: Built-in ssh

Linux Materials

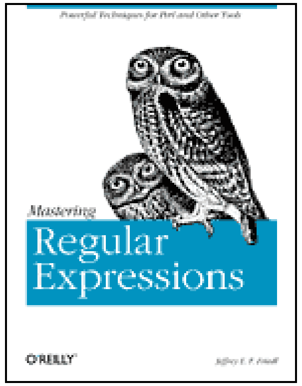
- **Linux Command**
  - https://linux.vbird.org/linux_basic/centos7/0210filepermission.php
- **Vim**
  - https://linux.vbird.org/linux_basic/centos7/0310vi.php
- **Shell Script**
  - https://linux.vbird.org/linux_basic/centos7/0340bashshell-scripts.php
- **Makefile**
  - http://www.cprogramming.com/tutorial/makefiles.html
  - http://jimmynuts.blogspot.tw/2010/12/gnu-makefile.html

Reference

- lex & yacc
  - by John R.Levine, Tony Mason & Doug Brown
  - O'　Reilly
  - ISBN: 1-56592-000-7



- Mastering Regular Expressions
  - by Jeffrey E.F. Friedl
  - O'　Reilly
  - ISBN: 1-56592-257-3



Contact Us

- TAs
  - 鄭盛元 sycheng@pllab.cs.nthu.edu.tw
  - 賴宏銘 hmlai@pllab.cs.nthu.edu.tw
  - 許嘉蓁 cc.hsu@pllab.cs.nthu.edu.tw