

Principles of Programming Languages

COMP3031: Lex (Flex) and Yacc (Bison)

Prof. Dekai Wu

Department of Computer Science and Engineering
The Hong Kong University of Science and Technology
Hong Kong, China

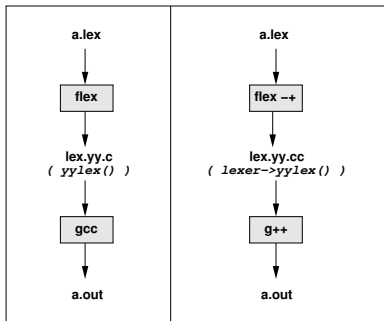


Fall 2012

Part I

flex

flex: Fast Lexical Analyzer



- flex is GNU's extended version of the standard UNIX utility `lex`, that generates **scanners** or **tokenizers** or **lexical analyzers**.
- flex reads a description of a scanner written in a **lex file** and outputs a C or C++ program containing a routine called `yylex()` in C or `(FlexLexer*)lexer->yylex()` in C++.
- flex compiles `lex.yy.c` to `a.out` which will be the lexical analyzer.

flex Example 1

```
%option noyywrap /* see pp. 30 */

%{
int numlines = 0;
int numchars = 0;
%}

%%
\n ++numlines; ++numchars;
. ++numchars;
%%

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

```
%{
```

```
    text to be copied exactly to the output
```

```
%}
```

```
flex Definitions
```

```
%%
```

```
    Rules = patterns in RE + actions in C or C++
```

```
%%
```

```
user code (in C or C++)
```

- Patterns, written in REs, must start on the first column, and action must start on the same line as its pattern.
- In the **Definitions** or **Rules** sections, any indented text or text enclosed in “%{” and “%}” is copied verbatim to the output.

How the Input is Matched?

- The generated lexical analyzer should have a loop calling the function `yylex()` for the input file to be scanned.
- Each call to `yylex()` will scan the input from left to right looking for strings that match any of the RE patterns.
- If it finds more than 1 match, it takes the **longest** match.
- If it finds 2 matches of the same length, it takes the first rule.
- When there is a match,

```
extern char* yytext = /* content of matched string */  
extern int yyleng = /* length of the matched string */
```

- If no rule is given, the **default rule** is to echo the input to the output.

flex Example 2: Default Rule

```
%option noyywrap

%%
%%

int main(int argc, char** argv)
{
    yylex();
    return 0;
}
```

How the Input is Matched? ..

- Actually the variable `yytext` can be specified as a pointer or an array in the flex-definition section.

```
%pointer    /* extern char* yytext */  
%array      /* extern char yytext[YYLMAX] */
```

- Using pointer for `yytext` renders faster operation and avoids buffer overflow for large tokens. While it may be modified but you should NOT lengthen it or modify beyond its length (as given by `yyleng`). Using array for `yytext` allows you to modify the matched string freely.
- You cannot use `%array` with C++ programs.

flex Example 3: Use of yytext

```
%option noyywrap

%{
#include <stdio.h>
%}

%%
[a-zA-Y]    printf("%c", *yytext + 1);
[zZ]        printf("%c", *yytext - 25);
.           printf("%c", *yytext);
%%

int main(int argc, char** argv)
{
    yylex();
    return 0;
}
```

2 flex Directives: ECHO, REJECT

- 1 **ECHO**: copy `yytext` to the output
- 2 **REJECT**: ignore the current match and proceed to the next match.
 - if there are 2 rules that match the same length of input, it may be used to select the 2nd rule.
 - may be used to select the rule that matches less text.

flex Example 4: REJECT

```
%option noyywrap

%{
#include <stdio.h>
%}

%%
a      |
ab     |
abc    |
abcd   ECHO; REJECT;
.|\n   printf("xx%c", *yytext);
%%

int main(int argc, char** argv)
{
    yylex(); return 0;
}
```

Global Variables/Classes

C Implementation	C++ Implementation
FILE* yyin	abstract base class: FlexLexer
FILE* yyout	derived class: yyFlexLexer
char* yytext	member function: const char* YYText()
int yyleng	member function: int YYLeng()

Exceptions about character class REs:

- For character class: special symbols like `*`, `+` lose their special meanings and you don't have to escape them. However, you still have to escape the following symbols: `\`, `-`, `]`, `^`, etc.
- There are some pre-defined special **character class expressions** enclosed inside `"[:"` and `"]"`, e.g.,

```
[:alnum:]  [:alpha:]  [:digit:]  
[:lower:]  [:upper:]
```

Some important command-line options:

Option	Meaning
<code>-d</code>	debug mode
<code>-p</code>	performance report
<code>-s</code>	suppress default rule; can find holes in rules
<code>-+</code>	generate C++ scanners

flex Example 5: Generating C++ Scanners

```
%option noyywrap

%{
int mylineno = 0;
%}

string  \("[^\\n"]+\\"
ws      [ \t]+
alpha   [A-Za-z]
dig     [0-9]
name    ({alpha}|{dig}|\$)({alpha}|{dig}|[_.\-/$])*
num1    [-+]?{dig}+\.\.?([eE] [-+]?{dig}+)?
num2    [-+]?{dig}*\.{dig}+([eE] [-+]?{dig}+)?
number  {num1}|{num2}

%%
{ws}    /* skip blanks and tabs */
{number} cout << "number " << YYText() << '\n';
{name}  cout << "name " << YYText() << '\n';
{string} cout << "string " << YYText() << '\n';
\n      ++mylineno;
```

flex Example 5: Generating C++ Scanners ..

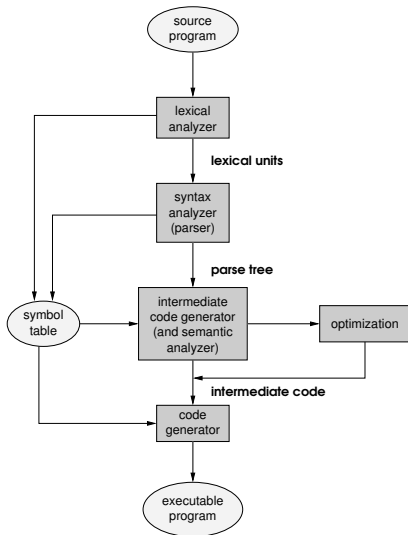
```
"/*"    { int c;
        while ((c = yyinput()) != 0)
        {
            if (c == '\n') {
                ++mylineno;
            } else if (c == '*') {
                if ((c = yyinput()) == '/') {
                    break;
                } else {
                    unput(c);
                }
            }
        }
    }
}
/* cout << "unrecognized " << YYText() << endl; */
%%

int main(int argc, char** argv)
{
    FlexLexer* lexer = new yyFlexLexer;
    while (lexer->yylex() != 0) {
    }
    return 0;
}
```

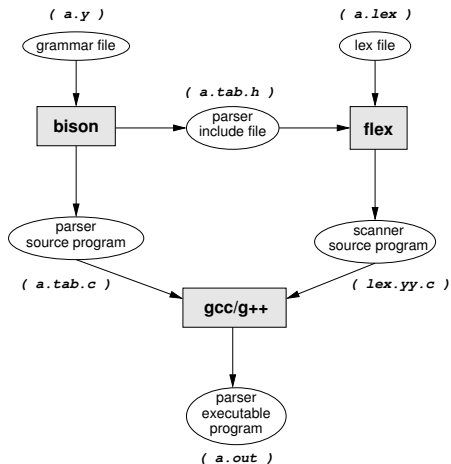
Part II

bison

Compilation (a revisit)



Syntax Analysis using bison and flex



bison : a Parser Generator

- bison is GNU's extended version of the standard UNIX utility `yacc`, that generates a **parser** for a given CFG. It is backward compatible with yacc (Yet Another Compiler Compiler), which was perhaps the first popular parser generator.
- bison reads a description of a CFG written in a **Grammar File**, and output a C program containing a routine called `yyparse()`.
- The default name of the output C program is `*.tab.c`. Compile `*.tab.c` to `a.out` which will be the parser.
- bison can only parse a subset of CFGs called **LALR(1) grammars**, using a bottom-up parsing algorithm with one look-ahead token.
- bison only generates a parser and does NOT provide a **scanner** automatically. To get both a parser and a scanner:
 - run both bison and flex
 - put the lexical analysis code in the section **Additional C Code**.

```
%{  
C Declarations  
%}  
  
bison Declarations  
  
%%  
Grammar Rules + Actions  
%%  
  
Additional C Codes
```

- Similar to flex, any statements between the `%{` and `%}`, as well as any **additional C code** will be copied verbatim to the output.

bison Example 1: Reverse Polish Notation Calculator

```
%{
#define YYSTYPE double
#include <math.h>
%}

%token NUM

%% /* grammar rules and actions follow */
input: /* empty */
    | input line
    ;

line: '\n'
    | exp '\n' { printf("\t%.10g\n", $1); }
    ;

exp: NUM { $$ = $1; }
    | exp exp '+' { $$ = $1 + $2; }
    | exp exp '-' { $$ = $1 - $2; }
    | exp exp '*' { $$ = $1 * $2; }
    | exp exp '/' { $$ = $1 / $2; }
    | exp exp '^' { $$ = pow($1, $2); }
    | exp 'n' { $$ = -$1; }
    ;
%%
```

bison Example 1 ..

```
/* additional C code */
#include <ctype.h>
#include <stdio.h>

int yylex(void)
{
    int c;

    while ((c = getchar()) == ' ' || c == '\t') ; /* skip white spaces */

    if (c == '.' || isdigit(c)) {
        /* process numbers */
        ungetc(c, stdin);
        scanf("%lf", &yylval);
        return NUM;
    }

    if (c == EOF) return 0;
    return c;
}

int main() { return yyparse(); }

int yyerror(const char* s) { printf("%s\n", s); return 0; }
```

① C Declarations

- define types and variables
- `#define`'s and `#include`'s

② bison Declarations

- declares names of the terminals/non-terminals symbols
- describe operator precedence and associativity
- data types of semantic values of variables

③ Grammar Rules

- production rules of the CFG

④ Additional C code

- definition of `yylex()`
- definition of `yerror()` and other supporting routines

Description of bison Grammar Rules

- Three ways to represent terminals:
 - ① character literals. e.g. '+' for the + operator.
 - ② C string constants. e.g. "else" for the keyword **else**.
 - ③ C-like identifiers. e.g. NUM (for numbers). The convention is to write it in upper case.
- Non-terminals are represented as C-like identifiers. The convention is to write them in lower case.
e.g. exp for <Expression>.
- Use : to represent ::=.
- A rule ends with a ';'.
- Example of a (production) rule in bison:

```
if-stmt : "if" bool-expr "then" stmt ';'
        | "if" bool-expr "then" stmt "else" stmt ';'
        ;
```


Type and Semantic Value

- Most terminals or tokens have

- 1 a **type**
- 2 a **semantic value**

e.g. the integer 123 has:

type : INTEGER
semantic value : one hundred twenty-three

- But *some* terminals do NOT. e.g. operator '+'.
• Non-terminals also have semantic values. e.g.
 - the semantic value of a math expression (e.g. $E = a + b$) is a real number — result computed from its constituents.
 - the semantic value of a compiled statement is a **parse tree**.

Semantic Actions

- Define the **semantics** of a program!
- Compute the **semantic value** of the non-terminal on the LHS of a grammar production rule based on the semantic values of the terminals and non-terminals on the RHS of the rule.
For example,

```
expr : expr '+' term    { $$ = $1 + $3 }
```

where

`$$` = semantic value of ‘`expr`’ on the LHS.

`$1` = semantic value of the 1st token on the RHS, which is the non-terminal ‘`expr`’.

`$3` = semantic value of the 3rd token on the RHS, which is the non-terminal ‘`term`’.

bison Types, Variables, Functions

Entity	Meaning
YYSTYPE	macro for the token type (default: int)
extern YYSTYPE <i>yylval</i>	value of an input token
extern int <i>yyparse</i> (void)	parser function
extern int <i>yyerror</i> (const char*)	error reporting function

- When no action is specified, the default action is: $$$ = 1 .
- Token type code of EOF = any non-positive value (including 0).

bison Example 1 again: rpn-calc.y

```
%{
#define YYSTYPE double
#include <math.h>
#include <stdio.h>
%}

%token NUM

%% /* Grammar rules and actions follow */
input: /* empty */
    | input line
    ;
line: '\n'
    | exp '\n' { printf("\t%.10g\n", $1); }
    ;
exp: NUM { $$ = $1; }
    | exp exp '+' { $$ = $1 + $2; }
    | exp exp '-' { $$ = $1 - $2; }
    | exp exp '*' { $$ = $1 * $2; }
    | exp exp '/' { $$ = $1 / $2; }
    | exp exp '^' { $$ = pow($1, $2); }
    | exp 'n' { $$ = -$1; }
    ;

%%

int main() { return yyparse(); }
int yyperror(const char* s) { printf("%s\n", s); return 0; }
```

bison Example 1 again: rpn-calc.tab.h

```
bison -d rpn-calc.y
```

produces 2 files:

- `rpn-calc.tab.h`: Some C declarations needed by the lex file
- `rpn-calc.tab.c`: Source program of the parser

```
#ifndef YYSTYPE
#define YYSTYPE int
#endif
#define NUM 257

extern YYSTYPE yylval;
```

bison Example 1 again: rpn-calc.lex

```
%option noyywrap

%{
#define YYSTYPE double      /* type for bison's var: yylval */
#include <stdlib.h>         /* for atof(const char*) */
#include "rpn-calc.tab.h"
%}

digits [0-9]
rn      (0|[1-9]+{digits}*)\.{digits}*
op      [+\-*/\^]
ws      [ \t]+

%%
{rn}    yylval = atof(yytext); return NUM;
{op}    |
\n      return *yytext;
{ws}    /* eats up white spaces */
%%

/* There is NO main function! */
```

bison Example 2: Infix-Notation Calculator

```
%{
#define YYSTYPE double
#include <math.h>
#include <stdio.h>
%}

%token NUM
%left '-' '+'
%left '*' '/'
%left NEG
%right '^'

%% /* Grammar rules and actions follow */
input:      /* empty */ | input line ;
line:      '\n' | exp '\n' { printf("\t%.10g\n", $1); } ;
exp: NUM { $$ = $1; }
    | exp '+' exp { $$ = $1 + $3; }
    | exp '-' exp { $$ = $1 - $3; }
    | exp '*' exp { $$ = $1 * $3; }
    | exp '/' exp { $$ = $1 / $3; }
    | '-' exp %prec NEG { $$ = -$2; }
    | exp '^' exp { $$ = pow($1, $3); }
    | '(' exp ')' { $$ = $2; } ;

%%

int main() { return yyparse(); }
int yyperror(const char* s) { printf("%s\n", s); return 0; }
```