Flex and Bison Tutorial

Ming-Hwa Wang, Ph.D
COEN 259 Compilers
Department of Computer Engineering
Santa Clara University

Flex

Flex is a scanner generator tool for lexical analysis, which is based on finite state machine (FSM). The input is a set of regular expressions, and the output is the code to implement the scanner according to the input rules.

To implement a scanner for calculator, we can write the file "cal1.l" as below:

```
/* this is only for scanner, not link with parser yet */
%{
int lineNum = 0;
%}

%%

"(" { printf("(\n"); }
")" { printf(")\n"); }
"+" { printf("+\n"); }
"*" { printf("*\n"); }
\n  { lineNum++; }
[ \t]+ { }
[0-9]+ { printf("%s\n", yytext); }

%%

int yywrap() {
   return 1;
}

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

Here is the Makefile used to build the scanner:

```
p1: lex.yy.o
   gcc -g -o p1 lex.yy.o

lex.yy.o: cal1.l
   flex cal1.l; gcc -g -c lex.yy.c

clean:
   rm -f p1 *.o lex.yy.c
```

Note: for more complex flex input file, you might get an error message like
`"parse tree too big, try %a num (or %e num)"`
Then you need to define %e <num>. You should put it between macro and %start symbol. The other options are %a, %o, %n, %p, etc.

Bison

Bison is a LALR(1) parser generator tool for syntax analysis, which is based on pushdown automata (PDA). The input is a set of context-free grammar (CFG) rules, and the output is the code to implement the parser according to the input rules.

To implement a parser for calculator, we can write the file "cal.y" as below:

```
%{
#include <stdio.h>
#include <ctype.h>
int lineNum = 1;
void yyerror(char *ps, ...) {        /* need this to avoid
link problem */
   printf("%s\n", ps);
}
%}

%union {
   int d;
}
// need to choose token type from union above
%token <d> NUMBER
%token '(' ')'
%left '+'
%left '*'
%type <d> exp factor term

%start cal

%%

cal
  : exp
   { printf("The result is %d\n", $1); }
  ;

exp
  : exp '+' factor
   { $$ = $1 + $3; }
  | factor
   { $$ = $1; }
  ;

factor
```

```
    : factor '*' term
      { $$ = $1 * $3; }
    | term
      { $$ = $1; }
    ;

term
    : NUMBER
      { $$ = $1; }
    | '(' exp ')'
      { $$ = $2; }
    ;

%%

int main() {
    yyparse();
    return 0;
}
```

To integrate both the scanner and parser, we need to modify the scanner input file "cal1.l" and save it as "cal.l" as below:

```
%{
#include <stdlib.h>     /* for atoi call */
#define DEBUG           /* for debuging: print tokens and
their line numbers */
#define NUMBER 258      /* copy this from cal.tab.c */
typedef union {         /* copy this from cal.tab.c */
    int d;
} YYSTYPE;
YYSTYPE yylval;  /* for passing value to parser */
extern int lineNum;     /* line number from cal.tab.c */
%}

%%

[ \t]+ {}
[\n]   { lineNum++; }
"(" {
#ifdef DEBUG
       printf("token '(' at line %d\n", lineNum);
#endif
       return '(';
    }
")" {
#ifdef DEBUG
       printf("token ')' at line %d\n", lineNum);
#endif
       return ')';
    }
"+" {
#ifdef DEBUG
       printf("token '+' at line %d\n", lineNum);
#endif
       return '+';
    }
"*" {
#ifdef DEBUG
       printf("token '*' at line %d\n", lineNum);
#endif
       return '*';
    }
[0-9]+ {
#ifdef DEBUG
    printf("token %s at line %d\n", yytext, lineNum);
#endif
    yylval.d = atoi(yytext);
    return NUMBER;
}

%%

int yywrap() {          /* need this to avoid link problem */
    return 1;
}
```

Here is the Makefile used to build the scanner and parser:

```
p2: lex.yy.o cal.tab.o
    gcc -o p2 lex.yy.o cal.tab.o

lex.yy.o: cal.l
    flex cal.l; gcc -c lex.yy.c

cal.tab.o: cal.y
    bison -d cal.y; gcc -c cal.tab.c

clean:
    rm -f p2 cal.output *.o cal.tab.c lex.yy.c
```

There are some tips to debug Bison.

1. Run Bison with –v option, then a file cal.output is generated. It contains all the conflicts and/or never reduced rules, and all the states generated by Bison.
2. To get debug information from Bison: first, add -DYYDEBUG when compiling cal.tab.c; second, set the environment variable YYDEBUG=1. Then it will print a bunch of debug information, e.g. how to shift or reduce.