More on Flex and Bison

Comp3031 Lab 07
Fall 2014

SU Zhiyang, JIA Xiaoying

Flex

- Flex is used to generate a lexical analyzer (scanner)

- The executable file is the generated scanner
  - Use the scanner to process text files

Flex File Structure

Definitions

Rules

User codes

```
%option noyywrap

%{
#include <stdio.h>
%}
%%
[a-yA-Y] printf("%c", *yytext+1);
[zZ]     printf("%c", *yytext-25);
.        printf("%c", *yytext);
%%
int main(int argc, char **argv)
{
    yylex();
    return 0;
}
```

Bison

- Generate a parser for a context free grammar
- Bison is often used together with Flex
  - Write a .lex file to generate a lexical scanner
  - Write a .y file to generate a parser based on the lexical scanner

Bison File Structure

C declarations

Bison declarations

Grammar rules

Additional C codes

```
%{
#define YYSTYPE double
#include <stdio.h>
%}
%token NUM
%% /* Grammer 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; }
;
%%
int main() { return yyparse(); }
int yyerror(const char* s)
{ printf("%s\n", s); return 0; }
```

Boundary Between Flex & Bison

Flex: tokenizer

- Tokenize the input stream

Bison: parser

- Parsing based on the given grammar

```
zsuab@ras1:~/lab/regextoken$ ./regextokenizer
ab*
CHAR CHAR OP RET
a(bc)+
CHAR OP CHAR CHAR OP OP RET
```

```
zsuab@ras1:~/lab/regex$ ./regex
ab*
        1
a(bc)+
        3
```

Example: Regex Exercise

```
%option noyywrap

%{
#define YYSTYPE int
#include "regex.tab.h"
%}

op   [()*?+]
ws   [ \t]+

%%
[a-z]    return CHAR;
{op}|\n  return *yytext;
{ws}     /* eat up white spaces */

%%
```

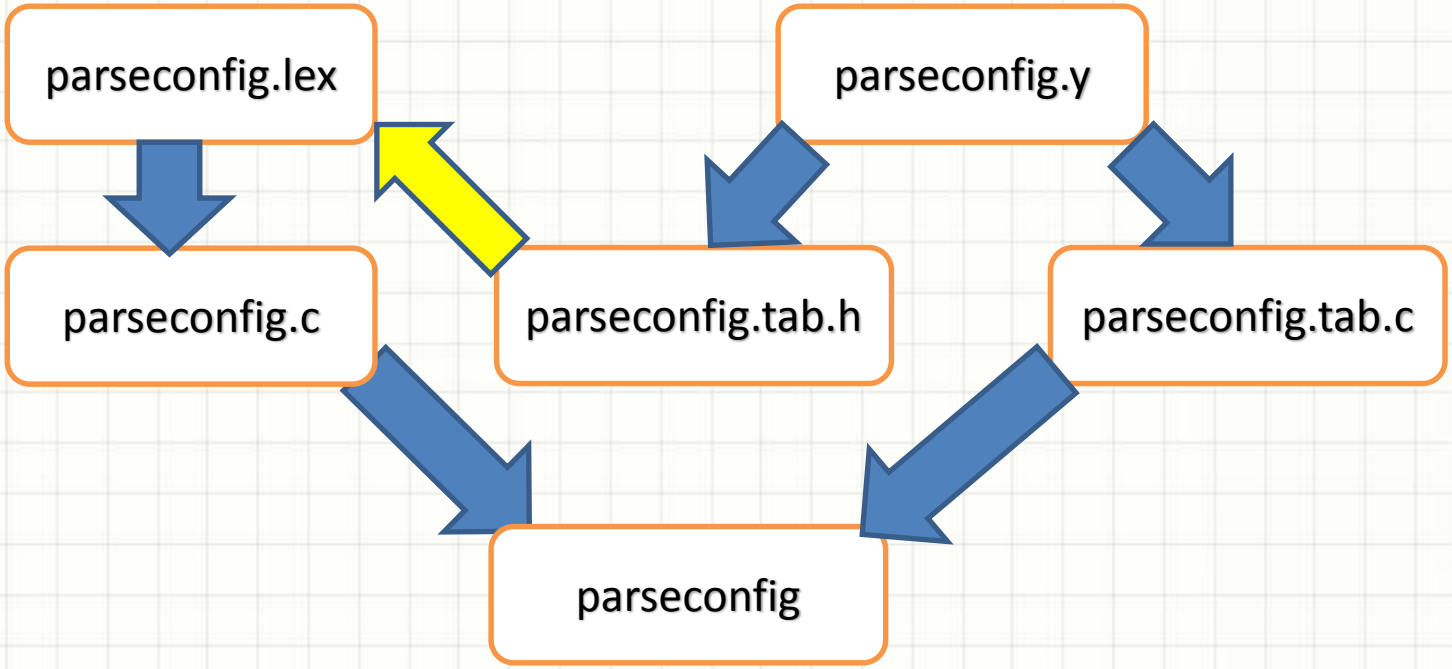
Flex: regex.lex

```
%token CHAR

%%
input: /* empty */
     | input line
     ;
line: '\n'
     | expr '\n' { printf("\t%d\n", $1); }
     ;
expr: expr term {$$ = $1 + $2;}
     | term     {$$ = $1;}
     ;
term: unit '?'  {$$ = 0;}
     | unit '*' {$$ = 0;}
     | unit '+' {$$ = $1;}
     | unit     {$$ = $1;}
     ;
unit: '(' expr ')' { $$ = $2; }
     | CHAR      { $$ = 1; }
     ;
%%
```

Bison: regex.y

Workflow of Parseconfig



- What Make do?
  - bison -d parseconfig.y
  - flex -oparseconfig.c parseconfig.lex
  - bison -d parseconfig.y
  - gcc -o parseconfig parseconfig.c parseconfig.tab.c

BNF for Parseconfig

$$< S >::= interface < NAME >< POOLBLOCK >< LIST >$$
$$< POOLBLOCK >::= \{< POOLS >\}$$
$$< POOLS >::= < empty > | < POOLS >< POOLSLASH > | < POOLS >< POOLRANGE >$$
$$< POOLSLASH >::= pool < IP > / < PREFIX >;$$
$$< POOLRANGE >::= pool < IP > - < IP >;$$
$$< LIST >::= < empty > | < LIST >< WHITELIST > | < LIST >< BLACKLIST >$$
$$< WHITELIST >::= whitelist\{< WIPLIST >< IP >;\}$$
$$< WIPLIST >::= < empty > | < WIPLIST >< IP >,$$
$$< BLACKLIST >::= blacklist\{< BIPLIST >< IP >;\}$$
$$< BIPLIST >::= < empty > | < BIPLIST >< IP >,$$

Bison

$$< NAME >::= < LETTER >< LETTERORDIGIT >$$
$$< LETTERORDIGIT >::= < empty > | < LETTERORDIGIT >< LETTER > | < LETTERORDIGIT >< D >$$
$$< LETTER >::= a|b|c \ldots z|A|B|C \ldots Z$$
$$< PREFIX >::= < D > |1 < D > |2 < D > |30|31|32$$
$$< IP >::= < I > . < I > . < I > . < I >$$
$$< I >::= < D > | < N >< D > |1 < D >< D > |2 < F >< D > |25 < V >$$
$$< D >::= 0|1|2|3|4|5|6|7|8|9$$
$$< N >::= 1|2|3|4|5|6|7|8|9$$
$$< F >::= 0|1|2|3|4$$
$$< V >::= 0|1|2|3|4|5$$

Flex

Basic Idea

- Read parseconfig.y and BNF
  - Find out what tokens are needed
- Add missing tokens to lex
  - And pass these tokens to bison
  - Type conversion
    - Char * -> int: atoi()
- Write bison grammar rules

Step by Step

- Make a simple lexer first
  - Tokenize the input file
  - Check if you correctly “split” the file
- Change “printf” to “return TOKEN;”

```
zsuab@ras1:~/assign2/lexer$ ./lexer config.txt
INTERFACEeth0
LBRACE
POOL 0.0.18.0PREFIX ;
POOL 0.0.1.100-0.0.1.105;
RBRACE
WHITELIST
LBRACE
0.0.1.2,0.0.1.201,0.0.3.4;
RBRACE
BLACKLIST
LBRACE
0.0.18.35,0.0.1.103,0.0.18.3;
RBRACE
WHITELIST
LBRACE
0.0.18.20;
RBRACE
```

Remarks

- Slightly difference between definition and implementation
  - prefix [/][\t]*{digits}+
    - Match “/30”, “/ 20”
  - Pass prefix to bison
    - {prefix} {yylval = getprefixnum(yytext); return PREFIX;}
- How to represent <empty>?
  - <S>::=<empty>|<A>
  - <S>::=<A>

```
S:
| A
;
```

```
S:
A
;
```

Remarks

- Process punctuation marks carefully
  - Especially in writing bison grammar rules
  - Write lex rules to match them and pass them to bison
- Complete missing bison grammar rules

```
%% /* Grammer rules and actions follow */
commands: /*empty*/
| commands command
;
command:
INTERFACE WORDS poolblock list {printf("%s:\n", $2);}
;
/* Start: add your grammar rules here */
```

Remarks: Debug

- We defined yyerror() in parseconfig.y

```
int yyerror(const char* s)
{
    extern int yylineno;
    extern char *yytext;
    printf("\n^%d: %s at %s #%d\n", yylineno, s, yytext, (int)(*yytext));
    return 0;
}
```

- You can debug your grammar rules by the output information

```
zsuab@ras1:~/assign2/skeleton$ ./parseconfig config1.txt

^1: syntax error at interface #105
```

  – Line 1 has a syntax error, when matching "interface", the first character's ascii: 105

Remarks

- Write program on Unix machines
  - csl2wk**.cse.ust.hk (**=01..40)
- Editors
  - emacs, vim, nano
- Make sure your program can be compiled & run properly
  - Otherwise, you may get 0 marks!

Demo

```
zsuab@ras1:~/assign2/sol$ make
bison -d parseconfig.y
flex -oparseconfig.c parseconfig.lex
make: Warning: File `parseconfig.tab.c' has modification time 1.4 s in the future
gcc -o parseconfig parseconfig.c parseconfig.tab.c
make: warning:  Clock skew detected.  Your build may be incomplete.
zsuab@ras1:~/assign2/sol$ ./parseconfig config.txt
eth0:
0.0.1.2
0.0.1.100
0.0.1.101
0.0.1.102
0.0.1.104
0.0.1.105
0.0.1.201
0.0.3.4
0.0.18.1
0.0.18.2
0.0.18.20
```

Q & A