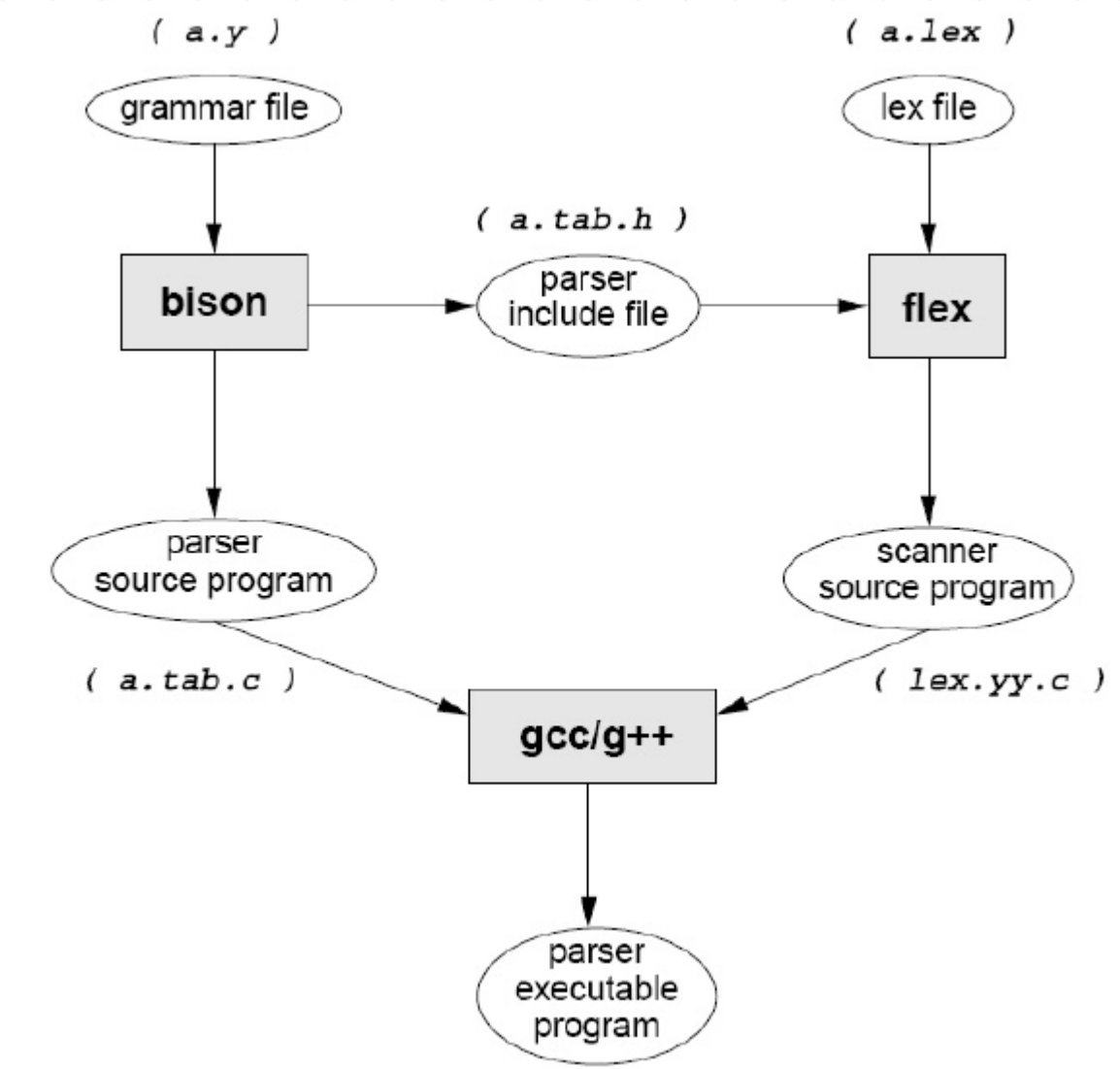
FLEX AND BISON

Comp3031 Lab 05
Fall 2014

SU Zhiyang, JIA Xiaoying

Overview about Flex & Bison



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;
}
```

Definition Section

- Name definition form

```
name pattern
```

  - name is the name of your macro and pattern is a regular expression
- Use previously defined names in your pattern
  - By its corresponding name surrounded by curly braces
  - The occurrence of "{name}" corresponds to (pattern)

```
digit [0-9]
group {digit}{4}
```

Definition Section

- Names can start with a letter or an underscore (‘_’)
- Comments
  - Directly output to the .c file
- Other codes
  - Directly output to the .c file

```
/* Comment here */
name definitions
```

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

Rule Section

- Rules form

```
pattern action
```

  - A pattern is a regular expression
  - The action is normally C/C++ code to do something with the string that matches the pattern
  - The string that matches the pattern is placed in a pre-defined char * variable named yytext
  - Actions are normally enclosed in curly braces ({})

User Code Section

- Place code to invoke the scanner
  - Any valid C/C++ code
- This section will be directly copied to the .c file
- It usually includes
  - main()
  - Other user defined functions

Indentation in Flex

- Flex is very finicky about indentation

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

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

  - Flex will complain that you have many errors
  - Or, gcc will produce dozens of error messages
- Suggestions
  - Start all flex commands on the first column

Flex Example

- Generate the scanner
  - flex -oroundchar.yy.c roundchar.lex
  - g++ -o roundchar roundchar.yy.c
  - ./roundchar

```
zsuab@ras1:~/flex$ flex -oroundchar.yy.c roundchar.lex
zsuab@ras1:~/flex$ g++ -o roundchar roundchar.yy.c
zsuab@ras1:~/flex$ ./roundchar
abcdefghijklmn
```

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; }
```

C Declaration Section

- Define types and variables used in the actions

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

Bison Declaration Section

- Bison declarations
  - Terminal symbols

```
%token CHAR
```

  - Non-terminal symbols
  - yyerror() function

Grammar Rule Section

- Bison grammars are similar to the BNF form

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

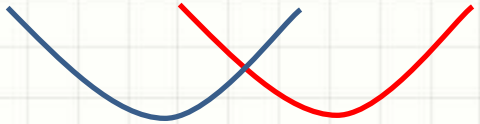
Production rule Action

- Notice that we use ":" instead of "::="
- The action is plain C code

Grammar Rule Section

- Semantics values

```
exp: NUM { $$ = $1; }
```



  - $$ is the semantic value of exp
  - $1 is the semantic value of NUM
  - If there are more components in the rule, we use $2, $3… to represent their semantic values
- Their value type
  - YYSTYPE

Additional C Code Section

- Contains all the rest of C code

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

Exercise example

- Write a program which reads in a regular expression and outputs the length of the shortest string(s) that the regular expression accepts
- The grammar
  - <Expr> ::= <Expr> <Term> | <Term>
  - <Term> ::= <Unit> <Op> | <Unit>
  - <Unit> ::= (<Expr>) | <Id>
  - <Op> ::= * | ? | +
  - <Id> ::= a | b | ... | z

Demonstration

- Download these two files
  - regex.lex
  - regex.y
- Run the following commands to compile
  - bison -d regex.y
  - flex -oregex.lex.yy.c regex.lex
  - gcc -o regex regex.lex.yy.c regex.tab.c
- "./regex" to run

Exercise

- Modify regex.y to output the number of non-letter characters ('(', ')', '*', '+', and '?')
- Example

```
ab*
 1
a(bc)+
 3
```