SKENZ.IT How To Wiki

Return to Home page

Flex Bison

Flex and Bison are tools for building programs that handle structured input. They were originally used to developing compilers, but they have proven to be useful in many other areas. They are modern replacements for the classic Lex and Yacc.

Both of them are available on Linux via APT. Windows versions are available as well:
Flex for Windows [http://gnuwin32.sourceforge.net/packages/flex.htm]
Bison for Windows [http://gnuwin32.sourceforge.net/packages/bison.htm]

The examples shown in the following paper and many others (Laboratories, exams and exercises done in class) are available here:
Flex Bison examples [https://www.skenz.it/compilers/flex_bison_examples]

Flex

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 file for a description of the 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 file, lex.yy.c by default, which defines a routine yylex(), that performs the real scanning.

Input files

A Flex input file is composed of three sections, with a line containing only %% between each of them.

```
Definitions section
%%
Rules section
%%
User code section
```

Definitions section

The definitions section is used to simplify the scanner specification by assigning a name to some regular expressions. A definition has the form:

```
name definition
```

where the name is a word beginning with a letter or an underscore ('_') followed by zero or more letters, digits, '_', or '-' (dash). The definition is taken to begin at the first non-whitespace character following the name and continuing to the end of the line. This is an example of declaration:

```
digit [0-9]
```

Optionally, a %top block can be added in this section. A %top block is copied verbatim to the top of the scanner implementation file and it is useful to define macros or include header files.

```
%top{
 /* This code goes at the top of the generated file. */
 #include <stdio.h>
}
```

Rules section

The rules section contains a series of rules in the form:

```
pattern action
```

where a pattern can be either a regular expression (enclosed in brackets) or a name defined in the previous section. The action consists in a block of C code. For example:

```
({letter}:)?(\\)?({id}\\)*{id}("."{id})? {
  printf("Path Correct: %s\n", yytext);
}
```

yytext points to the first character of the match in the input buffer. The string itself is part of the input buffer, and is not allocated separately. The value of yytext will be overwritten the next time yylex() is called. If the value of yytext needs to be preserved, utilities like strdup() can be adopted.

User code section

The user code section is simply copied to lex.yy.c verbatim. It is used for companion routines which call or are called by the scanner. The presence of this section is optional; if it is missing, the second %% in the input file may be skipped, too.

Comments

Comments may appear just about anywhere, with the following exceptions:

- Comments are forbidden whenever flex expects a regular expression. This means comments may not appear at the beginning of a line, or immediately following a list of scanner states.
- Comments may not appear on an '%option' line in the Definitions Section.

A first example

This basic example describes a scanner which recognizes a correct pathname.

Laboratories/Lab1/Lab1_ex1/scanner.l

```
1. %option noyywrap
2. digit [0-9]
3. letter [^\n\r\\/:*?"<>|{digit}]
4. id ({digit}|{letter})+
5. newLine \n|\r|\r\n
6. %%
7. ({letter}:)?(\\)?({id}\\)*{id}("."{id})? {
8.   printf("Path Correct: %s\n", yytext);
9. }
10.
11. .+ {
12.   printf("Error in path: %s\n", yytext);
13. }
14.
15. {newLine} {;}
16.
17. %%
18. int main(int argc, char const *argv[]) {
19.   yyin = fopen(argv[1], "r");
20.   yylex();
21.   fclose(yyin);
22.   return 0;
23. }
```

When the scanner function **yylex()** receives an end-of-file indication, it invokes yywrap(): if it returns false (zero), then yylex() assumes that **yyin** (the global input file) has been redirected to another source, and scanning continues; if it returns true (non-zero), then the scanning process terminates. Option noyywrap make yywrap() to return always true.
The scanner will ignore newlines thanks to the empty action at line 15.
The code section is used to implement the main function, which is quite straightforward: it opens a file whose pathname is received as argument and invokes yylex(), then closes the file.

Execution

In order to generate the scanner implementation file lex.yy.c, the following command has to be run:

```
flex scanner.l
```

Of course the this .c file has to be compiled:

```
gcc -o main lex.yy.c
```

Finally, the whole scanner can be run:

```
./main example.txt
```

The input file is available here: example.txt [https://www.skenz.it/repository/compilers/flex_bison/Laboratories/Lab1/Lab1_ex1/example.txt]

States

Flex provides a mechanism for conditionally activating rules. Any rule whose pattern is prefixed with <sc> will only be active when the scanner is in the state named sc. States are declared in the definitions section of the input using unindented lines beginning with either '%s' or '%x' followed by a list of names. The former declares inclusive states, the latter exclusive states. If the state is inclusive, then rules with no states at all will also be active. If it is exclusive, then only rules qualified with the start condition will be active. If the distinction between inclusive and exclusive states is still a little vague, here's a simple example illustrating the connection between the two. The set of rules:

```
%s example
%%
<example>foo do_something();
bar something_else();
```

is equivalent to

```
%x example
%%
<example>foo do_something();
<INITIAL,example>bar something_else();
```

A state is activated using the BEGIN action and specifying the next state, For example, here is a scanner which recognizes (and discards) C comments.

```
%x comment %%
"/*" {BEGIN(comment);}
<comment>. {;}
<comment>\n|\r|\r\n {;}
<comment>"*/" {BEGIN(INITIAL);}
```

Bison

Introduction

Bison is a parser generator: it receives a sequence of tokens and verifies if it belongs to a specified grammar. Even though the output parser can be produced also in C++ and Java, the following paper will focus on the techniques to create a parser written in the C programming language. In order to work Bison needs two functions:

1. The lexical analyzer function **yylex()** that recognizes tokens and return them to the parser.
2. The error reporting function **yyerror()**, invoked by the parser whenever it reads a token which cannot satisfy any grammar rule.

Starting from the input file, Bison creates the parser implementation file *parser.tab.c* which contains the parsing function **yyparse()**.

Input files

A Bison input file has four main sections:

```
%{
Prologue section
%}
Declarations section
%%
Grammar rules section
%%
Epilogue section
```

Comments enclosed in '/* ... */' may appear in any of the sections. As a GNU extension, '//' introduces a comment that continues until end of line.

Prologue section

The Prologue section contains macro definitions and declarations of functions and variables that are used in the actions of the grammar rules. As an alternative, Bison provides a %code directive with an explicit qualifier field, which identifies the purpose of the code and thus the location where Bison should generate it. The unqualified code directive is equivalent to the prologue section.
Available qualifiers are:

- **requires:** used for dependency code for YYSTYPE and YYLTYPE.
- **provides:** code which should be available to other modules.
- **top:** similar to the unqualified code directive, but code goes much nearer the top of the parser implementation file.

Declarations section

There are three ways in which terminal symbols in the grammar can be defined:

- A *named token type*

```
%token [<type>] name..
```

- A *character token type*

```
%token <int> 'n'
```

- A *string literal token*

```
%token <int> "integer"
```

A non-terminal symbol declaration has the syntax:

```
%type <type> nonterminal..
```

or

```
%nterm <type> nonterminal..
```

As well as character and string literal tokens ones, declarations of non-terminal symbols are mandatory only if the semantic data types needs to be specified.

Grammar rules section

A grammar rule is composed by a left hand side and a right hand side.

```
left_hand_side: right_hand_side_component1 right_hand_side_component2 ..
```

Multiple rules with the same left hand side can be condensed by using the vertical bar character | as follows:

```
left_hand_side: right_hand_side1 | right_hand_side2
```

Scattered among the components of the right hand side can be actions, which is a block of C code. Occasionally it is useful to put an action in the middle of a rule. These actions are written just like usual end-of-rule actions, but they are executed before the parser even recognizes the following components.

Epilogue section

This section is copied verbatim at the end of the parser implementation file. If it is empty, the last %% can be omitted.

Interfacing Flex and Bison

In order to make Flex and Bison to talk each other, it suffices to make them share the same list of available tokens. This task can be easily achieved by adding this line of code in the prologue of the scanner input file:

```
#include "parser.tab.h"
```

Thanks to a compilation option, *parser.tab.h* is produced together with *parser.tab.c* and contains the prototype of the parsing function yyparse() as long as the declaration of a series of constants, one for each token declared in the parser input file.
It has been decided to make use of an additional file, *main.c*, to write the error reporting function yyerror() and the main function. The choice of using a different file is arbitrary: one could also write them in the epilogue section of the parser input file as well as in the user code section of the scanner input file.
This is a basic implementation of yyerror:

```
void yyerror(char const *message){
  printf("Error: %s\n", message);
}
```

It simply prints on the standard output the message received as parameter. Normally the string is “syntax error”, but the option '%define parse.error verbose' allows bison to provide more detailed messages.
The sole responsibility of the main function is to set correctly the global input file yyin and to invoke yyparse:

```
int main(int argc, char const *argv[]) {
  yyin = fopen(argv[1], "r");
  int result_code = yyparse();
  fclose(yyin);
  return result_code;
}
```

Putting them together:

Laboratories/Lab3/Lab3_ex2/main.c

```
 1. #include <stdio.h>
 2. #include "lex.yy.h"
 3.
 4. extern int yyparse (void);
 5.
 6. void yyerror(char const *message){
 7.   printf("Error: %s\n", message);
 8. }
 9.
10. int main(int argc, char const *argv[]) {
11.   yyin = fopen(argv[1], "r");
12.   int result_code = yyparse();
13.   fclose(yyin);
14.   return result_code;
15. }
```

Directive include at line 2 allows yyin to be visible to the entire file. To make the scanner produce this header file, the following option needs to be inserted in the scanner source file:

```
%option header-file="lex.yy.h"
```

In this first example, the couple scanner/parser will recognize a language devoted to the management of a library (Laboratory 3 exercise 2).
In order to send a token to yyparse(), yylex just needs to return a constant from those defined in *parser.tab.h*:

Laboratories/Lab3/Lab3_ex2/scanner.l

```
%option header-file="lex.yy.h"
%option noyywrap

%top{
  /* This goes at the top of the generated file */
  #include "parser.tab.h"
}
isbn [0-9]{2}-[0-9]{2}-[0-9a-fA-F]{5}-[0-9a-fA-Z]
id \"[A-Za-z0-9 .,:]+\"
date {day}\/{month}\/{year}
day 0[1-9]|[1-2][0-9]|3[0-1]
month 0[1-9]|1[0-2]
year [0-9]{4}
%%
AV {return AV;}
BO {return BO;}
SO {return SO;}
LI {return LI;}
LS {return LS;}
[A-Z][A-Z]* {return LETTER;}
[0-9][0-9]* {return INTEGER;}
{isbn} {return ISBN;}
{id} {return ID;}
{date} {return DATE;}
"->" {return ARROW;}
, {return COMMA;}
; {return SC;}
: {return COLON;}
"%%" {return FILE_SEPARATOR;}
"\n"|"\r"|("\r\n") {;}
. {;}
```

In the parser source file, functions yylex() and yyerror() needs to be declared as extern, since yyparse() cannot work without them.

Laboratories/Lab3/Lab3_ex2/parser.y

```
%{
  /* This is the prologue section. This code goes
  on the top of the parser implementation file. */
  #include <stdio.h>
  extern int yyerror(char *message);
  extern int yylex(void);
%}

/* Declarations of terminals */
%token  FILE_SEPARATOR COMMA ID S COLON
        ARROW INTEGER LETTER ISBN LI LS
        BO AV SO DATE;

%%
/* Grammar rules */
file: writer_section FILE_SEPARATOR user_section
{printf("File correctly parsed.\n");};
writer_section: writer_section writer| writer;
writer: ID ARROW book_list SC;
book_list: book_list COMMA book | book;
book: ISBN COLON ID COLON INTEGER COLON collocation;
collocation:  /* Optional */
            | lit_gen INTEGER LETTER
            | lit_gen INTEGER;
lit_gen: LI AV | LI SO | LS AV | LS SO | LS BO;
user_section: user_section user | user;
user: ID COLON loan_list SC;
loan_list: loan_list COMMA loan | loan;
loan: DATE ISBN
```

Line 22 is an example of the so-called *empty rule*: a rule that matches the empty string. By convention is used to write a comment in each rule with no components.

Execution

Implementation files are generated using flex and bison:

```
flex scanner.l
bison -d parser.y
```

'd' stands for 'define' and allows the creation of the additional file *parser.tab.h* which contains macro definitions for the token type names defined in the grammar. Then it is possible to proceed with the actual compilation:

```
gcc -c -o scanner.o lex.yy.c
gcc -c -o parser.o parser.tab.c
gcc -c -o main.o main.c
```

and linking:

```
gcc -o main parser.o scanner.o main.o
```

Finally, the parser can be run:

```
./main example.txt
```

The input file is available here: example.txt [https://www.skenz.it/repository/compilers/flex_bison/Laboratories/Lab3/Lab3_ex2/example.txt]

Error handling and recovery

It is not usually acceptable to have a program terminate on a syntax error. For example, a compiler should recover sufficiently to parse the rest of the input file. It is possible to define how to recover from a syntax error by writing rules to recognize the special token error. This is a terminal symbol that is always defined (there is no need to declare it) and reserved for error handling. The Bison parser generates an error token whenever a syntax error happens; if a rule to recognize this token in the current context is present, the parsing can continue. For example:

```
stmt: error ';' /* On error, skip until ';' is read. */
```

If it is necessary, Bison makes the programmer able to start explicitly error recovery thanks to the macro YYERROR. As said previously, whenever the parser runs into a syntax error, the error reporting function yyerror() is invoked. At the same time, the variable yynerrs is incremented, as it counts the number of errors encountered during the parsing process. yyerror() can be also invoked programmatically.

Locations

To aid in error reporting, Bison offers locations, a feature to track the source line and column range for every symbol in the parser. Locations are enabled explicitly with %locations. The lexer has to track the current line and column and set the location range for each token in a variable called yylloc before returning the token. yylloc has type YYLTYPE, which by default is defined in this way:

```
typedef struct YYLTYPE {
    int first_line;
    int first_column;
    int last_line;
    int last_column;
} YYLTYPE;
```

In order to update yylloc a little-known feature of Flex can be adopted: YY_USER_ACTION is a macro automatically invoked for each token recognized by yylex. This is a possible implementation of YY_USER_ACTION:

```
#define YY_USER_ACTION \
    yylloc.first_line = yylloc.last_line; \
    yylloc.first_column = yylloc.last_column; \
```

```
    for(int i = 0; yytext[i] != '\0'; i++) { \
        if(yytext[i] == '\n') { \
            yylloc.last_line++; \
            yylloc.last_column = 0; \
        } \
        else { \
            yylloc.last_column++; \
        } \
    }
```

Precedence rules

Sometimes, the programmer has to choose between a simple grammar full of conflicts and a complex grammar which is conflict-free. In this context, the idea of operator precedence is crucial: precedence directives allow the programmer to keep the grammar simple and avoid conflicts. Bison offers the directives %left and %right to specify the associativity of symbols. The concept of associativity can be clarified with the following example.
Consider the input '1 - 2 - 5': should this be '(1 - 2) - 5' or should it be '1 - (2 - 5)'?
For most operators, the former, called left association, is preferred. The latter, right association, is desirable for assignment operators. The following directive declare symbol '-' as left-associative:

```
%left '-'
```

The relative precedence of different operators is controlled by the order in which they are declared.

```
%left '-'
%left '*'
```

In this case, the '*' operator has precedence over '-' operator.

Data types of Semantic values

The grammar rules for a language determine only the syntax. The semantics are determined by the semantic values associated with various tokens. YYSTYPE is the type of those tokens, and by default is equal to the int type. If necessary, it is possible to specify a different type:

```
#define YYSTYPE double
```

It is important to notice that in this way different data types for tokens are forbidden. To use more than one data type for semantic values in one parser, Bison requires two things:

- Specify the entire collection of possible data types. These are the main options:
    - use the %union Bison declaration
    - use a define directive and declare YYSTYPE to be a union.
- Include in the declaration of terminal and non-terminal the type linked to the symbol.

For example, consider this section of file parser.y belonging to the solution of Laboratory 5:

```
%union {
  float real_value;
  float *array;
  char *string;
}
//..
%type     <real_value>    scalar_expr scalar
//..
%token    <real_value>    NUM
```

In this case, non-terminals scalar_expr and scalar, as well as terminal NUM have been declared as symbols linked to a float value.

References

- Flex & Bison, John R. Levine
- Lexical analysis with Flex, Vern Paxson, Will Estes and John Millaway

- Bison, Charles Donnelly and Richard Stallman

If you found any error, or if you want to partecipate to the editing of this wiki, please contact: admin [at] skenz.it

---

/web/htdocs/www.skenz.it/home/data/pages/compilers/flex_bison.txt · Last modified: 2020/11/26 23:18 (external edit)

Privacy Policy