Yacc/Bison

Bison

Yacc (yet another compiler-compiler) is a LALR[a] parser generator created by S. C Johnson. Bison is an yacc like GNU parser generator[b].
It takes the language specification in the form of an LALR grammar and generates the parser.

---

[a] It can handle some amount of ambiguity. See reference (9) of the list of books.

[b] Bison has facility for generalized LR parsing. But that parser is slower and we shall not use it: `%glr-parser`

Input

Bison takes the parser specification from a file. Following the convention of yacc, the file name extension is .y[a]. The output file name by default uses the prefix of the input file and is named as <prefix>.tab.c[b].
The output file generated by yacc is named as y.tab.c. The Bison with -y command-line option will also generates this.

---

[a]If C++ output is required, the specification file extension should be .y++ or .ypp.

[b]<prefix>.tab.c++ or <prefix>.tab.cpp

Input

A `bison` input file (`bison` grammar file) has the following structure (three sections) with special punctuation symbols `%%`, `%{` and `%}`.

```
%{
Prologue e.g. C or C++ declarations
%}
bison declarations
%%
Grammar rules
%%
Epilogue e.g. Additional C or C++ code
```

Note

- The first two sections are required (although they may be empty).
- The last section with the third **%%** may be absent.

Example

We start with the following expression grammar: $\Sigma = \{\ \texttt{+ - * / ( ) fc ic}\ \}$, $N = \{\ \texttt{E}\ \}$, the start symbol is `E`, and the production rules are,

$$
\begin{aligned}
E \;\rightarrow\; & E + E \mid E - E \mid E * E \mid E/E \\
& \mid -E \mid +E \mid (E) \mid \texttt{fc} \mid \texttt{ic}
\end{aligned}
$$

Our goal is to implement a calculator using Flex and Bison software.

flex Specification: exp.l

```
%{
/*
 * exp.l is the flex specification for
 * exp.y++. The exp.tab.h++  will
 * be generated by bison compiler.
 * Compile as
 * $ flex exp.l
 * output: lex.yy.c
 */
#include <stdio.h>
#include <stdlib.h>
#include "exp.tab.h++"  /* Generated by bison */
```

```
    /* Copied verbatim in lex.yy.c */
%}

%option noyywrap

DELIM       ([ \t])
WHITESPACES ({DELIM}+)
NATNUM     ([0-9]+)
FLOAT       (([0-9]*\.[0-9]+)|([0-9]+\.[0-9]*))

%%
{WHITESPACES} { ; }
{NATNUM}      {
                yylval.integer = atoi(yytext);
```

```
                    return INT ;
                  }
{FLOAT}           {
                    yylval.real = (float)atof(yytext);
                    return FLOAT;
                  }
"+"               { return (int)'+' ; }
"-"               { return (int)'-' ; }
"/"               { return (int)'/' ; }
"*"               { return (int)'*' ;}
"\n"              { return (int)'\n';}
"("               { return (int)'(';}
")"               { return (int)')';}
%%
```

```
/* No C++ code */
```

Note

The `flex` specification will be compiled by the command

```
$ flex exp.l
```

The output file (C code for the scanner) `lex.yy.c` is generated.

The header file `exp.tab.h++` will be created by the parser generator bison.

bison Specification: exp.y++

```
/*
 * bison specification for infix calculator.
 * Compile as follows:
 * $ bison -d exp.y++
 * output: exp.tab.c++ and exp.tab.h++
 * $ bison -y -d exp.y
 * same as yacc -d exp.y
 */

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

```
using namespace std;
int yylex (void);       /* type of yylex()  */
void yyerror(char const *s);
#define YYDEBUG 1       /* enables compilation with trace fac
                        /* copied verbatim to exp.tab.c++  */
%}

%union  {               /* type of 'yylval' (value stack type
int integer ;           /* type name is YYSTYPE
float real ;            /* default #define YYSTYPE int ple ty
}

%token <integer> INT <real> FLOAT  /* tokens and types   */
%type  <real> exp       /* nonterminal and its type      */
```

```
                        /* non-terminal symbols are      */
                        /* lower-case by convention      */
%left '-' '+'           /* left associative character    */
%left '*' '/'           /* tokens: 'nonassoc', 'right'   */
%left UNEG UPOS         /* precedence of unary + -       */
                        /* + - lowest precedence         */
                        /* * / next higher               */
                        /* unary + - is the highest      */


%start s                /* start symbol                  */

%% /* Grammar rules and action follows   */
```

```
s:  s line
  |       /* Empty line */
;

line:     '\n'
  | exp   '\n'                { cout << $1 ; }
  | error '\n'                { yyerrok ;    }
;        /* 'error' is a special token and yyerrok()
          * is a macro defined by Bison
          */

exp:     INT                 {  $$ = (float)$1;}
   |     FLOAT               /* Default action $$ = $1; */
   | exp '+' exp             {  $$ = $1 + $3 ; }
```

```
    | exp '-' exp          {  $$ = $1 - $3 ; }
    | exp '*' exp          {  $$ = $1 * $3 ; }
    | exp '/' exp          {
                  if($3 == 0) yyerror("Divide by zero");
                  else $$ = $1 / $3 ;
                           }
    | '-' exp %prec UNEG   {  $$ = - $2 ;    }  /* Context de
    | '+' exp %prec UPOS   {  $$ =  $2 ;     }  /* precedence
    | '(' exp ')'          {  $$ =  $2 ;     }
;

%%
int main()
{
```

```
//  yydebug = 1 ;           To get trace information
    return yyparse() ;
}

/*
 * called by yyparse() on error
 */
void yyerror(char const *s) {cerr << s;}
```

Note

The `bison` specification will be compiled by the command[a] `$ bison -d exp.y++`
The output files (C/C++ code for the parser and the header file) `exp.tab.c++` and `exp.tab.h++` are generated.
If the option `-v` is given,
`$ bison -d -v exp.y`
the bison compiler creates a file `exp.output` with the description of the parser states.

[a] If `bison` is expected to behave like `yacc`, the option is `$ bison -y -d exp.y`

Makefile

```
src = exp
objfiles = $(src).tab.o lex.yy.o

calc :  $(objfiles)
   c++ $(objfiles) -o calc

$(src).tab.c++ : $(src).y++
    bison -d $(src).y++

lex.yy.c : $(src).l calc.h
   flex $(src).l
```

```
$(src).tab.o: $(src).tab.c++ calc.h
	c++ -Wall -c $(src).tab.c++

lex.yy.o : lex.yy.c
	c++ -Wall -c lex.yy.c

clean :
	rm calc $(src).tab.c++ $(src).tab.h++ lex.yy.c $(objfile
```

Input File and Run

```
3 + 2
3  2 * 5
7 / 2

$ calc < input
5
syntax error
3.5
```

Note

- `%start s` - specifies the start symbol of the grammar.
- `s: s line`
  `|            /* Empty string */ ;` - is equivalent to $s \rightarrow \varepsilon \mid s$ line; both 's' and 'line' are no-terminals.

No actions are associated with these two rules.

Note

```
line:     '\n'
      | exp   '\n' { printf("%f\n", $1); }
      | error '\n' { yyerrok ;}
;
```

A 'line' may be '\n' or an expression (exp) followed by '\n'. The call to printf() is the semantic action taken when exp '\n' is reduced to line.

$1 is the pseudo variable for the attribute value of exp[a], the first symbol of the right-hand side of the rule.

[a] The value of the expression in this case.

Note

- On detecting a syntax error, bison calls the function `yyerror()`.
- The third rule is used for simple error recovery. The parser skips up to the newline character and continues.
- '`error`' is called an error token. It is used to find the synchronization point from where the parsing can continue. In this case it is the newline character.

Note

- `yyerrok` is a macro. It informs the parser (bison) that the error recovery is complete and the parser can start from normal state.
- Bison after reporting an error, removes states and symbols from the parsing stack until it is in a state where it can shift `error` token.

Note

- Then the parser discards all input until it reaches the synchronization input following the `error` token.
- It then enters in recovery state. In this case `yyerrok` brings the parser to normal state.

Note

```
exp:  INT  {  $$ = (float)$1;}
        |   FLOAT /* Default action */
```

The attribute of the token `INT` is available in the pseudo variable '\$1'. It is assigned as the value of the pseudo variable \$\$ corresponding to the left-hand non-terminal. The second rule uses the default action `$$ = $1;`.
Types of pseudo variables are specified in `%type` deceleration.

Note

- The action takes place during the reduction of the handle INT, a terminal, to the non-terminal exp.
- The attribute coming from the scanner is saved as a synthesized attribute of the non-terminal on the value stack.

Note

```
exp:
        | '-' exp %prec UNEG  { $$= -$2;}
```

The `%prec` directive tells the bison compiler that the precedence of the rule is that of `UNEG` that is higher than the binary operators. This differentiates between the unary and binary operators with the same symbol.

Symbol Locations

The location of a token or the range of a string corresponding to a non-terminal in the input stream may be important for several reasons e.g. error detection.
`bison` provides facility to define datatype (`YYLTYPE`) for a location. There is a default type that can be redefined if necessary.

Default YYLTYPE

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

Pseudo Variables: `@$`, `@n`

If the parser reduces $\alpha_1\alpha_2\cdots\alpha_k\cdots\alpha_n$ to A corresponding to the production rule $A \rightarrow \alpha_1\alpha_2\cdots\alpha_k\cdots\alpha_n$, the location of α_k is available in the pseudo variable `@k` and the location of A will be stored in `@$`.
Similar to the default semantic action, there is a default action for location. It is executed on every match and reduction of a rule, and sets `@$` to the beginning of `@1` and the end of `@n`

Default Action on Location

```
exp:
     | exp '+' exp
       {
        @$.first_column = @1.first_column;
        @$.first_line = @1.first_line;
        @$.last_column = @3.last_column;
        @$.last_line = @3.last_line;
        $$ = $1 + $3; // not a default action
       }
```

Global Variable `yylloc`

The scanner should supply the location information of tokens to make it useful to the parser. The global variable `yylloc` of type `YYLTYPE` is used to pass the information. The scanner puts different location values e.g. line number, column number etc. of a token in this variable and returns to the parser.

Example: scanner

```
{NATNUM} {
  yylloc.first_column = yylloc.last_column+1;
  yylval.integer = atoi(yytext) ;
  yylloc.last_column += strlen(yytext);
  return INT ;
       }
```

Example: parser

```
int main()
{
   yylloc.first_line=yylloc.last_line=1;
   yylloc.first_column=yylloc.last_column=0;
   return yyparse() ;
}
```

Example: parser

```
exp: ......
 | exp '/' exp  {
    $$ = $1/$3;
    if($3 == 0)
     fprintf (stderr, "Divide by zero: %d-%d (col)\n",
              @3.first_column, @3.last_column);
                }
```