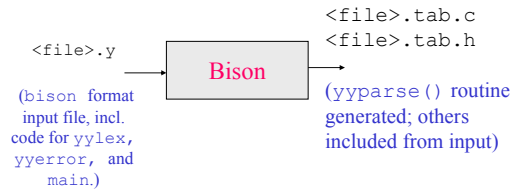


# Bison Parser Generator

Adapted from material by:  
Charles Donnelly and Richard Stallman  
John Levine

## Overview of Bison

- (YACC-compatible) Bottom-up (specifically, LALR(1)) parser generator
- Interfaces with scanner generated by Flex
  - Scanner called as a subroutine when parser needs the next token.



## Bison input file format

- The input file consists of three sections, separated by a line with just `%%` on it:

```

%%
C declarations (types, variables, functions,
preprocessor commands)
%%
Bison declarations (grammar symbols,
operator precedence decl.,
attribute data type)
%%
Grammar rules
%%
Additional C code (incl. scanner yylex)
    
```

## Bison Declarations

- define terminals and nonterminals
- define attributes and their associations with terminals and nonterminals
- specify precedence and associativity

```

%union {
int val;
char *varname;
}
%type <val> exp
%token <varname> NAME
%right =
%left + -
%left * /
    
```

## Rules



- General form of a rule

```
LHS: rule1-RHS... { action 1}
    | rule2-RHS... { action 2}   ...
    ;
```

- LHS is a nonterminal
- rule-RHS is a sequence of nonterminals and terminals.
- An **action** can contain C-code, possibly involving attributes, which is executed when the associated grammar rule is reduced.

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

## Semantic Values and Actions



- Actions can manipulate semantic values associated with a nonterminal.

- ❑  $S_n$  refers to the semantic value (synthesized attribute) of the n-th symbol on the RHS.

- ❑  $$$$  refers to the semantic value of the LHS nonterminal.

- ❑ Typically, an action is of the form:

$$$ = f( \$1, \$2, \dots, \$m)$

- ❑ The types for the semantic values are specified in the declaration section.

## A Simple Bison Example : calc.y



```
... Bison Declarations ...
%%
stmt: NAME '=' expr { printf("%c = %d\n", $1, $3); }
    | expr          { printf("%d\n", $1); }
    ;

expr: expr '+' NUMBER    { $$ = $1 + $3; }
    | expr '-' NUMBER    { $$ = $1 - $3; }
    | NUMBER             { $$ = $1; }
%%
... User code ...
```

## (cont'd)



```
#{
#include <stdio.h>
#}
%union {
int val;
char var;
}
%token <val> NUMBER
%token <var> NAME
%type <val> expr
%%
... Grammar Rules ...
%%
yyerror(char *s) { printf("%s\n", s); }
main() { yyparse(); }
```

## calc.flex



```
#{
#include "calc.tab.h"
extern YYSTYPE yyval;
%}

%%
[0-9]+      {yyval.val = atoi(yytext);
              return NUMBER;}
[ \t]+      ; /* ignore whitespaces */
[a-zA-Z]    {yyval.var = yytext[0];
              return NAME;}

\n          return 0; /* logical EOF */
.           return yytext[0];
%%
```

## Generating Parser



- Create `<EG>.y` and `<EG>.flex` files
- Run bison and flex (in that order)
  - `bison -d <EG>.y`
    - ☐ `<EG>.y` contains `yyerror()` and `main()`
    - ☐ bison generates `<EG>.tab.c` `<EG>.tab.h`
  - `flex <EG>.flex`
    - ☐ `<EG>.flex` includes `<EG>.tab.h`;
    - ☐ flex generates `lex.yy.c`
- Compile generated C files
  - `gcc -o eg <EG>.tab.c lex.yy.c -lfl`
- Execute the application

```
eg
p = 23 - 5 + 4
p = 22
```

## calc.tab.h



```
typedef union {
    int val;
    char var;
} YYSTYPE;

#define NUMBER 257
#define NAME 258

extern YYSTYPE yyval;
```

## Precedence and Associativity



```
. . .
%type <val> stmt
%right '='
%left '-' '+'
%nonassoc UMINUS
%%
stmt: NAME '=' stmt { $$ = $3;
                    printf("%s = %d\n", $1, $3); }
    | expr          {}
    ;

expr:  expr '+' expr  { $$ = $1 + $3; }
    | expr '-' expr   { $$ = $1 - $3; }
    | NUMBER          { $$ = $1; }
    | '-' NUMBER %prec UMINUS { $$ = - $2; }
```

## Sample Run



egAdv

j = k = l = 56

l = 56

k = 56

j = 56

egAdv

p = 1 + 2 - 3 - 4

p = -4

egAdv

q = - 3 - 4

q = - 7

egAdv

q = - - 4

parse error

## A Cool Parser



1. Check for correct syntax
  - Write Bison grammar rules which match the Cool grammar in CoolAid
2. Build an Abstract Syntax tree (AST)
  - Write actions in C/C++ to build the Syntax tree
  - Semantic values for the grammar symbols will be (pointers to) AST nodes
  - AST is output from *parsetest* program in outline form
  - Use C++ classes for the three nodes, provided in Cool support code
3. Perform Error recovery for common cases
  - Use Bison `error` token