Bison: a parser generator

A **parser generator** is a program that takes as input a specification of the syntax of a language in some form, and produces as its output a parse procedure for that language.

Historically, parser generators were called **compiler-compilers**, since traditionally all compilation steps were performed as actions included within the parser.

The modern view is to consider the parser to be just one part of the compilation process, so the term compiler-compiler is out of date.

One widely used parser generator that incorporates the LALR(1) parsing algorithm is called **Yacc** (for yet another compiler-compiler).

The public domain versions of Yacc are commonly called **Bison**.

General layout of a Bison specification

```
} Definitions
%%
} Rules
%%
} Auxiliary Routines
```

We explain the basic layout of a Bison specification by using the following grammar:

$$exp \rightarrow exp\ +\ term \mid exp\ -\ term \mid term$$
$$term \rightarrow term\ *\ factor \mid factor$$
$$factor \rightarrow (\ exp\ ) \mid \textbf{number}$$

A First Bison specification

```
%{
#include <stdio.h>
#include <ctype.h>
%}
%token NUMBER
%%
command : exp { printf("%d\n",$1); }
        ;

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

term : term '*' factor {$$=$1*$3;}
     | factor {$$ = $1;}
     ;

factor : NUMBER  {$$=$1;}
       | '(' exp ')' {$$=$2;}
       ;
```


Bison specification continue

```
%%
main()
{
return yyparse();
}

int yylex()
{
 int c;
 while((c=getchar())==' ');
 if (isdigit(c)){
     ungetc(c,stdin);
     scanf("%d",&yylval);
     return(NUMBER);
 }
 if (c =='\n') return 0;
 return(c);
}

int yyerror(char *s)
{ fprintf(stderr,"%s\n",s);
 return 0;
}
```

Using Bison

Save Bison specification as calculator.y

[abvdm@qed bison]$ bison calculator.y

Now we have an output file of C source code called calculator.tab.c

[abvdm@qed bison]$ gcc calculator.tab.c −o calculator

[abvdm@qed bison]$ ./calculator
12+5
17
[abvdm@wiles bison]$ calculator
12++4
syntax error

Using Bison

We change the rules section of the previous bison specification:

```
command : exp {printf("command−>exp\n");}
        ;

exp : exp '+' term  {printf("exp−>exp+term \n");}
    | exp '−' term  {printf("exp−>exp−term \n");}
    | term  {printf("exp−>term \n");}
    ;

term : term '*' factor {printf("term−>term*factor \n");}
     | factor   {printf("term−>factor \n");}
     ;

factor : NUMBER  {printf("factor−>number \n");}
       | '(' exp ')'  {printf("factor−>(exp) \n");}
       ;
```


Using Bison

[abvdm@qed bison]$ bison calculator.y
[abvdm@qed bison]$ gcc calculator.tab.c −o calculator
[abvdm@qed bison]$ ./calculator
12+7*5
factor−>number
term−>factor
exp−>term
factor−>number
term−>factor
factor−>number
term−>term*factor
exp−>exp+term
command−>exp

Verbose Option

We can analize the grammar in the rules section by using the verbose option.

[abvdm@qed bison]$ bison −v calculator.y

This will give use the file calculator.output

For the verbose option we only have to include a skeletal version of the specification.

```
%token NUMBER
%%
command : exp
        ;
exp : exp '+' term
    | exp '−' term
    | term
    ;
term : term '*' factor
     | factor
     ;
factor : NUMBER
       | '(' exp ')'
       ;
```

This is the listing of calculator.output

Grammar

```
0 $accept: command $end

1 command: exp

2 exp: exp '+' term
3    | exp '−' term
4    | term

5 term: term '*' factor
6     | factor

7 factor: NUMBER
8       | '(' exp ')'
```


Terminals, with rules where they appear

```
$end (0) 0
'(' (40) 8
')' (41) 8
'*' (42) 5
'+' (43) 2
'−' (45) 3
error (256)
NUMBER (258) 7
```

Nonterminals, with rules where they appear

```
$accept (9)
   on left: 0
command (10)
   on left: 1, on right: 0
exp (11)
   on left: 2 3 4, on right: 1 2 3 8
term (12)
   on left: 5 6, on right: 2 3 4 5
factor (13)
   on left: 7 8, on right: 5 6
```



```
state 0

  0 $accept: . command $end
  NUMBER  shift, and go to state 1
  '('     shift, and go to state 2
  command  go to state 3
  exp      go to state 4
  term     go to state 5
  factor   go to state 6


state 1

  7 factor: NUMBER .
  $default  reduce using rule 7 (factor)

state 2

  8 factor: '(' . exp ')'
  NUMBER  shift, and go to state 1

  '('     shift, and go to state 2

  exp     go to state 7
  term    go to state 5
  factor  go to state 6
```



```
state 3

  0 $accept: command . $end

  $end  shift, and go to state 8


state 4

  1 command: exp .
  2 exp: exp . '+' term
  3    | exp . '−' term

  '+'  shift, and go to state 9
  '−'  shift, and go to state 10

  $default  reduce using rule 1 (command)


state 5

  4 exp: term .
  5 term: term . '*' factor

  '*'  shift, and go to state 11

  $default  reduce using rule 4 (exp)
```

Listing of calculator.output

```
state 6

    6 term: factor .

    $default  reduce using rule 6 (term)


state 7

    2 exp: exp . '+' term
    3    | exp . '−' term
    8 factor: '(' exp . ')'

    '+'  shift, and go to state 9
    '−'  shift, and go to state 10
    ')'  shift, and go to state 12


state 8

    0 $accept: command $end .

    $default  accept
```


Listing of calculator.output

```
state 9
    2 exp: exp '+' . term
    NUMBER  shift, and go to state 1
    '('     shift, and go to state 2
    term    go to state 13
    factor  go to state 6

state 10
    3 exp: exp '−' . term
    NUMBER  shift, and go to state 1
    '('     shift, and go to state 2
    term    go to state 14
    factor  go to state 6

state 11
    5 term: term '*' . factor
    NUMBER  shift, and go to state 1
    '('     shift, and go to state 2
    factor  go to state 15

state 12
    8 factor: '(' exp ')' .
    $default  reduce using rule 8 (factor)
```


Listing of calculator.output

```
state 13

    2 exp: exp '+' term .
    5 term: term . '*' factor

    '*'  shift, and go to state 11

    $default  reduce using rule 2 (exp)


state 14

    3 exp: exp '−' term .
    5 term: term . '*' factor

    '*'  shift, and go to state 11

    $default  reduce using rule 3 (exp)


state 15

    5 term: term '*' factor .

    $default  reduce using rule 5 (term)
```


Tracing output using yydebug

Change the definition section of specification on p3 and p4 as follows:

Definition Section:

```
%{
#include <stdio.h>
#include <ctype.h>
#define YYDEBUG 1
%}

%token NUMBER
```

First part of Auxiliary Section:

```
main()
{
yydebug=1;
return yyparse();
}
```

Tracing output using yydebug

[abvdm@qed bison]$ ./calculator
Starting parse
Entering state 0
Reading a token: 12+5
Next token is token NUMBER ()
Shifting token NUMBER, Entering state 1
Reducing stack by rule 7 (line 24), NUMBER -> factor
Stack now 0
Entering state 6
Reducing stack by rule 6 (line 21), factor -> term
Stack now 0
Entering state 5
Reading a token: Next token is token '+' ()
Reducing stack by rule 4 (line 17), term -> exp
Stack now 0
Entering state 4
Next token is token '+' ()
Shifting token '+', Entering state 9

Tracing output using yydebug

Reading a token: Next token is token NUMBER ()
Shifting token NUMBER, Entering state 1
Reducing stack by rule 7 (line 24), NUMBER -> factor
Stack now 0 4 9
Entering state 6
Reducing stack by rule 6 (line 21), factor -> term
Stack now 0 4 9
Entering state 13
Reading a token: Now at end of input.
Reducing stack by rule 2 (line 15), exp '+' term -> exp
Stack now 0
Entering state 4
Now at end of input.
Reducing stack by rule 1 (line 12), exp -> command
17
Stack now 0
Entering state 3
Now at end of input.

Grammar Rules

Tokens:
Bison has two ways of recognizing tokens.
1. Any characters inside single quotes in a grammar rule will be recognized as itself. We have for example '+', '-'
or
2. For example %token NUMBER
in the definition section.

Productions:
For example $exp\ :\ exp\ '+'\ term$
instead of $exp->\ exp\ '+'\ term$

Communication between scanner and parser

- A function `yylex()` that implements the scanner must be supplied. This can be supplied in the auxiliary section of Bison or by using Flex.
- If there is a value associated with the token, it should be assigned to the variable **`yylval`**.

Conflicts

- Conflicts may be *shift-reduce* or *reduce-reduce*.
  - In a *shift-reduce* conflict, the default is to *shift*.
  - In a *reduce-reduce* conflict, the default is to *reduce* using the rule that is listed first in the rules section.

Using Bison with Flex

Bison specification

```
%{
#include <stdio.h>
%}
%token NUMBER
%%
command : exp  { printf("%d\n",$1);};
exp : exp '+' term {$$=$1+$3;}
| exp '−' term {$$=$1−$3;}| term {$$=$1;};
 term : term '*' factor {$$=$1*$3;}| factor {$$ = $1;};
factor : NUMBER  {$$=$1;} | '(' exp ')' {$$=$2;};
%%
main()
{ return yyparse();  }
int yyerror(char *s)
{ fprintf(stderr,"%s\n",s);
  return 0;}
```


Using Bison with Flex - Continue

Flex specification:

```
%{
#include "calculator.tab.h"
%}
```

Get this with bison −d calculator.y

```
%%

[\t ]+ ;
[0−9]+ {yylval=atoi(yytext);return NUMBER;}
. {return yytext[0];}
"\n" {return 0;}
```

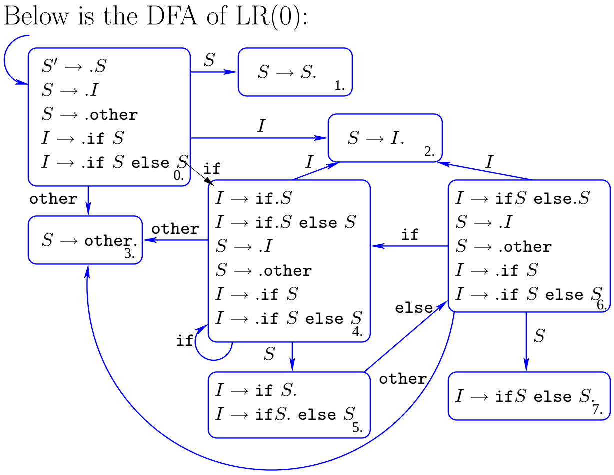
```
[abvdm@qed bison]$ flex scan.lex

[abvdm@qed bison]$ gcc calculator.tab.c lex.yy.c −lfl −o calc

[abvdm@qed bison]$ ./calc
12+34
46
```


Bison: Parsing Conflicts

Consider the DFA of LR(0) items of the following ambiguous grammar:

$$S \rightarrow I \mid \texttt{other}$$
$$I \rightarrow \texttt{if } S \mid \texttt{if } S \texttt{ else } S$$

Below is the DFA of LR(0):



.output listing for example on p3

Bison Specification:

```
%token OTHER IF ELSE
%%
 S:I | OTHER;
 I:IF S | IF S ELSE S;
```

.output file obtained with bison −v

```
State 5 conflicts: 1 shift/reduce
Grammar
0 $accept: S $end
  1 S: I
  2  | OTHER
  3 I: IF S
  4  | IF S ELSE S
```

.output listing for example on p3 - Continue

state 0
0 $accept: . S $end
OTHER shift, and go to state 1
IF shift, and go to state 2
S go to state 3
I go to state 4

state 1
2 S: OTHER .
$default reduce using rule 2 (S)

state 2
3 I: IF . S
4 | IF . S ELSE S
OTHER shift, and go to state 1
IF shift, and go to state 2
S go to state 5
I go to state 4

state 3
0 $accept: S . $end
$end shift, and go to state 6

state 4
1 S: I .
$default reduce using rule 1 (S)

.output listing for example on p3 - Continue

state 5
3 I: IF S .
4 | IF S . ELSE S
ELSE shift, and go to state 7
ELSE [reduce using rule 3 (I)]
$default reduce using rule 3 (I)

state 6
0 $accept: S $end .
$default accept

state 7
4 I: IF S ELSE . S
OTHER shift, and go to state 1
IF shift, and go to state 2
S go to state 8
I go to state 4

state 8
4 I: IF S ELSE S .
$default reduce using rule 4 (I)

.output listing NEW EXAMPLE

Bison specification
%%
S:A | B;
A:’a’;
B:’a’;

.output listing
Rules never reduced
4 B: ’a’
State 1 conflicts: 1 reduce/reduce
Grammar
0 $accept: S $end
1 S: A
2 | B
3 A: ’a’
4 B: ’a’

state 0
0 $accept: . S $end
’a’ shift, and go to state 1
S go to state 2
A go to state 3
B go to state 4

state 1
3 A: ’a’ .
4 B: ’a’ .
$end reduce using rule 3 (A)
$end [reduce using rule 4 (B)]
$default reduce using rule 3 (A)

.output listing for example on p7

state 2
0 $accept: S . $end
$end shift, and go to state 5

state 3
1 S: A .
$default reduce using rule 1 (S)

state 4
2 S: B .
$default reduce using rule 2 (S)

state 5
0 $accept: S $end .
$default accept

Changing type of the value stack

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

%token NUMBER
%union {double val;
      char op;}
%type <val> exp term factor NUMBER
%type <op> addop mulop

%%

command : exp { printf("%lf\n",$1); };


exp : exp addop term {switch($2){
                 case '+':$$=$1+$3; break;
                 case '−':$$=$1+$3; break;
                  }
                 }
     | term {$$=$1;};
```


Changing type of the value stack - Continue

```
term : term mulop factor {switch($2){
                         case '*':$$=$1*$3; break;
                         case '/':$$=$1/$3; break;
                                }
                         }

   | factor {$$=$1;};

addop: '+' {$$='+';}| '−' {$$='−';};
mulop: '*' {$$='*';}| '/' {$$='/';};

factor : NUMBER  {$$=$1;}
   | '(' exp ')' {$$=$2;};

%%
main()
{  return yyparse(); }
int yylex()
{  int c;
   while((c=getchar())==' ');
   if (isdigit(c)){
      ungetc(c,stdin);
      scanf("%lf",&yylval);
      return(NUMBER);
      if (c =='\n') return 0;
   }
   return(c);
}
int yyerror(char *s)
{ fprintf(stderr,"%s\n",s);
  return 0;
}
```


Specifying Precedence and Associativity

As Before

```
%token NUMBER
%left '+' '−'
%left '*'
%%
command : exp { printf("%d\n",$1); };
exp : NUMBER {$$=$1;}
   | exp '+' exp {$$=$1+$3;}
   | exp '−' exp {$$=$1−$3;}
   | exp '*' exp {$$=$1*$3;}
   | '(' exp ')' {$$=$2;};
```

As Before

Error Recovery in Bison

Bison uses **error** productions (that is, rules of the form $factor \rightarrow error$) as error recovery mechanism.

When the parser detects an error during a parse, it pops states from the parsing stack until it reaches a state in which the **error token** is a legal lookahead.

If there are no error productions, then **error** is never a legal lookahead token and the parsing stack will be emptied.

Once the parser has found a state on the stack in which **error** is a legal lookahead, the effect is as though **error** were seen just before the original lookahead.

Error Recovery Example

Change the rule $factor \rightarrow NUMBER$ in our calculator example (as on p1) as follows:

```
factor : NUMBER   {$$ =$1;}
       | '(' exp ')'     {$$=$2;}
       |  error  {$$=0;}
       ;
```

Now we use the debug option in conjunction with the calculator.output file to understand the effect of the **error** transition.

Error Recovery Example - debug slide 1

```
[abvdm@qed bison]$ ./calculator
Starting parse
Entering state 0
Reading a token: 2++3
Next token is token NUMBER ()
Shifting token NUMBER, Entering state 2
Reducing stack by rule 7 (line 24), NUMBER -> factor
Stack now 0
Entering state 7
Reducing stack by rule 6 (line 21), factor -> term
Stack now 0
Entering state 6
Reading a token: Next token is token '+' ()
Reducing stack by rule 4 (line 17), term -> exp
Stack now 0
Entering state 5
Next token is token '+' ()
Shifting token '+', Entering state 10
Reading a token: Next token is token '+' ()
syntax error
Shifting error token, Entering state 1
Reducing stack by rule 9 (line 26), error -> factor
Stack now 0 5 10
Entering state 7
Reducing stack by rule 6 (line 21), factor -> term
```


Error Recovery Example - debug slide 2

```
Stack now 0 5 10
Entering state 14
Next token is token '+' ()
Reducing stack by rule 2 (line 15), exp '+' term -> exp
Stack now 0
Entering state 5
Next token is token '+' ()
Shifting token '+', Entering state 10
Reading a token: Next token is token NUMBER ()
Shifting token NUMBER, Entering state 2
Reducing stack by rule 7 (line 24), NUMBER -> factor
Stack now 0 5 10
Entering state 7
Reducing stack by rule 6 (line 21), factor -> term
Stack now 0 5 10
Entering state 14
Reading a token: Now at end of input.
Reducing stack by rule 2 (line 15), exp '+' term -> exp
Stack now 0
Entering state 5
Now at end of input.
Reducing stack by rule 1 (line 12), exp -> command
5
Stack now 0
Entering state 4
Now at end of input.
```


Error Recovery Example - calculator.output

```
 0 $accept: command $end
 1 command: exp
 2 exp: exp '+' term
 3    | exp '-' term
 4    | term
 5 term: term '*' factor
 6     | factor
 7 factor: NUMBER
 8       | '(' exp ')'
 9       | error

 state 0

    0 $accept: . command $end
    error   shift, and go to state 1
    NUMBER  shift, and go to state 2
    '('     shift, and go to state 3
    command  go to state 4
    exp      go to state 5
    term     go to state 6
    factor   go to state 7

 state 1

    9 factor: error .
    $default  reduce using rule 9 (factor)

 state 2

    7 factor: NUMBER .
    $default  reduce using rule 7 (factor)
```

Error Recovery Example - calculator.output - Continue

state 3
8 factor: '(' . exp ')'
error shift, and go to state 1
NUMBER shift, and go to state 2
'(' shift, and go to state 3
exp go to state 8
term go to state 6
factor go to state 7

state 4
0 $accept: command . $end
$end shift, and go to state 9

state 5
1 command: exp .
2 exp: exp . '+' term
3 | exp . '−' term
'+' shift, and go to state 10
'−' shift, and go to state 11
$default reduce using rule 1 (command)

state 6
4 exp: term .
5 term: term . '*' factor
'*' shift, and go to state 12
$default reduce using rule 4 (exp)

state 7
6 term: factor .
$default reduce using rule 6 (term)

Error Recovery Example - calculator.output - Continue

state 8
2 exp: exp . '+' term
3 | exp . '−' term
8 factor: '(' exp . ')'
'+' shift, and go to state 10
'−' shift, and go to state 11
')' shift, and go to state 13

state 9
0 $accept: command $end .
$default accept

state 10
2 exp: exp '+' . term
error shift, and go to state 1
NUMBER shift, and go to state 2
'(' shift, and go to state 3
term go to state 14
factor go to state 7

state 11
3 exp: exp '−' . term
error shift, and go to state 1
NUMBER shift, and go to state 2
'(' shift, and go to state 3
term go to state 15
factor go to state 7

state 12
5 term: term '*' . factor
error shift, and go to state 1
NUMBER shift, and go to state 2
'(' shift, and go to state 3
factor go to state 16

Error Recovery Example - calculator.output - Continue

state 13
8 factor: '(' exp ')' .
$default reduce using rule 8 (factor)

state 14
2 exp: exp '+' term .
5 term: term . '*' factor
'*' shift, and go to state 12
$default reduce using rule 2 (exp)

state 15
3 exp: exp '−' term .
5 term: term . '*' factor
'*' shift, and go to state 12
$default reduce using rule 3 (exp)

state 16
5 term: term '*' factor .
$default reduce using rule 5 (term)

Error Recovery Example - Parse Tree