Details of Yacc

AN EXAMPLE

As a first exercise, we shall develop a parser for the following expression language:

$$
\begin{array}{llll}
exp & \rightarrow & & num \\
 & & | & exp + exp \\
 & & | & exp * exp \\
 & & | & - \ exp \\
 & & | & (\ exp\ )
\end{array}
$$

```
/*C declarations*/
%{
%}

/* YACC Declarations */
%union
{
    int tokenval;
}

%token NUM
%left '-' '+'
%left NEG     /* negation--unary minus */
```

```
/* Grammar follows */
%%

exp:      NUM                  {printf("rule 1\n");}
        | exp '+' exp          {printf("rule 2\n");}
        | exp '-' exp          {printf("rule 3\n");}
        | '-' exp  %prec NEG   {printf("rule 4\n");}
        | '(' exp ')'          {printf("rule 5\n");}
        ;
%%
yyerror(s)
  char* s;
 {
    printf("%s\n", s);
 }


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

Assume that this yacc script is in the file example1.y. If the lex script for the language is in example1.l, then the parser for the language is generated by the following:

```
# lex example1.l
# yacc -d example1.y
# cc -o example1 lex.yy.c y.tab.c -ll
```

Now suppose that the file `junk` contains:

```
23.4+6.2--2.5+(1.0+2)
```

then the command

```
# example1 < junk
```

will produce:

```
rule 1
rule 1
rule 2
rule 1
rule 4
rule 3
rule 1
rule 1
rule 2
rule 5
rule 2
```

It is instructive to trace the parse tree for this example.

Some observations:

- **Just like a lex script, a yacc script has three parts:**

  . . . definition section. . .
  %%
  . . . rules section. . .
  %%
  . . . user defined functions. . .

- **The *definition section* is divided into two parts, *C definitions* and *yacc definitions*. The C definitions are placed verbatim in the early part of the generated parser.**

- **A *rule* is a *production* – *action* pair. A production action pair is of the form:**

  ```
  A : B C D C-statement;
  E : G H    C-statement;
  I : J      C-statement;
  ```

  Or if the the left hand side of the production are the same, then

  ```
  A : B C D C-statement
    | G H     C-statement
    | J       C-statement
    ;
  ```

Large YACC DEFINITION SECTION
The yacc definition section aloows token definition, precedence and associativity definition, start nonterminal definition, and type declaration of token attributes.

- **The line**

  ```
  %token NUM
  ```

  announces the fact `NUM` **is a token. This helps in communication with the lexical analyser. The lines**

  ```
  %left '-' '+'
  %left NEG
  ```

  also declare `NEG` **to be a token. Note that single character tokens do not have to be declared to be tokens, However, the above lines also convey more information.**

- **A single character**

- **These declarations say that** `'-'` **and** `'+'` **are left associative and have the same precedence. Also that** `NEG` **has a higher precedence than either of them.**

- **The token** `NEG` **is a hypothetical token. Its purpose is to give unary** `'-'` **a higher precedence than either** `'-'` **or** `'+'` **through the**

  ```
  exp:   '-' exp  %prec NEG {printf("rule 4\n");}
  ```

  grammar rule.

YACC DISAMBIGUATING RULES

In absence of any associativity and precedence information:

1. In case of a shift-reduce conflict, shift.
2. In case of a reduce-reduce conflict, reduce with the earlier rule.

From the precedence and associativity information of tokens, associate precedence and associativity to grammar rules; it is the precedence and associativity of the last token in the body of the grammar rule.
In case of a shift-reduce conflict, compare the precedence of the symbol to be shifted with the precedence of the rule to be used for reduction.

3. If precedence of symbol is higher then shift, else reduce,

if the precedences of both are the same, then:

4. If the symbol (and the grammar rule) is left associative then reduce, else shift.

ILLUSTRATING DISAMBIGUATION

We shall illustrate the disambiguating rules by using the debugging facilities of yacc. To do this, change the main program to set a variable `yydebug` to a non-zero value. Also run yacc using the `-t` switch.

EXAMPLE 1

For the yacc grammar:

```
%start A
%%
D:       'b' 'c';

A:       'a' D
          | 'a' 'b' 'c'
          ;
%%
```

the output for the string `abc` is:

```
Starting parse
Entering state 0
Reading a token: Next token is 97 ('a')
Shifting token 97 ('a'), Entering state 1
Reading a token: Next token is 98 ('b')
Shifting token 98 ('b'), Entering state 2
Reading a token: Next token is 99 ('c')
Shifting token 99 ('c'), Entering state 4
Reducing via rule 1 (line 6), 'b' 'c'  -> D
state stack now 0 1
Entering state 3
Reducing via rule 2 (line 8), 'a' D  -> A
```

```
state stack now 0
Entering state 5
Reading a token: Now at end of input.
Shifting token 0 ($), Entering state 6
Now at end of input.
```

EXAMPLE 2

Consider an example in which no associativity and precedence information is given.

```
%token NUM

%%
exp:      NUM
         | exp '+' exp
         | exp '-' exp
         | '-' exp
         | '(' exp ')'
         ;
%%
```

For the input `23.4+6.2-2.5`**, the output is:**

```
Starting parse
Entering state 0
Reading a token: Next token is 258 (NUM)
Shifting token 258 (NUM), Entering state 1
Reducing via rule 1 (line 11), NUM  -> exp
reducing by rule 1
state stack now 0
```

```
Entering state 4
Reading a token: Next token is 43 ('+')
Shifting token 43 ('+'), Entering state 8
Reading a token: Next token is 258 (NUM)
Shifting token 258 (NUM), Entering state 1
Reducing via rule 1 (line 11), NUM  -> exp
reducing by rule 1
state stack now 0 4 8
Entering state 11
Reading a token: Next token is 45 ('-')
Shifting token 45 ('-'), Entering state 7
Reading a token: Next token is 258 (NUM)
Shifting token 258 (NUM), Entering state 1
Reducing via rule 1 (line 11), NUM  -> exp
reducing by rule 1
state stack now 0 4 8 11 7
Entering state 10
Reading a token: Now at end of input.
Reducing via rule 3 (line 13), exp '-' exp  -> exp
reducing by rule 3
state stack now 0 4 8
Entering state 11
Now at end of input.
Reducing via rule 2 (line 12), exp '+' exp  -> exp
reducing by rule 2
state stack now 0
Entering state 4
Now at end of input.
Shifting token 0 ($), Entering state 12
Now at end of input.
```

EXAMPLE 3

For our original example, for the same input, the output will be:

```
Starting parse
Entering state 0
Reading a token: Next token is 258 (NUM)
Shifting token 258 (NUM), Entering state 1
Reducing via rule 1 (line 14), NUM  -> exp
reducing by rule 1
state stack now 0
Entering state 4
Reading a token: Next token is 43 ('+')
Shifting token 43 ('+'), Entering state 8
Reading a token: Next token is 258 (NUM)
Shifting token 258 (NUM), Entering state 1
Reducing via rule 1 (line 14), NUM  -> exp
reducing by rule 1
state stack now 0 4 8
Entering state 11
Reducing via rule 2 (line 15), exp '+' exp  -> exp
reducing by rule 2
state stack now 0
Entering state 4
Reading a token: Next token is 45 ('-')
Shifting token 45 ('-'), Entering state 7
Reading a token: Next token is 258 (NUM)
Shifting token 258 (NUM), Entering state 1
Reducing via rule 1 (line 14), NUM  -> exp
reducing by rule 1
```

```
state stack now 0 4 7
Entering state 10
Reducing via rule 3 (line 16), exp '-' exp  -> exp
reducing by rule 3
state stack now 0
Entering state 4
Reading a token: Now at end of input.
Shifting token 0 ($), Entering state 12
Now at end of input.
```

INTERACTION

How does the parser interact with the parser. One way is to physically insert the `yylex()` function in the generated parser. Then the declarations in the generated parser are available to the lexer.
A better way is to communicate through the yacc generated header file `y.tab.h`, using the switch `-d`. Here is `y.tab.h` for a variation of the first example. Only the declaration part is shown: example.

```
/* YACC Declarations */

%union
{
    double tokenval;
}


%token <tokenval> NUM
%type <tokenval> exp

%left  '-' '+'
%left '*' '/'
%left NEG     /* negation--unary minus */

/* Grammar follows */
%%

exp:        NUM
          | exp '+' exp
          | exp '-' exp
```

```
        ;

%%
```

The generated `y.tab.h` file is:

```
typedef union
{
    int double;
} YYSTYPE;
#define NUM     258
#define NEG     259


extern YYSTYPE yylval;
```

- **For single character tokens, each character has a ordinal number which is agreed to between the lexer and the parser. So no communication is necessary**
- **For other tokens, yacc generates a number starting from 256. This information goes in the `y.tab.h` file.**
- **yacc also defines the type `YYSTYPE` based on the `%union` declaration and declares the variable `yylval` to be of this type. `yylval` is variable used for passing token attributes, also called semantic values.**

This file `y.tab.h` is included in the lexer. The lexer for this example looks like:

```
%{
#include "example1.tab.h"
```

```
%}

ws [ \t\n]+
letter [A-Za-z]
digit  [0-9]
number {digit}+(\.{digit}+)?(E[+\-]?{digit}+)?

%%

{ws}     {/*no action, no return*/}
{number} {yylval.tokenval=atof(yytext); return(NUM);}
"+"      {return('+');}
"-"      {return('-');}
"("      {return('(');}
")"      {return(')');}

%%
```