next up previous
Next: YACC: Further usage **Up:** CS2121: The Implementation and **Previous:** LEX

Subsections

YACC

YACC

YACC program for an infix calculator

We will keep the same operators as the postfix calculator, but we will give them their usual associativity (left), precedence (* and / before + and -) and we will also need brackets.

(`$CS2121/e*/infix1/*`)

```
%{ /* C declarations used in actions */
#include <stdio.h>
%}

/* yacc definitions */

%union {int a_number;}
%start line
%type <a_number> exp term factor number digit

%%
/*descriptions of expected inputs       corresponding actions (in C)*/

line    : exp ';' '\n'                  {printf ("result is %d\n", $1);}
        ;
exp     : term                          {$$ = $1;}
        | exp '+' term                  {$$ = $1 + $3;}
        | exp '-' term                  {$$ = $1 - $3;}
        ;
term    : factor                        {$$ = $1;}
        | term '*' factor               {$$ = $1 * $3;}
```

```
       | term '/' factor              {$$ = $1 / $3;}
       ;
factor : number                       {$$ = $1;}
       | '(' exp ')'                  {$$ = $2;}
       ;
number : digit                        {$$ = $1;}
       | number digit                 {$$ = $1*10 + $2;}
       ;
digit  : '0'                          {$$ = 0;}
       | '1'                          {$$ = 1;}
       | '2'                          {$$ = 2;}
       | '3'                          {$$ = 3;}
       | '4'                          {$$ = 4;}
       | '5'                          {$$ = 5;}
       | '6'                          {$$ = 6;}
       | '7'                          {$$ = 7;}
       | '8'                          {$$ = 8;}
       | '9'                          {$$ = 9;}
       ;

%%                       /* C code */

int main (void) {return yyparse ( );}

int yylex (void) {return getchar ( );}

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

Format of the grammar rules for YACC

```
name    : names and 'single character's
        | alternatives
        ;
```

YACC definitions

`%start line`	means the whole input should match `line`
`%union`	lists all possible types for values associated with parts of the grammar and gives each a field-name
`%type`	gives an individual type for the values associated with each part of the grammar,
	using the field-names from the %union declaration

Actions, C declarations and code

`$$`	resulting value for any part of the grammar
`$1`, `$2`, etc.	values from sub-parts of the grammar
`yyparse`	routine created by YACC from (expected input, action) lists.
	(It actually returns a value indicating if it failed to recognise the input.)
`yylex`	routine called by yyparse for all its input.
	We are using getchar, which just reads characters from the input.
`yyerror`	routine called by yyparse whenever it detects an error in its input.

How YACC is used

BYACC : calc.y $\rightarrow$ calc.c
GCC : calc.c $\rightarrow$ calc
calc : expression $\rightarrow$ result

YACC has other facilities, some of which we will use elsewhere, but those described above are among the most important. Further details and examples can be found in the readings (§3.10).

Using LEX and YACC together

Unfortunately, YACC cannot represent numbers as `[0-9]+` nor easily obtain the corresponding value, nor can it easily be used to ignore white space and comments. Therefore, we need to use both LEX and YACC together; LEX for the simple parts (e.g. numbers, white space, comments) and YACC for more complex parts (e.g. expressions).

YACC code for infix calculator using LEX and YACC (`$CS2121/e*/infix2/*`):

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

%union {int a_number;}
%start line
%token <a_number> number
%type <a_number> exp term factor

%%

line   : exp ';'          {printf ("result is %d\n", $1);}
       ;
exp    : term             {$$ = $1;}
       | exp '+' term     {$$ = $1 + $3;}
       | exp '-' term     {$$ = $1 - $3;}
       ;
term   : factor           {$$ = $1;}
       | term '*' factor  {$$ = $1 * $3;}
       | term '/' factor  {$$ = $1 / $3;}
       ;
factor : number           {$$ = $1;}
       | '(' exp ')'      {$$ = $2;}
       ;
%%

int main (void) {return yyparse ( );}

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

LEX code for infix calculator using LEX and YACC (`$CS2121/e*/infix2/*`):

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

%%

[0-9]+                {yylval.a_number = atoi(yytext); return number;}
```

```
[ \t\n]                 ;
[-+*/();]               {return yytext[0];}
.                       {ECHO; yyerror ("unexpected character");}

%%

int yywrap (void) {return 1;}
```

YACC declarations

`%token`	declare each grammar rule used by YACC that is recognised by LEX and give type of value

LEX declarations and actions

`y.tab.h`	gives LEX the names and type declarations etc. from YACC
`yylval`	name used for values set in LEX e.g.

```
yylval.a_number = atoi (yytext);

yylval.a_name = findname (yytext);
```

How LEX and YACC are used together

BYACC : calcy.y $\rightarrow$ calcy.c + y.tab.h
GCC : calcy.c $\rightarrow$ calcy.o
FLEX : calcl.l + y.tab.h $\rightarrow$ calcl.c
GCC : calcl.c $\rightarrow$ calcl.o
GCC : calcl.o + calcy.o $\rightarrow$ calc
calc : expression $\rightarrow$ result

Epilogue

Using a tool like YACC, infix, postfix and prefix expressions are equally simple to implement - it automatically checks that we have the correct number and layout of operands. We will see in the next section that YACC can also cope with precedence and associativity.

We now have two different ways of describing patterns in text - regular expressions and BNF - and two different tools to deal with them - LEX and YACC. Why don't we just use the better of these two and forget the other one? Any pattern we can describe using regular expressions can also be described using BNF, but not vice-versa, so in this sense BNF is the more powerful of the two notations. However, that power has a price, both in terms of how hard it can be to write the BNF (e.g. recognising numbers), and in terms of how poorly YACC performs compared with LEX.

As we have seen in the example programs, both LEX and YACC can be used independantly of each other. In fact, on many occasions I have written small text-processing programs just using LEX. However, if I was going to deal with a real programming language I would always use LEX and YACC together, partly for performance reasons, but mainly because it makes writing the patterns (and their actions) much simpler if it can be divided into these two parts.

This notational convenience is particularly obvious in two simple and common situations: recognising sets of characters using e.g. `[0-9]` or `[A-Z]`, and recognising spaces and comments which then must be discarded (e.g. a bonus in the third lab exercise).

Exercises

(† indicates harder problems)

- Why do we quote e.g. `'='` `'+'` `'*'` etc. in the descriptions of expected inputs for YACC, when we don't have to for LEX?
- Although we can use single characters like `'+'` and `'*'` in YACC, we can not use multi-character strings like `'mod'`. Extend the calculator in §3.6 to include multi-character operators like `mod` and `div`. 5
- Extend the calculator in §3.6 to include some mathematical functions e.g. `sqrt ( expression )`.
- Design a grammar to recognise a string of the form `AA...ABB...B`, i.e. any number of `A`s followed by any number of `B`s. Would it be better to use LEX or YACC to recognise it? Change your grammar to recognise strings with equal numbers of `A`s and `B`s - now which is best?
- † What if we wanted equal numbers of `A`s, `B`s and `C`s?

Readings

Louden: chs. 4.2, 4.6
Johnson
Levine, Mason & Brown: chs. 1, 3, 7
Capon & Jinks: chs. 8.1, 8.4, 8.5
Aho, Sethi & Ullman: chs. 4.9, 2.1-2.5

---

ne: u previoı
Next: YACC: Further usage **Up:** CS2121: The Implementation and **Previous:** LEX
Pete Jinks
2004-10-26