Compiler Construction Labsheet-8

YACC

Return values

In the last lab, we saw an example a very sketchy example where Lex only returned the information that there was a number, not the actual number. For this, we need a further mechanism. In addition to specifying the return code, the lex parser can return a value put on top of the stack, so that yacc can access it. This symbol is returned in the variable yylval. This is defined as an int by default, so the lex program would have

```
extern int yylval;
%%
[0-9]+ {yylval=atoi(yytext); return NUMBER;}
```

In addition to specifying the return code, the lex parser can return a value that is put on top of the stack, so that yacc can access it. This symbol is returned in the variable yylval. By default, this is defined as an int, so the lex program would have-

```
extern int yylval;
%%
[0-9]+ {yylval=atoi(yytext); return NUMBER;}
```

Stack values

The example 1 had such rules as:

```
expr:
expr '+' mulex { $$ = $1 + $3; }
| expr '-' mulex { $$ = $1 - $3; }
| mulex { $$ = $1; }
```

The action belonging to the different righthand sides refer to $n quantities and to $$. The latter refers to the stack top, so by assigning to it a new item is put on the stack top. The former variables are assigned the values on the top of the stack: if the right hand side has three terms, terminal or nonterminal, then $1 through $3 are assigned and the three values are removed from the stack top.

The lex rule also sets a variable yylval; this puts a value on the stack top, where yacc can find it with $1, $2, et cetera. All of this will be explained in detail below.

Example:1
// sample program to print the integer value of NUM token returned by lexer in yacc code using yylval

Example 2: Write a yacc and lex program to perform the addition of two numbers.

More than one return type

If more than just integers need to be returned, the specifications in the yacc code become more complicated. Suppose we are writing a calculator with variables, so we want to return double values, and integer indices in a table. The following three actions are needed.
1. The possible return values need to be stated:
%union {int ival; double dval;}
2. These types need to be connected to the possible return tokens:
%token <ival> INDEX
%token <dval> NUMBER
3. The types of non-terminals need to be given:
%type <dval> expr
%type <dval> mulex
%type <dval> term

The generated .h file will now have
#define INDEX 258
#define NUMBER 259

typedef union {int ival; double dval;} YYSTYPE;
extern YYSTYPE yylval;

Example: 3 yacc program that computes sum of two integers or two double values and but prints as double value

Example 4: Write Yacc and Lex program that computes sum of-
i) INT + DOUBLE
ii) DOUBLE + DOUBLE
iii) INT+ DOUBLE
iv) DOUBLE+INT
But independent of INT or FLOAT operands, the program prints the final value as DOUBLE value.

Example 5: Simple calculator program

This calculator evaluates simple arithmetic expressions. The lex program matches numbers. And operators and returns them; it ignores white space, returns newlines, and gives an error message or anything else.

Use the following grammar:

expr: expr + mulex | expr - mulex | mulex

mulex: mulex * term | mulex / term | term

term: (expr) | INTEGER

Operators: precedence and associativity

The above example had separate rules for addition/subtraction and multiplication/division. We could simplify the grammar by writing.

```
expr: expr '+' expr ;
      | expr '-' expr ;
      | expr '*' expr ;
      | expr '/' expr ;
      | expr '^' expr ;
      | number ;
```

But this would have 1+2*3 evaluated to 9. To indicate operator precedence, we can have lines

```
%left '+' '-'
%left '*' '/'
%right '^'
```

The sequence of lines indicates increasing operator precedence, and the keyword sets the associativity type: we want 5-1-2 to be 2, so minus is left associative; we want 2^2^3 to be 256, not 64, so exponentiation is right-associative.

Example: rewrite the simple calculator program using the grammar

```
expr: expr '+' expr ;
      | expr '-' expr ;
      | expr '*' expr ;
      | expr '/' expr ;
      | expr '^' expr ;
      | number ;
```

Homework: Write a parser program that checks for the equality of integer/string

Sample Input:
12=12
Output : They are Equal
Sample Input:
OK=OK
Output : They are Equal
Sample Input:
12=10
Output : They are Not Equal
Sample Input:
12=ok
Output : Syntax error
Sample Input:
ok=13
Output : Syntax error