# Writing an Interpreter with Lex, Yacc, and Memphis

Here is a small example that shows how to write an interpreter with Lex, Yacc, and Memphis.

Our example language provides arithmetic and relational expressions as well as assignment and print statements. To structure programs it features conditional and repetitive statements and the possibility to group statements to sequences.

Here is a typical program in our example language:

```
// Greatest Common Divisor
x := 8;
y := 12;
WHILE x != y DO
   IF x > y THEN x := x-y
   ELSE y := y-x
   FI
OD;
PRINT x
```

Our processor for this language will be decomposed into two parts.

The task of the first part (the **analizer**) is to read the source program and to discover its structure.

The task of the second part (the **tree walker**) is to process this structure, thereby evaluating expressions and executing statements.

The **glue** between these parts is an abstract program representation.

## The Analizer

The task to structure the program is decomposed into lexical analysis and syntactical analysis.

**Lexical analysis** splits the source text into a sequence of tokens, skipping blanks, newlines, and comments. For example, the source text

```
x :=   // multiply x
x*100  // by hundred
```

is handled as the sequence of tokens

```
x
:=
x
*
100
```

Each token belongs to a token class. There are simple tokens such as ``:=", it belongs to the class `ASSIGN` which has only this member. And there are more complex tokens such `100`, it belongs to the class `Number` which comprises the strings that form decimal numbers. Simple tokens can be specified simply by the string that represents them. Complex tokens are defined by a *regular expression* that covers the strings of the token class. For example, the regular expression

```
[0-9]+
```

specifies nonempty sequences of decimal digits. In case of simple tokens we just need to know the token class, in case of complex tokens some additional processing is neccessary. E.g. the strings that matches the regular expression for numbers must be converted to an integer that holds its numerical value.

The lexical analysis is implemented by a function `yylex()` that reads a token from the input stream and returns its name (token class). In addition, it assign the semantic value (e.g. of numbers) to the global variable `yylval`.

Such a function can be generated by the tool Lex. Its input is a set of pairs

```
regular-expression { action }
```

The action is performed when the current input matches the regular expression. For example,

```
":=" { return ASSIGN; }
```

defines `ASSIGN` tokens and

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

specifies how to handle numbers.

Here is the input to Lex:

```
%{
#include "y.tab.h"
extern int yylval;
%}
%%
"="      { return EQ; }
"!="     { return NE; }
"<"      { return LT; }
"<="     { return LE; }
">"      { return GT; }
">="     { return GE; }
"+"      { return PLUS; }
"-"      { return MINUS; }
"*"      { return MULT; }
"/"      { return DIVIDE; }
")"      { return RPAREN; }
"("      { return LPAREN; }
":="     { return ASSIGN; }
";"      { return SEMICOLON; }
"IF"     { return IF; }
"THEN"   { return THEN; }
"ELSE"   { return ELSE; }
"FI"     { return FI; }
"WHILE"  { return WHILE; }
"DO"     { return DO; }
"OD"     { return OD; }
"PRINT"  { return PRINT; }
[0-9]+   { yylval = atoi(yytext); return NUMBER; }
[a-z]    { yylval = yytext[0] - 'a'; return NAME; }
\        { ; }
\n       { nextline(); }
\t       { ; }
"//".*\n { nextline(); }
.        { yyerror("illegal token"); }
%%
#ifndef yywrap
yywrap() { return 1; }
#endif
```

**Syntactical analysis** imposes a hierarchical structure on the program. This structure is specified by the rules of a *context-free grammar*. A syntactical phrase is introduced by giving one or more alternatives. An alternative specifies how to construct an instance of the phrase. It list the members that build up the phrase, where such a member is either atoken or the name of a phrase (a *nonterminal*).

Consider the rule to define `statements`:

```
statement:
  designator ASSIGN expression
| PRINT expression
| IF expression THEN stmtseq ELSE stmtseq FI
| IF expression THEN stmtseq FI
| WHILE expression DO stmtseq OD
;
```

For example, the first alternative specifies that if *D* is a `designator` and if *E* is an `expression` then *D* := *E* is a `statement`.

We use the tool Yacc to generate the syntactical analizer. Its input is a context-free grammar from which it creates a function `yyparse()` that parses the source text according to that grammar. (`yyparse()` invokes `yylex()` to obtain the next token).

With rules like the one given above, `yyparse()` would only be able to check whether a given source is consistent with the grammar. As we did with the Lex specification, we attach semantic actions. They are executed whenever an alternative matches a phrase of the input and are used to construct an abstract program representation.

The rule for `statement` becomes:

```
statement:
  designator ASSIGN expression {$$ = assignment($1, $3);}
| PRINT expression {$$ = print($2);}
| IF expression THEN stmtseq ELSE stmtseq FI
    {$$ = ifstmt($2, $4, $6);}
| IF expression THEN stmtseq FI
    {$$ = ifstmt($2, $4, empty());}
| WHILE expression DO stmtseq OD {$$ = whilestmt($2, $4);}
;
```

Consider again the first alternative. The semantic action attached to it constructs an abstract representation of an assignment statement and defines this as the structural value of the phrase, i.e. it assigns it to the special variable `$$`. the value is constructed by applying the function `assignment()` to the value of the first member (`designator`), denoted by `$1`, and the value of the third member (`expression`), denoted by `$3`.

Here is the input to Yacc:

```
%start ROOT

%token EQ
%token NE
%token LT
%token LE
%token GT
%token GE
%token PLUS
%token MINUS
%token MULT
%token DIVIDE
%token RPAREN
%token LPAREN
%token ASSIGN
%token SEMICOLON
%token IF
%token THEN
%token ELSE
%token FI
%token WHILE
%token DO
%token OD
%token PRINT
%token NUMBER
```

```
%token NAME

%%

ROOT:
  stmtseq { execute($1); }
;

statement:
  designator ASSIGN expression { $$ = assignment($1, $3); }
| PRINT expression { $$ = print($2); }
| IF expression THEN stmtseq ELSE stmtseq FI
     { $$ = ifstmt($2, $4, $6); }
| IF expression THEN stmtseq FI
     { $$ = ifstmt($2, $4, empty()); }
| WHILE expression DO stmtseq OD { $$ = whilestmt($2, $4); }
;

stmtseq:
  stmtseq SEMICOLON statement { $$ = seq($1, $3); }
| statement { $$ = $1; }
;

expression:
  expr2 { $$ = $1; }
| expr2 EQ expr2 { $$ = eq($1, $3); }
| expr2 NE expr2 { $$ = ne($1, $3); }
| expr2 LT expr2 { $$ = le($1, $3); }
| expr2 LE expr2 { $$ = le($1, $3); }
| expr2 GT expr2 { $$ = gt($1, $3); }
| expr2 GE expr2 { $$ = gt($1, $3); }
;

expr2:
  expr3 { $$ == $1; }
| expr2 PLUS expr3 { $$ = plus($1, $3); }
| expr2 MINUS expr3 { $$ = minus($1, $3); }
;

expr3:
  expr4 { $$ = $1; }
| expr3 MULT expr4 { $$ = mult($1, $3); }
| expr3 DIVIDE expr4 { $$ = divide ($1, $3); }
;

expr4:
  PLUS expr4 { $$ = $2; }
| MINUS expr4 { $$ = neg($2); }
| LPAREN expression RPAREN { $$ = $2; }
| NUMBER { $$ = number($1); }
| designator { $$ = $1; }
;

designator:
  NAME { $$ = name($1); }
;
```

## The Glue

As we have seen with `assignment()`, the abstract representation, or *abstract syntax*, is constructed by functions that take the representation of constituents and build the representation of a larger construct.

This results in a tree structure: the functions construct nodes whose childs are subtrees representing the constituents.

In language processors the abstract syntax plays a central role. It does not only define the glue between passes, it also determines the design of functions that process the program: they often inductively follow the structure of the abstract representation.

Hence it is a good idea to provide a clean definition. We classify the nodes into into node types and list the types of its childs.

For our example language we introduce two node types: `Statement` and `Expression`. An example of nodes of type `Statement` is `assignment` that takes two arguments (`lhs` and `rhs`) of type `Expression`. This is specified by listing

```
assignment (Expression lhs, Expression rhs)
```

as an alternative of type `Statement`.

We use domain declarations for the specification.

For example, `Statement` is introduced by a declaration of the form

```
domain Statement {
   ...
}
```

that lists the `Statement` alternatives. One of them is

```
assignment (Expression lhs, Expression rhs)
```

Here is the complete definition of the abstract syntax:

```
domain Statement {

   assignment (Expression lhs, Expression rhs)
   print (Expression x)
   ifstmt (Expression cond, Statement thenpart, Statement elsepart)
   whilestmt (Expression cond, Statement body)
   seq (Statement s1, Statement s2)
   empty ()

}

domain Expression {

   eq (Expression x, Expression y)
   ne (Expression x, Expression y)
   lt (Expression x, Expression y)
   le (Expression x, Expression y)
   gt (Expression x, Expression y)
   ge (Expression x, Expression y)
   plus (Expression x, Expression y)
   minus (Expression x, Expression y)
   mult (Expression x, Expression y)
   divide (Expression x, Expression y)
   neg (Expression x)
   number (int x)
   name (int location)

}
```

Note that this definition can be read as a grammar defining the abstract syntax.

The definition not only provides documentation (as it is valuable even if we write the corresponding C/C++ data types and the functions manually), it also enables the Memphis precompiler to generate the implementation automatically.

## The Tree Walker

We are now ready to write the tree walker.

It will consist of two functions (one for each domain of the abstract syntax): `evaluate (Expression e)` that evaluates an `Expression` `e` and returns its numerical value, and `execute (Statement s)` that executes a `Statement` `s`.

Such functions are generally written by providing a piece of code for each possible alternative of the argument, where this code recursively visits the constituents the argument.

In Memphis we can use the `match` statement to describe this style of processing.

The `evaluate` function takes the form

```
int evaluate(Expression e)
{
   match e {
      ...
   }
}
```

The body of the `match` statement lists specific rules that handle the `Expression e` according to its structure.

One of these rules is

```
rule plus(x, y) : return evaluate(x) + evaluate(y);
```

If `e` has the form `plus(x, y)` then this rule is applied. It recursively evaluates `x` and `y` and returns the sum of their numerical values.

Here is the tree walker:

```
with ast;

extern "C" printf(...);
extern "C" execute(Statement s);

int var[26];

int evaluate(Expression e)
{
   match e {
      rule eq(x, y)     :  return evaluate(x) == evaluate(y);
      rule ne(x, y)     :  return evaluate(x) != evaluate(y);
      rule lt(x, y)     :  return evaluate(x) <  evaluate(y);
      rule le(x, y)     :  return evaluate(x) <= evaluate(y);
      rule gt(x, y)     :  return evaluate(x) >  evaluate(y);
      rule ge(x, y)     :  return evaluate(x) >= evaluate(y);
      rule plus(x, y)   :  return evaluate(x) +  evaluate(y);
      rule minus(x, y)  :  return evaluate(x) -  evaluate(y);
      rule mult(x, y)   :  return evaluate(x) *  evaluate(y);
      rule divide(x, y) :  return evaluate(x) /  evaluate(y);
      rule neg(x)       :  return - evaluate(x);
      rule number(x)    :  return x;
      rule name(x)      :  return var[x];
   }
}

execute (Statement s)
{
   match s {
      rule assignment(name(x), rhs) :
         var[x] = evaluate(rhs);
      rule print(x) :
         printf("%d\n", evaluate(x));
      rule ifstmt(c, s1, s2) :
         if(evaluate(c)) execute(s1); else execute(s2);
      rule whilestmt(c, s) :
         while(evaluate(c)) execute(s);
      rule seq(s1, s2) :
         execute(s1); execute(s2);
      rule empty() :
         ;
   }
}
```

Note that this notation is similar to the Yacc style. A syntactic pattern is followed by an associated action. But here the pattern describes abstract syntax instead of concrete source text.

Again, the notation is more concise than the corresponding manual implementation. The Memphis precompiler not only generates the implementation, it also allows to check statically that constituents are only used in a context where they are indeed fields of the actual item.