Memphis C/C++
A Language for Compiler Writers

This document gives an overview on *Memphis*, a *C/C++* extension for compiler writers and other programmers having to manipulate symbolic data.

Memphis provides a new kind of type declarations to describe such data via recursive definitions in a grammatical style. It also introduces a new statement to process such structures via rules that are selected by pattern matching. These concepts are known from functional languages [1] and modern compiler construction tools [2].

The new concepts are seamlessly integrated with the *C/C++* programming language [3, 4]. One may use the new features without giving up anything of the power and flexibility of the host language. *Memphis* is implemented by a precompiler.

The next section introduces *domain declarations* and *match statements*. We then show how the traditional *C/C++* concepts apply to the new data types. The last section provides an example how to write an interpreter with *Lex*, *Yacc*, and *Memphis*.

1 Domain Declarations and Match Statements

Memphis extends *C/C++* with *domain declarations* and *match statements*.

Consider the example in Fig. 1. This is a complete *Memphis* program; it uses a domain declaration to introduce a new data type `Tree`, and a match statement to process `Tree` values.

A `Tree` is given by a node that has an integer attribute `val` and two subtrees `left` and `right`. A `Tree` may also be the empty tree.

Here is graphical representation of a `Tree` that has a `val` field of `22` and two empty subtrees:

```
node
|
+- 22
|
+- empty
|
+- empty
```

The data type `Tree` is given by a domain declaration as follows.

```
domain Tree {
   node(int val, Tree left, Tree right)
   empty()
}
```

A domain declaration specifies a type by enumerating possible alternatives how to denote values of that type. According to the above declaration, values of type `Tree` have two alternative forms:

```
node(v, l, r)
```

or

```
empty()
```

An alternative specification gives a name for the alternative and lists the types and names of its fields. The alternative

```
node(int val, Tree left, Tree right)
```

from the above declaration defines values of kind `node`, they have an `int` field named `val` and two `Tree` fields named `left` and `right`.

Because `22` is an integer and `empty()` stands for the empty tree,

```
node(22, empty(), empty())
```

```
// Define a data type Tree.
// A Tree is either
//    a node with an int field val and two subtrees left and right
//    denoted node(val, left, right)
// or
//    an empty tree
//    denoted empty()

domain Tree {
   node(int val, Tree left, Tree right)
   empty()
}

// Define a function ExampleTree that constructs a Tree
//
//    node
//    |
//    +- 11
//    |
//    +- empty
//    |
//    +- node
//       |
//       +- 22
//       |
//       +- empty
//       |
//       +- empty

Tree ExampleTree ()
{
   return node(11, empty(), node(22, empty(), empty()));
}

// Define a function Sum that computes the sum
// of the values of a all nodes of a given Tree t

int Sum (Tree t)
{
   // inspect the structure of t

   match t {

      rule node(v, l, r) :

         // if t has the form node(v, l, r)
         // the result is obtained by adding v to the
         // recursively computed sums of l and r

         return v + Sum(l) + Sum(r);

      rule empty() :

         // if t is the empty tree
         // the result is 0

         return 0;

   }
}

// The main function

extern "C" void printf(...);

main ()
{
   printf ("Sum of ExampleTree is %d\n", Sum(ExampleTree()));
}
```

Fig. 1. An Example Program

stands for the `Tree` depicted above.

This notation can be used in an expression. The function `ExampleTree` returns a slightly more complex `Tree`:

```
Tree ExampleTree ()
{
   return node(11, empty(), node(22, empty(), empty()));
}
```

The value returned may be graphically represented as

```
node
|
+- 11
|
+- empty
|
+- node
   |
   +- 22
   |
   +- empty
   |
   +- empty
```

We now define a function `Sum` that computes the sum of the values of a all nodes of a given `Tree t`. This will be done by recursively following the structure of a `Tree`.

If `t` has the form

```
node(v, l, r)
```

the result is obtained by adding `v` to the recursively computed sums of `l` and `r`.

If `t` has the form

```
empty()
```

the result is `0`.

For example

```
Sum( node(22, empty(), empty()) )
```

is `22 + 0 + 0`, i.e. `22`.

The body of the function is given as a match statement.

```
int Sum (Tree t)
{
   match t {
      rule node(v, l, r) :
         return v + Sum(l) + Sum(r);
      rule empty() :
         return 0;
   }
}
```

A match statement names an item the structure of which has to be inspected.

```
match t {
   ...
}
```

means: inspect the structure of `t`.

It then lists in braces a number of rules from which one is selected according to the structure of the value. The rules

```
rule node(v, l, r) :
   return v + Sum(l) + Sum(r);
rule empty() :
   return 0;
```

cover the two cases.

A rule gives a pattern for a value and a list of statement that are executed when the given value matches that pattern.

Let `t` be the tree

```
node(22, empty(), empty())
```

and consider the rule

```
rule node(v, l, r) :
   return v + Sum(l) + Sum(r);
```

Matching the value

```
node(22, empty(), empty())
```

against the pattern

```
node(v, l, r)
```

succeeds. It also defines the variables `v`, `l`, and `r` as `22`, `empty()`, and `empty()`, respectively. These variables are implicitly declared as local to the rule.

With these values the rule body

```
return v + Sum(l) + Sum(r);
```

is executed. It returns the value `22 + 0 + 0`, i.e. `22`.

The second rule would not have been applicable, since

```
node(22, empty(), empty())
```

does not match the pattern

```
empty()
```

Memphis allows nested patterns of arbitrary depth. However, the two rules in our example use simple pattern that correspond to the two alternatives of the domain declaration. This style is very common: The structure of algorithm mirrors the structure of data.

2 Domains as Classes with Subclasses

In this section we discuss how traditional *C/C++* concepts apply to domain types.

As an example we use again the data type `Tree` as defined in the previous section. Let us define a function that increments the `val` fields of a given tree.

In *C/C++*, if `n` is a pointer to a structure representing a `node`, we can write

```
n->val
```

to designate its `val` field. We can use this as the target of an assignment and modify the field:

```
n->val = n->val + 1;
```

The same is possible possible in *Memphis*. We have to assert that `n` indeed refers to a `node`; if `n` would refer to the `empty` variant then there would be no `val` field. This can be done by assigning a name to a pattern as in

```
node(v,l,r) n
```

If a value matches the pattern `node(v,l,r)` then `n` represents this value which is now known to be of the specific variant.

We may omit arguments and simply write

```
node() n
```

to introduce `n`.

`n` acts as a variable local to the rule. We may access the fields of `n` using the notation `n->val`.

Here is a function `IncrementValFields(t)` that adds one to each `val` field of of each node `node(val, left, right)` of a given `Tree t`.

```
IncrementValFields(Tree t)
{
   match t {
      rule node() n :
         n->val = n->val + 1;
         IncrementValFields(n->left);
         IncrementValFields(n->right);
      rule empty() :
         ;
   }
}
```

Inside the body of the first rule we are able to access the fields of `n` using the traditional "`->`" notation. Sometimes we want to write a piece of code using this style, but we don't want to include that code into the body of rule. One reason could be that we want to package this code as a function that could be invoked from different places.

Let us define a function `IncrementValFieldOfNode(n)` that increments the `val` field of its argument by one. This function can only be invoked with `node` values, hence the argument type `Tree` would be to general.

There is also name for the specific type. It is obtained by appending "`_subtype`" to the name of the variant. For example, `node_subtype` is the specific type of `node` values.

Using this type we can write our function:

```
IncrementValFieldOfNode (node_subtype n)
{
   n->val = n->val + 1;
}
```

Because `n` is declared as of the specific type we can use `n->val` to denote its `val` field.

The function can only be invoked with an argument for which it is clear that it is an instance of the specific variant.

This is the case if the actual argument of the function is introduced using a pattern like

```
node() n
```

So we can rewrite our function using `IncrementValFieldOfNode`.

```
IncrementValFields(Tree t)
{
   match t {
      rule node() n :
         IncrementValFieldOfNode(n);
         IncrementValFields(n->left);
         IncrementValFields(n->right);
      rule empty() :
         ;
   }
}
```

We can also declare variables of the specific subtype:

```
node_subtype n;
```

A variable of a subtype may be used whenever a variable of the corresponding general type is valid. So we can invoke function `Sum` that expects a `Tree` value with this variable:

```
Sum(n);
```

The reason is that a value of the `node` variant is also a value of the `Tree` type.

The value of an expression of the form

```
node (v, l, r)
```

is actually a value of the specific subtype. It can be assign to a subtype variable as well as to variable of the domain type.

```
n = node(11, empty(), node(22, empty(), empty()));
```

is valid if `n` is declared as above.

Whereas the style discussed in the preceding section leads to a declarative flavour of programs, the style discussed here supports an imperative style where you can modify values by side effects.

Note that the pure declarative style supports tree-like values, whereas the imperative style can also deal with general graphs. Modifications of fields allow to introduce arbitrary connections including cycles.

Experience in many projects has shown that the declarative style leads to more readable and reliable code and is sufficient for most cases. The imperative style should be used as the exception, not as the rule.

3 Writing an Interpreter with Lex, Yacc, and Memphis

In this section we apply the new concepts and show how to write an interpreter with *Lex* [5], *Yacc* [6], and *Memphis*. (See [7] for a more detailed discussion of how to use *Lex* and *Yacc* and how to define and process abstract syntax.)

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
```

```
%{
#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
```

Fig. 2. Lex Specification

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.

Fig. 2 is the input to *Lex*.

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 a token 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
;
```

```
%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); }
;
```

Fig. 3. Yacc Specification

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`.

Fig 3 is the input to *Yacc*.

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

```
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)

}
```

Fig. 4. Abstract Syntax in Memphis

```
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)
```

Fig. 4 gives the complete definition of the abstract syntax.

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.

```
with ast;

extern "C" void 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() :
         ;
   }
}
```

Fig. 5. Tree Walker in Memphis

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.

Fig. 5 describes the tree walker.

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.

References

[1] Robert Harper, Robin Milner, Mads Tofte:
The Definition of Standard ML (Version 2),
MIT Press (1989)

[2] Friedrich Wilhelm Schröer:
Gentle,
In Studien der GMD 166,
German National Research Center for Information Technology (1989)

[3] Brian W. Kernigham, Dennis M. Ritchie:
The C Programming Language, Second Edition,
Prentice Hall (1978)

[4] Ellis Stroustrup:
The C++ Programming Language, Third Edition,
Addison Wesley Longman (1997)

[5] Mike E. Lesk:
Lex - A Lexical Analyzer Generator,
Comp. Sci. Tech. Rep. No. 39,
Bell Laboratories (1976)

[6] Stephen C. Johnson:
Yacc - Yet Another Compiler-Compiler,
Comp. Sci. Tech. Rep. No. 32,
Bell Laboratories (1975),

[7] Andrew W. Appel with Maia Ginsburg:
Modern Compiler Implementation in C,
Cambridge University Press (1998)

See Also

Memphis Language Reference Manual
Memphis User Manual
`memphis.compilertools.net`