Memphis
User Manual

This document describes how to install and use the precompiler for *Memphis*, a *C/C++* extension for compiler writers.

1 Installation

For *Unix* systems *Memphis* is distributed as a compressed tar file `memphis-1.9.tar.gz` . To unpack this file type

```
gunzip memphis-1.9.tar.gz
tar -xvf memphis-1.9.tar
```

This creates a directory `memphis-1.9` with the following subdirectories

`memphis`	The *C* code of the *Memphis* precompiler
`mrts`	Contains a simple module to be linked with *Memphis* programs
`examples`	Some small examples
`doc`	The documentation

Go to directory `memphis` and type

```
build
```

This compiles the *Memphis* precompiler.

Go to directory `mrts` and type

```
build
```

This compiles the file `mrts.cpp` that must be linked to a *Memphis* program.

To test your installation you may go to directory `examples/trees` and type

```
build
```

This translates and executes a simple *Memphis* program.

The program should print the result

```
Sum of ExampleTree is 33
```

The next section discusses how to process this example.

2 Processing a Simple Program

The directory `examples/trees` contains a file `prog.m` as depicted in Fig. 1. This file provides a *domain declaration* to introduce a data type `Tree` and a function that uses a *match statemenent* to process `Tree` values (see the *Memphis C/C++, A Language for Compiler Writers* for details).

Here is a shell script (file `build`) to compile this file and to execute the resulting program.

```
// 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. A Simple Program (`prog.m`)

```
MEMPHIS=../../memphis/memphis
MRTS=../../mrts/mrts.o
CPP=g++

$MEMPHIS prog.m
$CPP -c prog.cpp
$CPP -o prog prog.o $MRTS
./prog
```

The first threee lines define the location of the *Memphis* precompiler, the path of the *Memphis* runtime system, and the *C++* compiler.

The next lines invoke the *Memphis* precompiler, the *C++* compiler, and then the resulting program.

The command

```
$MEMPHIS prog.m
```

invokes the *Memphis* precompiler with the source file `prog.m`. *Memphis* program files must have the extension "`.m`". The command creates target file `prog.cpp`.

The precompiler analyses domain declarations and match statements and translates them into `C++` code. Other material (i.e. ordinary *C/C++* code) is passed to the target file just as it is.

The command

```
$CPP -c prog.cpp
```

invokes the *C++* compiler to translate the generated file `prog.cpp`. This results in an object file `prog.o`.

The command

```
$CPP -o prog prog.o $MRTS
```

links the object file `prog.o` and the *Memphis* runtime system `mrts.o`. The result is an executable program `prog`.

The *Memphis* runtime system merely provides an error function that is called when a match statement fails.

The command

```
./prog
```

finally invokes the executable program.

```
// Define a data type Tree.

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

Fig. 2. Tree Definition (`treedef.m`)

```
// Use the Tree definition

with treedef;

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

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

// The main function

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

main ()
{
   Tree ExampleTree = node(11, empty(), node(22, empty(), empty()));

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

Fig. 3. Tree Application (`treeuse.m`)

3 Multiple File Programs

A *Memphis* program may also be composed from several files. Some files may provide domain declarations that are then used in several other files.

For example, we can split the program discussed in the previous section (Fig. 1) into two files:

The file `treedef.m` (Fig. 2) introduces a domain type `Tree`. Separated in this way, the `Tree` defiinition could be used at several places.

The file `treeuse.m` (Fig. 3) applies the definition. It refers to it using a *with clause*

```
with treedef;
```

that makes the data type visible for this file.

These files are contained in directory `examples/multi`. Here is a script (file `build`) to process them:

```
MEMPHIS=../../memphis/memphis
MRTS=../../mrts/mrts.o
CPP=g++

$MEMPHIS -h treedef.m

$MEMPHIS treedef.m
$MEMPHIS treeuse.m

$CPP -c treedef.cpp
$CPP -c treeuse.cpp

$CPP -o walker treedef.o treeuse.o $MRTS

./walker
```

If a file contains domain declarations that should be available to other files then this file must be processed twice by the *Memphis* precompiler. If the precompiler is invoked with the "`-h`" option it only processes domain declarations and generates interface information (this is used later when files are processed that refer the domain declarations). Then the precompiler is invoked to produce a *C++* file as described above in the previous section.

The command

```
$MEMPHIS -h treedef.m
```

invokes the *Memphis* precompiler to generate interface information for the domain declarations in file `treedef.m`.

This results into three files:

`treedef.sig`	contains the *signatures* of the domain declarations (i.e. an abstract representation) that is read when another *Memphis* file refers `treedef.m`.
`treedef.h`	contains *C++* headers that are included into *C++* code that is generated for those files that refer `treedef.m`.
`treedef.f`	is used like `treedef.h` but contains only forward declarations that allow mutually recursive domains in the generated *C++* code.

The command

```
$MEMPHIS treedef.m
```

generates the *C++* implementation of `treedef.m` and results in a file `treedef.cpp`.

The command

```
$MEMPHIS treeuse.m
```

generates the *C++* implementation of `treeuse.m` and results in a file `treeuse.cpp`.

The file `treeuse.m` contains a clause

```
with treedef;
```

Hence the file `treedef.sig` must be available. The generated *C++* code contains `include` statements for `treedef.h` and `treedef.f`.

The commands

```
$CPP -c treedef.cpp
$CPP -c treeuse.cpp
```

compile the *C++* files `treedef.cpp` and `treeuse.cpp` resulting in object files `treedef.o` and `treeuse.o`.

When compiling `treeuse.cpp`, `treedef.h` and `treedef.f` must be available.

The command

```
$CPP -o walker treedef.o treeuse.o $MRTS
```

links the object files `treedef.o` and `treeuse.o` and the *Memphis* runtime system. The result is an executable program `walker`.

The command

```
./walker
```

invokes the executable program.

In *Memphis*, domain declarations may be mutually recursive. Similarly, *Memphis* files may refer each other in a cyclic way.

To process such a program first precompile the files that appear in `with` clauses, where you use the "`-h`" option to generate interface information.

Then precompile all *Memphis* files to generate *C++* code.

Note that the interface information must be regenerated when a domain declaration is changed.

4 Interoperability with Classical C

Memphis has been designed such that it can be used by *C* programmers who are not familiar with *C++*. It can be understood as a language extending *C* by domain declarations and match statements.

However, domain types are mapped on *C++* classes, and hence a *C++* compiler is required to compile the files generated by *Memphis*. Because *C* is a subset of *C++*, a programmer can write *Memphis* code without using *C++* specific features.

A notable exception to this is that *C++* uses so-called *name mangling* to encode type information into the names of functions. This has to be suppressed if one wants to call a *C* function from *Memphis code*. A way to do this is to declare a *C* function as `extern "C"`. For example, in our introductory example (Fig. 1) we used the classical *C* style for output. This reqired a declaration

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

Just as classical *C* functions can be called from *Memphis*, *Memphis* terms can also be constructed in classical *C* code, i.e. code that needs to be processed by a classical *C* compiler. This is made possible because the *Memphis* precompiler declares the generated functions as `extern "C"`.

Hence one can construct *Memphis* trees inside the semantic actions of *Yacc* code, although this code will be compiled by a *C* compiler without precompilation with *Memphis*.

See the example in directory `example/inter` and the discussion in the *Memphis C/C++, A Language for Compiler Writers*
how to write an interpreter using *Lex*, *Yacc*, and *Memphis*.

5 Interoperability with Gentle

Memphis code can also process terms that are generated by code which is written in the *Gentle* compiler description language.

Like *Memphis*, *Gentle* supports domains types and pattern matching. Unlike *Memphis*, *Gentle* provides a high level of abstraction to express language recognition, transformation, and code generation in a uniform way.

Combining *Gentle* and *Memphis* allows one to use a specialized and productive compiler description language for the translation tasks of an application, and also to use *C/C++* programming, e.g. to implement the user interface. *Memphis* enables seamless integration of *Gentle* and *C/C++*.

Because *Gentle* is able to capture most errors by analyzing the specification it drastically reduces debbugging efforts. Its uniform framework provides a strong guidance when designing a language translator. *Gentle* offers capabilities not present in *Memphis*, such as strongly typed grammar annotation for mapping concrete to abstract syntax, smart traversal of trees (eliminating the need to write "trivial" rules), and dynamic programming for optimal code selection.

Just as a command

```
memphis -h treedef.m
```

makes the types defined in the *Memphis* file `treedef.m` available to other *Memphis* files, a command

```
gentle -h treedef.g
```

makes the types defined in the *Gentle* file `treedef.g` available to *Memphis* programs.

See the example in directory `examples/polish` how to write a translator in *Gentle* and add a function in *Memphis*.

See Also

Memphis C/C++, A Language for Compiler Writers
Memphis Language Reference Manual
`memphis.compilertools.net`