Exercise Compiler Construction (9)

Hans de Nivelle

Due: Jan 2010

In this exercise, we will implement a simple stack machine. In the next exercise, we will write a compiler that translates a higher-level programming language into this stack machine.

1. The stack machine operates only on **double**. It has two stacks, one for the double arguments, and one for return addresses.

   All intermediate results, and also all local variables are stored on the stack. Most of the instructions take their operands from the top of the stack, and store their result back on the top of the stack. We first list the simple instructions:

   add Add the two numbers of the top of the stack, remove them, and put the result back on the stack.

   multiply/divide/sub Multiply/divide/subtract the two numbers on the top of the stack, remove them, and put the result back on the the stack. You may ignore the problem of division by zero. (Just define $x/0 = 0$)

   negate Negate the number on the top of the stack. (Replace x by $-x$.)

   less/greater/lessequal/greaterequal/equal/notequal Compare the two numbers on the top of the stack, remove them, and write back 1.0 if the comparison succeeds. Otherwise, write back 0.0.

   push X Push the constant X on the stack.

   The read/write operations write or read the top of the stack from a deeper position. There operations are required for storing/retrieving local variables.

   read n Read the number that is n positions back at the stack, and push a copy of it on the stack. n is an unsigned integer.

   pop Remove the top of the stack. Do nothing with the result.

   write n Write the number on the top of the stack n positions back at the stack, and remove it. $n$is an unsigned integer.

jump *n* Jump *n* positions forward or backward. *n* is a signed integer.

jumpnonzero *n* Jump *n* positions forward or backward if the number on the top of the stack is not 0.0. In all cases, the number on the top of the stack is removed.

jumpzero *n* Same as previous, but jump when zero.

call *n* Jump *n* positions forward or backward, but remember the return adress.

return Jump to a remembered return adress.

Define some data type

```
enum opcode = { ..... };
struct instruction
{
   opcode code;
   double d;
   int x;       // Also for unsigned int.
};
   // In practice, one would use some more low-level implementation,
   // but for this exercise, this is good enough.
```

Implement this stack machine. Add a trace function. Try it out on:

```
   int fact(x)
   if( x == 0 )
      return 1;
   else
      return x * fact(x-1)



   // On entry, we assume that x is on the stack.

fact:
   read 0;    // Put copy of x on the stack.
   push 0.0;  // Put 0 on the stack.
   equal;     // Compare the two numbers.
   jumpzero rec   // In reality, this is a number (3)

   pop;       // Remove local paramter x.
   push 1.0   //
   return

rec:
```

```
read 0;     // Put copy of x on the stack
read 1;     // Another copy of x.
push 1.0
sub;        // Now we have x, (x-1) on the stack.
call fact;  // Replaces (x-1) by fact(x-1)
             // In reality, fact is a number -12.
multipy;     // Now we have x.fact(x-1) on the stack.
store 1;     // Overwrite local variable x with result.
return;
```

2. Write some more functions, and try the stack machine on them.