Final Project Compiler Construction

Hans de Nivelle

Due: Feb 26, 2010

The final project is based on Exercise 9 about stack machines. First add the following 3 instructions to the implementation of the stack machine. That should't be too much work.

output Write the floating point number on the top of the stack to standard output and remove it from the stack.

input Read a number from standard input and push in on the top of the stack.

outputchar Interpret the number of the top of the stack as character (using its ASCII code) and print it to standard output.

1 Task Description

The task is to write a compiler that compiles some programming language into instructions for the stack machine. You may either create the instructions directly directly, or create some type of assembler text that can be read by the stack machine. The language must have the following constructions:

- `if ... then ... else ...` , where the `else` part is optional.
- `while ... do ...`
- `do ... while ...`
- Some construction for grouping statements, either `{ ... }` or `begin ... end`.
- The language must have the usual operators, with sensible priorities and associativities: Arithmetic operators `+, -, *, /,` . Comparison operators `==, !=, <, <=, >, >=` Note that the comparison operators create numbers (1.0 when the outcome is true, 0.0 if the outcome is false.)
- `print "....."` Print the string to standard output. (This must be translated into a sequence of **outputchar**s.
- `print ...` Print a double.
- `read ...` Reads a double.

- `return ...`
- A program consists of a collection of function definitions. Execution starts at a function with name `main`, which has no arguments.

  You many assume that every function, except `main` returns a single `double`. You may also assume that no function is used before it is defined. (No forward declarations.)
- Local variables must have initializers, as in `local a  = 1`.

.

`#if I_WANT_TO_IMPRESS_MY_TEACHTER == 1`

you may add one or more of the following features:

- Local variables in every block, not only at function definitions.
- A conditional operator operator `... ?   ... : ...` . Logical operators `||, &&, !`, that evaluate their arguments only when required. (These operators can be defined using `... ? ... : ...`)

`#endif`

2 Example Program

```
// This could be an implementation of factorial:

factorial( n )
{                      // You may also use 'begin'
   if( n == 0 )
      return 1
   else
      return n * factorial( n - 1 );
}


// This is another possibility:

factorial2( n )
{
   local result = 1;
   while( n > 0 )
   {
      result = result * n;
      n = n - 1;
   }
}


main( )
{
   local a = 0;
   print "please type a number: ";

   read a;
   print "factorial is";
   print factorial(a);
}
```

3 Suggestions

As far as I can see, the hardest part of the project is implementing the following three containers:
Container for Local Variables

The container must be able to store the addresses of local variables. It must be able to know how many additional, nameless results are on the stack, so that it can decide how far a variable is away from the top of the stack. It should have the following methods:

- Add a local variable.
- Add a nameless variable. (That corresponds to an intermediate value of an expression.)
- Popping a local variable. (Corresponds to the fact that during evaluation of an expression, the stack decreases.)
- Looking up a local variable. It should return a number that specifies how far away from the top-of-stack the local variable is situated.
- Clear. If you have local variables only the start of a function block, it is enough to have a clear method. If you have local variables at every block, you need to remember the length of the container at each block entry, and restore the length when you are finished with the block.
- size. Returns the size of the container. This is needed when you want to return a value, and want to decide where to write the result.

Code Container
The most elegant solution is to generate a kind of readable assembler text for the stack instructions. In that case, you don't need the code container. The code container is in principle easy, but it needs to be able to deal with the fact

that there exist forward jumps. When the compiler generates the code for the jump, it does not yet know the address of the jump, so that it has to come back later and fill in the address.

The best solution is to let the code container do this. Create a label class that is associated to the code container. The code container can be asked to create a new label, after which the label can be used in emitted code. As long as the label has no definition, the code container will keep track of where the label was used. When a definition is given, the code container fills in the correct address everywhere where the label was used. If the label is used more after its definition, the code container can simply fill in the value.
The code container should have the following methods:

- Append a statement to the code. The statement can contain labels. If the value of the label is known, it can be replaced and forgotten, otherwise it has to be remembered.
- Create a new label.
- Assign a value to a label.

Container for Function Definitions
This container is easy. It contains a map from pairs containing a function name and an arity, to the starting addresses and the return type (a single number or nothing) of the function.

4 Parser Generator

You can try your luck without, but believe me, the project is a lot easier if you use a parser generator. In all cases, I want to see a formal grammar of the language.

You should make the three containers as global variables, and add actions to the rules that emit the code. (Use `%global`) for this.