

Exercise sheet 7 for 26 March 2003

2003-03-19

Do exercises 7.1, 7.2, 7.3. Hand in the solutions.

Exercise 7.1 Compile the example micro-C programs `imp/ex3.c` and `imp/ex5.c` and `imp/ex9.c` using `compile2file` from `comp.sml`.

Study the generated symbolic machine code for some small programs. Write up the code in a more structured way with labels only at the beginning of the line (as in the lecture). Write the corresponding micro-C code to the right of the stack machine code.

Run the compiled programs using `java Machine ex3.out 10` and similar. Note that these programs require a command line argument (an integer) when they are executed.

Try to trace the execution using `java Machinetrace ex3.out 4`. Check that you understand the stack contents (the stack top is to the right).

Exercise 7.2 Compile and run the micro-C example programs you wrote in Exercise 6.1. It is rather cumbersome to fill an array with values by hand, so the function `squares` from that exercise is very handy.

Exercise 7.3 This abstract syntax for postincrement `i++` and postdecrement `i--` was introduced in Exercise 6.2:

```
datatype expr =
  ...
  | PostInc of access (* Java i++ or a[e]++ *)
  | PostDec of access (* Java i-- or a[e]-- *)
```

Modify the compiler (function `cExpr`) to generate code for `PostInc(acc)` and `PostDec(acc)`.

This should be done so that the address is computed only once. For instance, `i++` should compile to something like this: `GETBP, <offset>, ADD, DUP, LDI, CSTI 1, ADD, STI, CSTI 1, SUB` where the address of `i` is computed once by `GETBP, <offset>, ADD` and then duplicated.

The final `CSTI 1, SUB` is a hack to leave the value of the variable *before* increment on the stack top.

Write a program to check that this works. If you are brave, try it on expressions of the form `a[i++]++` and check that `i` and the elements of `a` have the right values afterwards.

Exercise 7.4 Compile `ex8.c` and study the symbolic machine code to see why it is so much slower than the hand-written 20 million loop in `imp/prog1`.

Compile `imp/ex13.c` and study the symbolic machine code to see how loops and conditionals interact.

In lecture 9 we shall see an improved compiler that it generates much fewer extraneous labels and jumps.

Exercise 7.5 (Would be convenient) Write a disassembler that can display a machine code program in a more readable way. You can write it in Java, using a variant of the method `insname` from `imp/Machine.java`.

Exercise 7.6 Write more micro-C programs; compile and disassemble them. (Warning: The abstract machine and the compiler have not been tested systematically, so in case of confusion, don't hesitate to write me an email).

For instance, write a program to make a table of the frequencies of a list of 20 numbers in the interval 0-99 given as arguments on the command line.

Study the effect of nested scopes (blocks) on the result of compilation.

Exercise 7.7 (May be more complicated) Extend the language and compiler to accept initialized declarations such as

```
int i = j + 32;
```

Doing this for local variables (inside functions) should not be too hard, but for global ones it requires more changes.

Exercise 7.8 (May be more complicated) Extend the micro-C language, the abstract syntax, the lexer, the parser, and the compiler to implement conditional expressions of the form `(e1 ? e2 : e3)`.