

Exercise sheet 9 for 17 April 2002

2002-04-10

Do exercises 9.1, 9.2, 9.3, and 9.4. Hand in the solutions.

Exercise 9.1 The continuation-based micro-C compiler (file `imp/contcomp.sml`) still generates clumsy code in some cases. For instance, the statement (file `imp/ex16.c`):

```
if (n)
  { }
else
  print 1111;
  print 2222;
```

is compiled to this machine code:

```
GETBP, LDI, IFZERO L2,
GOTO L1
L2: 1111, PRINTI, INCSP ~1,
L1: CSTI 2222, PRINTI
```

which could be optimized to this by inverting the conditional jump:

```
GETBP, LDI, IFNZRO L1,
1111, PRINTI, INCSP ~1,
L1: CSTI 2222, PRINTI
```

Improve the compiler to recognize this situation. It must recognize that it is about to generate code of one of these forms:

```
IFZERO L2, GOTO L1, L2: ....
IFNZRO L2, GOTO L1, L2: ....
```

where a conditional jump just skips over an unconditional jump. In those cases it should instead generate code such as this:

```
IFNZRO L1, L2: ....
IFZERO L1, L2: ....
```

Exercise 9.2 Improve code generation in the continuation-based micro-C compiler so that a less-than comparison with constant arguments is compiled to its truth value. For instance, `11 < 22` should compile to the same code as `true`, and `22 < 11` should compile to the same code as `false`. This can be done by a small extension of the `addCST` function.

Further improve the code generation so that all comparisons with constant arguments are compiled to the same code as `true` (e.g. `11 <= 22`, `11 != 22`, `22 > 11`, `22 >= 11`) or `false`.

Check that `if (11 <= 22) print 33;` compiles to code that unconditionally executes `print 33` without performing any test at all.

Exercise 9.3 Extend the micro-C abstract syntax (file `imp/Absyn.sml`) with conditional expressions `Cond(e1, e2, e3)`, corresponding to the C/C++/Java/C# concrete syntax:

```
e1 ? e2 : e3
```

The expression `Cond(e1, e2, e3)` must evaluate `e1`, and if the result is non-zero, must evaluate `e2`, otherwise `e3`. (If you want to extend also the lexer and parser to accept this new syntax, then note that `?` and `:` are right associative. But implementing them in the lexer and parser is not strictly necessary for this exercise).

Extend the continuation-based micro-C compiler (file `imp/contcomp.sml`) to compile conditional expressions to stack machine code. Check that your compiler compiles the following kinds of examples properly:

```
true ? 1111 : 2222
```

```
false ? 1111 : 2222
```

The abstract syntax for the first expression is `Cond(Cst(CstI 1), Cst(CstI 1111), Cst(CstI 2222))`.

Exercise 9.4 Note that

$$\begin{aligned} e1 \ \&\& \ e2 \ \text{ is equivalent to } & (e1 \ ? \ e2 \ : \ 0) \\ e1 \ || \ e2 \ \text{ is equivalent to } & (e1 \ ? \ 1 \ : \ e2) \end{aligned}$$

Does your extended compiler (from Exercise 9.3) optimize $(e1 \ ? \ e2 \ : \ e3)$ so well that you can just implement `e1 && e2` and `e1 || e2` as shorthands for conditional expressions? Test it on file `imp/ex13.c`, for example. If it does optimize well enough, you can drop `Andalso` and `OrElse` from the abstract syntax, and considerably simplify `cExpr` in the compiler, which is desirable.

Exercise 9.5 Improve the compilation of assignment expressions that are really just increment operations, such as

```
i = i + 1
```

and possibly

```
a[i] = a[i] + 1
```

It is best to recognize such cases in the abstract syntax, not by looking at the code continuation.

Exercise 9.6 Try to make sense of the code generated by the continuation-based compiler for the n -queens program in file `imp/ex11.c`. Draw a flowchart.