

CS 558 Homework 4 – due 5pm, Tuesday, February 5, 2008

Homework must be submitted by mail to `apt@cs.pdx.edu`, with the subject line “CS558 HW4”. All submitted files (two for this assignment) must be sent as plain-text *attachments* to the mail message (the contents of the message itself will be ignored). It is *your* responsibility to submit the homework in the proper format.

All programs mentioned can be downloaded from the course web page.

1. Consider a variant of our familiar simple language with imperative expressions and functions (but no global variables or pairs), which we’ll call “E4.” Its “concretized” abstract syntax is given by the following grammar:

```
prog := '(' { fundef } ')' exp
fundef := '(' 'fun' fname '(' { var } ')' exp ')'
exp := var
      | int
      | '(' ':' var exp ')'
      | '(' 'while' exp exp ')'
      | '(' 'if' exp exp exp ')'
      | '(' 'write' exp ')'
      | '(' 'block' { exp } ')'
      | '(' '@' fname { exp } ')'
      | '(' '+' exp exp ')'
      | '(' '-' exp exp ')'
      | '(' '*' exp exp ')'
      | '(' '/' exp exp ')'
      | '(' '<=' exp exp ')'
fname := letter { letter | digit }
var := letter { letter | digit }
```

As before, comments may be included by enclosing them between `'{'` and `'}'` characters, and they may be nested.

The semantics of E4 expressions and functions are similar to those of E3. However, all scoping is static. The scope of each function name is the entire program, allowing two or more functions to be mutually recursive. The only variables are function parameters; there are no globals. It is a static error to use an undefined function or variable name. All function arguments are passed by value.

An E4 interpreter in SML (only) has been provided (`hw4_1.sml`). It reads a file containing an E4 program in the syntax described above, echoes the program (to confirm correct parsing), translates it into stack machine code, prints out the stack machine code (for debugging purposes), executes it (possibly producing output from `write` expressions), and displays the overall result. For more debugging help, you can trace the behavior of the stack machine by executing the top-level command `Machine.traceOn := true;` before executing `Interp.interp`.

The stack machine has been substantially altered to deal with functions and their parameters. There is no longer a dynamic environment; instead, variables (which are all function parameters) are stored on the stack at statically known offsets from the frame pointer. The `LOAD` and `STORE` instructions now operate on the stack location whose *address* is at the top of the stack; the `FPOFF` instruction computes and pushes such addresses. There are `CALL` and `RET` instructions to transfer control to and from subroutines (which have ordinary code labels). You are strongly advised to work through the mechanics of simple subroutine calls before making the changes specified below.

Your task is to change the E4 language and the interpreter to support (optional) call-by-reference function arguments. The revised syntax is

```
fundef := '( 'fun' fname '( { ['!'] var } )' exp )'
```

The optional `!` indicates that the argument is to be passed by reference rather than by value. The parsing support for this extension is already present; what you need to do is extend the abstract syntax for function definitions, the printing code, and the compilation code. Do *not* change or add to the stack machine. Put your solution in a file `sol4.1.sml`.

The actual argument provided for a CBR parameter must be a variable (that is, a parameter of the calling function); if a more complicated expression is provided, your compiler should issue a static error.

As an example, the following code should produce the answer 111. (If the `!` is removed, it should produce the answer 101.)

```
(
  (fun doadd (!x y) (:= x (+ x y)))

  (fun f (a)
    (block
      (@doadd a 10)
      (+ a 100)))
)
(@f 1)
```

Implementation hints: Add a boolean flag to the AST for each function parameter to indicate whether it is passed using CBR or not. You'll need to make changes to the compilation code to keep track of which (if any) parameters of the current function are CBR, and adjust the code generated for `VAR` and `ASGN` expressions accordingly. You'll also need to keep track of whether a function being *called* has CBR arguments, and if so, adjust the way code is generated for those arguments.

2. Consider the quicksort program in file `hw4.2.java`. Systematically remove the recursion from function `quicksort`, following the general approach described in lecture for `printtree`.

Do not change the `partition` function. Your modified version of `quicksort` must be non-recursive, and should make the same sequence of calls to `partition`, with the same arguments, as the original function does.

Hints: You may find it easier to develop your solution in C (or a pseudo-Java with labels and `gotos`), so that you can use `goto` in the intermediate stages of your program transformation. Eventually, you'll need to use only structured control operators. Note that you'll need to stack *two* pieces of information; it is probably easiest to use two parallel stacks.

Put your modified program into file `sol4_2.java` and submit this file.