Assessed coursework – code generation for functions

Notes and feedback on solutions

Most people did pretty well

There were a few subtle errors…

- Not removing the parameter register when evaluating the actual parameter expression
- Forgetting to reverse pop instructions when restoring registers
- What is the “weight” of “apply”?

Keeping the parameter register safe

```
transExp (Apply s e) (dst:rest)
 = (saveRegs ...) ++
   transExp e (paramReg:dst:rest) ++
   [Jsr s] ++
   [Mov (Reg resultReg) (Reg dst)] ++
   (restoreRegs ...)
```

Simple, elegant, wrong

In *two* ways…

Keeping the parameter register safe

```
transExp (Apply s e) (dst:rest)
 = (saveRegs regsInUse) ++
   transExp e (allRegs \\ [paramReg]) ++
   [Mov (Reg (head (allRegs \\ paramReg))(Reg paramReg)] ++
   [Jsr s] ++
   [Mov (Reg resultReg) (Reg dst)] ++
   (restoreRegs regsInUse)
   where regsInUse = allRegs \\ (dst:rest)
```

After saveRegs, *all* registers are free

Except one

Once registers have been saved, they're all free

Push push push, pop pop pop

```
saveRegs regsInUse
 = [Mov (Reg r) Push | r <- regsInUse]

restoreRegs regsInUse
 = [Mov Pop (Reg r) | r <- reverse regsInUse]
```

weight of “apply”

- The purpose of the “weight” is to decide the order of subexpression evaluation
- Consider:

  e1 + f(x)

  Which is better – e1 first? Or do the call first?

weight of “apply”

- The purpose of the “weight” is to decide the order of subexpression evaluation
- Consider:

  e1 + f(x)

  Which is better – e1 first? Or do the call first?

  If you evaluate e1 first, you have to store it in a register

  When you call “f”, you have to save and restore that register

weight of “apply”

- The purpose of the “weight” is to decide the order of subexpression evaluation
- Consider:

  e1 + f(x)

  Which is better – e1 first? Or do the call first?

  If you evaluate e1 first, you have to store it in a register

  When you call “f”, you have to save and restore that register

  So better to give function calls a high weight

Interprocedural register allocation

- We should be able to avoid unnecessary saving of registers, and we should be able to pass a callee's parameters/results without displacing the caller's.
- Class/file-private functions could be register-allocated at compile-time, but to do it globally it would have to be at link-time.
- It's not common because linking has to be fast but operates on whole programs
- It's also infeasible when linking dynamically.
- It's attractive where linking is static, and inlining is undesirable - eg in space-constrained embedded systems
- And when you have lots of registers

Examples

- SN systems compilers for Sony Playstation 2 and PSP:
  - http://www.snsys.com/psp/prodg.asp
- Altium's VX compiler toolset for the ARM processor:
  - http://www.tasking.com/products/ARM/ARM-ds.pdf
- Lots of research projects

- Profile-directed:
  - Allocate registers to values used on hot paths
- “leaf” functions are functions that have no callees
  - “About 1/3” of functions are “syntactic” leaf functions
  - “About 2/3” of actual calls are to “effective” leaf functions - no callee is actually called
- Allocate so spill code is on cold paths