CS362 2011F *Compilers*

Class 40: Translating Procedure Calls

Held: Monday, 5 December 2011

Summary: Today we conclude our discussion of translation by considering the translation of function and procedure calls.

Related Pages:

- EBoard.
- Reading: Aho *et al.* 8.4.

Overview:

- Leftovers: For loops.
- Format of stack frames.
- Tracing the stack.
- Function/procedure initialization.
- Function/procedure cleanup.
- Division of responsibilities.

A Left-over Question: For Loops

- We've noted that there may be a problem with going beyond the upper bound in the for loop.
- What's your solution?

Translating Procedure Calls

- We've come full circle in our discussion of translation.
  - We started by thinking about stack frames as a mechanism for holding parameters and local variables.
  - We now return to stack frames from the perspective of the code needed to build and use them.

Translating Declarations

- Our translation of procedure calls depends first upon the translation of procedure declarations.
- That, in turn, depends on our design for stack frames (and for the stack).

The Stack Frame, Revisited

- Review: What goes in the stack frame?
- My list includes
  - Space for the return value (if there is one)
  - The previous frame pointer (aka base pointer)
  - The return address
  - Pointers to parent frames (for ancestor variables)
  - Parameters
  - Local variables
  - Local temporaries
- What else does your list include?
- Here's what should happen in a procedure call.
  - Initially, the stack looks like this (with invented addresses)

```
     | ... |
     |     |  <- fp
832  +-----+
     | ... |
     |     |  <- sp
814  +-----+
     | ... |
```

  - While the procedure is executing, the frame pointer has moved to the start of the next frame, and the stack pointer has moved to the end of that frame

```
     | ... |
     +-----+
     |     |
832  +-----+
     | ... |
     |     |  <- fp
814  +-----+
     | ... |
     |     |
     |     |  <- sp
     +-----+
     | ... |
```

  - Finally, when the procedure finishes, we want to restore the previous state of the stack.

```
     | ... |
     |     |  <- fp
832  +-----+
     | ... |
     |     |  <- sp
814  +-----+
     | ... |
```

- How do we arrange this? Typically, when we call a procedure, we put the previous frame pointer on the stack.

```
     | ... |
     |     |  <- fp
832  +-----+
     | ... |
     | 832 |  <- sp
814  +-----+
     | ... |
```

- Then, when we set the frame pointer to the stack pointer and increment the stack pointer by the amount of space needed for the AR.

```
     | ... |
     |     |
832  +-----+
     | ... |
     | 832 |  <- fp
814  +-----+
     | ... |
     |     |  <- sp
     +-----+
```

- Restoring the stack is then fairly easy. We can just write

```
MOV %sp %fp
MOV %fp %fp(0)
```

- Note that his restoration works *no matter how much more we've added to the stack.*

Who Maintains the Stack?

- We need to consider whether caller or callee takes responsibility for different aspects of the maintenance.
  - Remembering and updating old frame pointer.
  - Remembering and updating old stack pointer.
- You'll note that there are many tradeoffs in these decisions
  - We can have caller push parameters and callee push room for local variables, but that seems to be two sets of pushes. However, if the caller pushes all the necessary space, it needs to know how much the callee is likely to need.
- My normal strategy is to give the responsibility to the caller.