SICP section 5.1

February 22, 2008 at 17:44 Tags SICP

The code for chapter 5 will be written in Scheme[1] (using PLT Scheme).

Introduction

As usual, I've jumped forward and implemented the register machine simulator of section 5.2, in order to be able to test the solutions to the exercises of this section. The simulator is available for download here, with a test suite (using the SchemeUnit package) here.

Exercises 5.1 – 5.2

I'll skip the drawings, and will present the code code the iterative factorial process with the register-machine language:

```
(define iter-fact
  (make-machine
    '(n product counter)
    `((> ,>) (* ,*) (+ ,+))
    '(
      init
        (assign counter (const 1))
        (assign product (const 1))
      loop
        (test (op >) (reg counter) (reg n))
        (branch (label end-fib))
        (assign product (op *) (reg counter) (reg product))
        (assign counter (op +) (reg counter) (const 1))
        (goto (label loop))
      end-fib
    )))

(set-register-contents! iter-fact 'n 6)
(start iter-fact)
(get-register-contents iter-fact 'product)
=>
720
```

Note a convenience I'll be using throughout the chapter: instead of specifying the operations as a`list` of `list` explicitly, I'm using the backquote operator with commas. If this syntax is unfamiliar, consult this section of the Scheme manual.

Exercise 5.3

Once again, I'll skip the data path drawings and will get right to the register-machine code that can be simulated and tested. First, let's see the version that assumes `good-enough?` and `improve` are available as primitives:

```
(define (good-enough? guess x)
  (< (abs (- (square guess) x)) 0.001))

(define (improve guess x)
  (average guess (/ x guess)))

(define newton-with-ops
  (make-machine
    '(x guess)
    `((good-enough? ,good-enough?) (improve ,improve))
    '(
      init
        (assign guess (const 1.0))
      sqrt-iter
        (test (op good-enough?) (reg guess) (reg x))
        (branch (label end-sqrt))
        (assign guess (op improve) (reg guess) (reg x))
        (goto (label sqrt-iter))
      end-sqrt
    )))
```

To actually make them work as primitives, I define them as Scheme functions and later pass them to the machine in the *operations list*. Note that I made both functions accept `x` explicitly as an argument in order to separate them from the main function.

Let's see how this works:

```
(set-register-contents! newton-with-ops 'x 2)
(start newton-with-ops)
(printf "~a~%" (get-register-contents newton-with-ops 'guess))
=>
1.4142156862745097
```

Good. Now, let's implement these operations in the register-machine language and incorporate them into the machine:

```
(define newton-full
  (make-machine
    '(x guess r1)
    `((< ,<) (abs ,abs) (- ,-) (/ ,/)
      (square ,square) (average ,average))
    '(
      init
        (assign guess (const 1.0))
      sqrt-iter
        ; good-enough?
        (assign r1 (op square) (reg guess))
        (assign r1 (op -) (reg r1) (reg x))
        (assign r1 (op abs) (reg r1))
        (test (op <) (reg r1) (const 0.001))
        (branch (label end-sqrt))
        ; not good-enough? then improve
        (assign r1 (op /) (reg x) (reg guess))
        (assign guess (op average) (reg guess) (reg r1))
        (goto (label sqrt-iter))
      end-sqrt
    )))

(set-register-contents! newton-full 'x 2)
(start newton-full)
(printf "~a~%" (get-register-contents newton-full 'guess))
=>
1.4142156862745097
```

I'm using `r1` as a temporary register[2], and the bulding blocks of `improve` and `good-enough?` as primitives – the operations `square`, `average` and `abs`, which could be further broken down to register-machine language, of course.

Recursive procedures

Maybe it's just me, but I found the recursive implementation of the factorial procedure[3] by the authors in section 5.1.4 hard to understand. The reason is that it's an *optimized* hand-coded implementation, and I'm used to thinking in terms of compilers translating higher-order languages (like C) into assembly. So, I've decided to see how I would first think to implement this recursive factorial.

So, if you share my confusion, and aren't afraid of a little asembly code, you'll find this detour interesting.

So, how is recursion traditionally implemented in assembly ? Not much differently from the way normal procedures are implemented, it appears. Calling a procedure is just jumping to the procedure's label in some sense. Many assembly languages have the operations `call` and `ret`.

`call` does the following:

1. Saves the address of the next instruction on the stack
2. Jumps to the procedure

`ret` does the following:

1. Pops the return address from the stack
2. Jumps to the return address

However, calling procedures isn't enough. To make them really useful, procedures have to be passed arguments and return values. For this, several conventions exist:

- Registers can be used for arguments and return values. While this is the most efficient method, it is not generally applicable, because sometimes the call-sequences of procedures are long (proc1 calls proc2, which calls proc3 etc.) and there's not enough registers. Moreover, recursive procedures make this approach completely infeasible.
- The stack can be used for arguments and return values. This is the less efficient, but a much more general approach that works in all cases. The stack's LIFO nature guarantees the consistency of data, and arbitrary call-trees can be implemented, including recursive ones.
- Mixed approach: for example, arguments on stack, return value in a register. This is actually used by C compilers (recall that C can return only a single value…), unless a large object like a `struct` is returned.

Another important thing to keep in mind is that a well-behaving procedure must not ruin the data of its caller, and hence must leave all the registers intact. Usually, upon entry a procedure saves all the registers it's going to use on the stack, and before returning restores their value, thus preserving transparency.

So, let's use this approach to implement the recursive factorial procedure in our register-machine language:

- Since we have no `call` instruction, we'll explicitly push on the stack the label where we want to go after the procedure returns, and then jump to the procedure.
- Since we have no `ret` instruction, we'll expicitly gather the return address from the stack and jump to it
- We'll pass arguments (only `n` in our case) on the stack, and the return value in a register callec`retval`.
- The factorial procedure will save all the registers it uses on the stack when it starts and restore them when it ends.
- There's a minor difficuly in our language, presented by the fact that the access to the stack is only to and from its top. In x86 assembly, there's a comfortable addressing mode that allows us to read values from the stack without popping it[4], so much less pushing and popping is required.

Here's the code:

```
(define factorial-rec
  (make-machine
    '(n temp retval retaddr)
    `((= ,=) (+ ,+) (- ,-) (* ,*) (printf ,printf))
    '(
        (goto (label machine-start))

        ;;; procedure fact
      fact
        (restore retaddr)       ; return address
        (restore temp)          ; argument
        (save n)                ; save caller's n and retaddr
        (save retaddr)
        (assign n (reg temp))   ; working on n
        (test (op =) (reg n) (const 1))
        (branch (label fact-base))
        (assign temp (op -) (reg n) (const 1))
        ; prepare for the recursive call:
        ;  push the argument and return value on stack
        (save temp)
        (assign retaddr (label fact-after-rec-return))
        (save retaddr)
        (goto (label fact))     ; the recursive call
      fact-after-rec-return
        (assign retval (op *) (reg retval) (reg n))
        (goto (label fact-end))

      fact-base
        (assign retval (const 1))

      fact-end
        ; restore the caller's registers we've saved
        ;
        (restore retaddr)
        (restore n)
        (goto (reg retaddr))    ; return to caller
        ;;; end procedure fact

      machine-start
        ; to call fact, push n and a return address on stack
        ; and jump to fact
        (save n)
        (assign retaddr (label machine-end))
        (save retaddr)
        (goto (label fact))

      machine-end
    )))
```

Although it's considerably longer than the authors' version, I find it easier to understand, because it has a very standard structure.

In the beginning, we jump to the `machine-start` label which calls the factorial procedure. Note the call: first `n` is pushed, then the return address. This is going to be the calling convention throughout the code. The procedure `fact` itself is marked with a pair of comments.

`fact` does some intensive stack work in the beginning. First, it fetches the return address into `retaddr` and the argument into `temp`. Then, it saves the return address back again, together with `n` because it's going to use these registers in its work. If these steps seem unnecessary, think again. The procedure must have its argument and return value, and they're on the stack. The only way to get them is pop them from the stack. But then, as I said, in order to be transparent a procedure must save all the registers it's going to use and restore them before it returns.

Next it tests for `n = 0`, and jumps to `fact-base` if it is. If it isn't, it computes `n - 1` and calls itself recursively (note that it's done in exactly the same way as the original call from `machine-start`). `fact-end` is a common return point that restores the registers the procedure corrupted and jumps to the return address.

Now if you look at the authors' version, you'll see all the same building blocks, but some of the things were optimized away because the procedure is written not in a general way, but fine-tuned to the factorial problem. While for real machine language programs this is the better way to go, I think that for educational purposes my approach would be better.

Exercise 5.4

This is so similar to the factorial procedure that I don't think implementing it would teach us anything new. The only real difference is an additional argument `b`. But note that it's not changed throughout the procedure, so it can just be left in a register of its own, making the rest of the code even more similar to factorial.

Exercise 5.5

I'll simulate my implementation of `factorial` with the input `n = 2`. The stack is shown with its top on the right hand side. I.e. pushing 1, then 5, then 7 into an empty stack results in:

```
1 5 7
```

OK, so we're in `machine-start` calling `fact` with `n = 2`:

```
stack: 2 (label machine-end)
goto (label fact)
.
restoring retaddr and n
stack:
saving n and retaddr:
stack: 2 (label machine-end)
n != 1, hence a recursive call:
pushing new argument and return address on stack
stack: 2 (label machine-end) 1 (label fact-after-rec-return)
goto (label fact)
.
restoring retaddr and n
stack: 2 (label machine-end)
saving n and retaddr:
stack: 2 (label machine-end) 1 (label fact-after-rec-return)
n = 1, hence goto (label fact-base)
.
retval <- 1
restoring retaddr and n
stack: 2 (label machine-end)
goto retaddr, which is (label fact-after-rec-return)
.
retval <- 1 * 2
goto (label fact-end)
.
restoring retaddr and n
stack:
goto retaddr, which is (label machine-end)
.
at (label machine-end)
retval = 2
```

Exercise 5.6

In `afterfib-n-1`, `(restore continue)` and `(save continue)` can be safely removed, leaving us with:

```
(controller
   (assign continue (label fib-done))
 fib-loop
   (test (op <) (reg n) (const 2))
   (branch (label immediate-answer))
   ;; set up to compute Fib(n - 1)
   (save continue)
   (assign continue (label afterfib-n-1))
   (save n)                           ; save old value of n
   (assign n (op -) (reg n) (const 1)); clobber n to n - 1
   (goto (label fib-loop))            ; perform recursive call
 afterfib-n-1                         ; upon return, val contains Fib(n - 1)
   (restore n)
   ;; >> was: (restore continue)
   ;; set up to compute Fib(n - 2)
   (assign n (op -) (reg n) (const 2))
   ;; >> was: (save continue)
   (assign continue (label afterfib-n-2))
   (save val)                         ; save Fib(n - 1)
   (goto (label fib-loop))
 afterfib-n-2                         ; upon return, val contains Fib(n - 2)
   (assign n (reg val))               ; n now contains Fib(n - 2)
   (restore val)                      ; val now contains Fib(n - 1)
   (restore continue)
   (assign val                        ;  Fib(n - 1) +  Fib(n - 2)
           (op +) (reg val) (reg n))
   (goto (reg continue))              ; return to caller, answer is in val
 immediate-answer
   (assign val (reg n))               ; base case:  Fib(n) = n
   (goto (reg continue))
```

Why is this safe ?

What it does is take out `continue` from the stack and place it back again, so the stack isn't changed by these two instructions. But perhaps the value popped into the `continue` register is used ?

It is not, because on the next line, the `continue` register is assigned another value.

[1] I've decided to switch to Scheme because the register machine simulator is written in the *dispatch style*, which is much less cumbersome in Scheme than in Common Lisp.

[2] And following the convention of real machines where the *general purpose registers* which are used for temporary computations are usually called r1, r2, r3, etc.

[3] `factorial` is a notorious example because implementing it recursively is very inefficient compared to the simple interative implementation. However, it serves as a good example of implementing recursion, because it contains, in a very simplified way, everything one needs to know about recursive functions.

[4] In x86 there's a *base pointer* register `bp` which allows to access arguments, for example `add ax, 4[bp]` adds to the register `ax` the argument that was pushed last on the stack (before the return address).

For comments, please send me an email.

Back to top