SICP section 5.3

March 07, 2008 at 16:54 Tags SICP

I liked the way the authors used vectors to simply implement list structures. While cumbersome, the`car`, `cdr` and `cons` they've defined actually work very well. Here's a simple example that builds a cons cell and prints it out:

```
(define fib
  (make-machine
    '(n x y a b c val n1 the-cars the-cdrs free)
    `((= ,=) (+ ,+) (printf ,printf)
      (vector-ref ,vector-ref)
      (vector-set! ,vector-set!)
      (make-vector ,make-vector))
    '(
      ; create the-cars and the-cdrs vectors
      (assign the-cars (op make-vector) (const 100))
      (assign the-cdrs (op make-vector) (const 100))

      ; init the free pointer
      (assign free (const 0))

      ; n <- (cons x y)
      (perform (op vector-set!) (reg the-cars) (reg free) (reg x))
      (perform (op vector-set!) (reg the-cdrs) (reg free) (reg y))
      (assign n (reg free))
      (assign free (op +) (reg free) (const 1))

      ; print (car x)
      (assign val (op vector-ref) (reg the-cars) (reg n))
      (perform (op printf) (const "~a~%") (reg val))

      ; print (cdr y)
      (assign val (op vector-ref) (reg the-cdrs) (reg n))
      (perform (op printf) (const "~a~%") (reg val))
    )))

(set-register-contents! fib 'x 100)
(set-register-contents! fib 'y 222)
(start fib)

=>

100
222
```

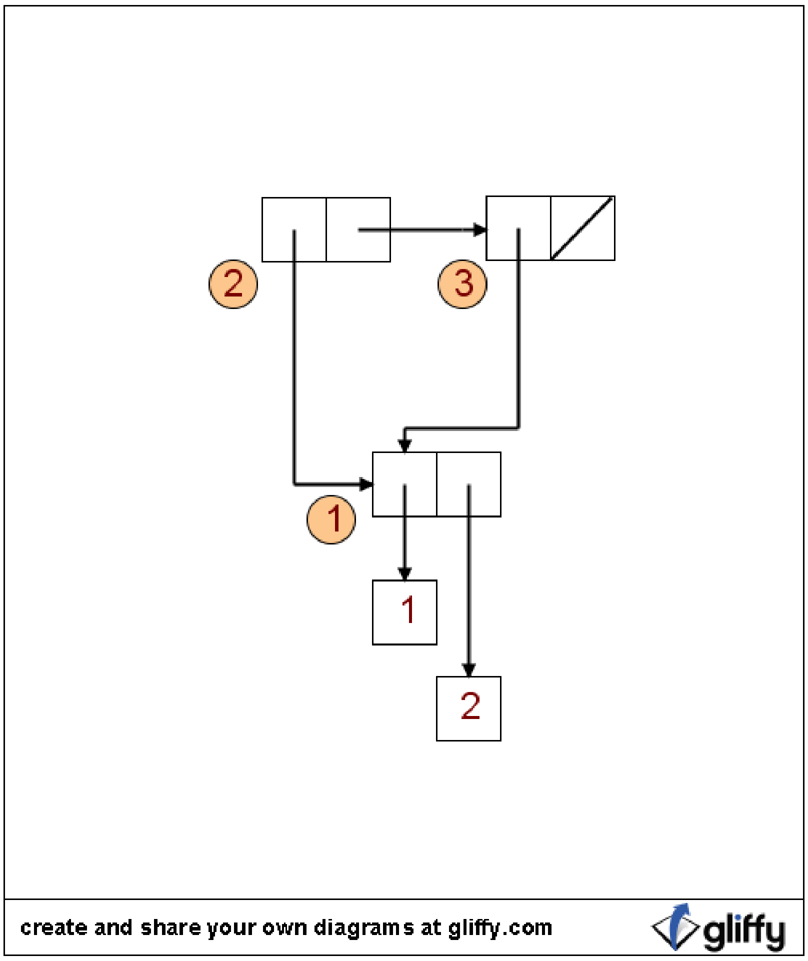
Unfortunately they don't go on implementing this system, and leave the*typed pointers* out. Hence, predicates such as `pair?` and `null?` aren't available.

Exercise 5.20

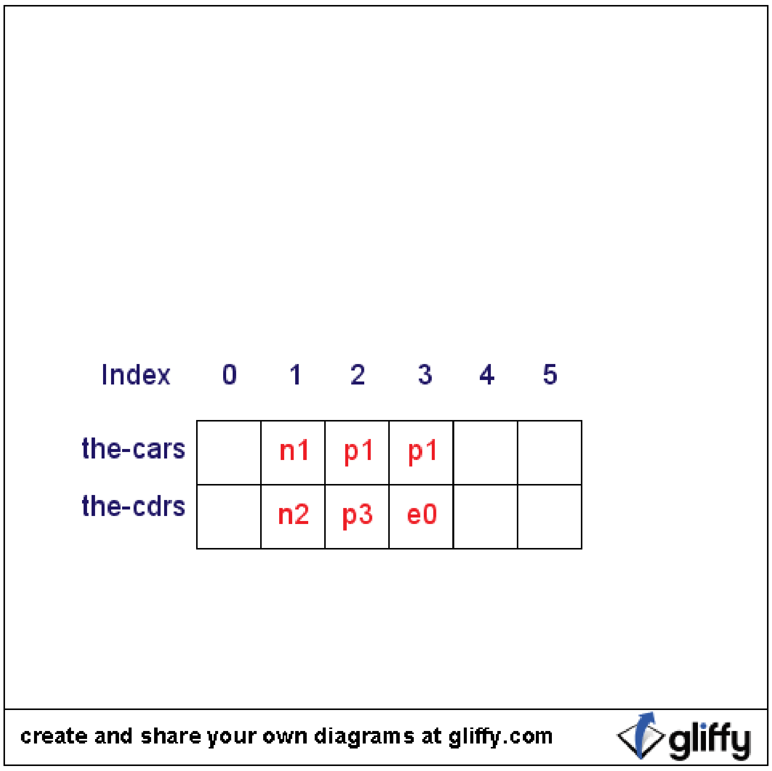
The box and pointer diagram created by:

```
(define x (cons 1 2))
(define y (list x x))
```

Is:



The numbers in orange circles are the cons-cell numbers that are referenced in the memory-vector representation, which is:



Index	0	1	2	3	4	5
the-cars		n1	p1	p1		
the-cdrs		n2	p3	e0		

create and share your own diagrams at gliffy.com
gliffy

`free` will contain `p4` after these assignments, since the cell at index 4 is the next unallocated memory slot.

Exercise 5.21

a.

The weirdest thing has just happened. I banged in the machine code for this computation and it worked the first time. No debugging, no tracing, nada. And this is a double-recursive function we're talking about ! I attribute this success to the structured way to write recursive procedures I've spoken about here. Although that was for a single recursion, double recursion isn't really more difficult if you follow the same guidelines. So here is the code, note how well it adheres to the rules I've previously outlined:

```
(define count-leaves-rec
  (make-machine
    '(tree n retval temp retaddr)
    `((= ,=) (+ ,+) (car ,car) (cdr ,cdr)
      (not ,not) (null? ,null?) (pair? ,pair?))
    '(
        (goto (label machine-start))

      ;;; procedure count-leaves-rec
      count-leaves-rec
        (restore retaddr)       ; return address
        (restore temp)          ; argument
        (save tree)             ; save caller's regs
        (save n)
        (save retaddr)
        (assign tree (reg temp)) ; working on 'tree'
        (test (op null?) (reg tree))
        (branch (label count-leaves-null))
        (test (op pair?) (reg tree))
        (branch (label count-leaves-pair))
        (goto (label count-leaves-atom))

      count-leaves-pair
        ; First recursive call, push arguments &
        ; return address, and execute call.
        ;
        (assign temp (op car) (reg tree))
        (save temp)
        (assign retaddr (label after-first-return))
        (save retaddr)
        (goto (label count-leaves-rec))

      after-first-return
        (assign n (reg retval))

        ; Second recursive call
        ;
        (assign temp (op cdr) (reg tree))
        (save temp)
        (assign retaddr (label after-second-return))
        (save retaddr)
        (goto (label count-leaves-rec))

      after-second-return
        (assign retval (op +) (reg n) (reg retval))
        (goto (label count-leaves-end))

      count-leaves-null
        (assign retval (const 0))
        (goto (label count-leaves-end))

      count-leaves-atom
        (assign retval (const 1))
        (goto (label count-leaves-end))

      count-leaves-end
        (restore retaddr)       ; restore caller's regs
        (restore n)
        (restore tree)
        (goto (reg retaddr))    ; return to caller
      ;;; end procedure count-leaves-rec
      machine-start
        (save tree)
        (assign retaddr (label machine-end))
        (save retaddr)
        (goto (label count-leaves-rec))

      machine-end
      )))
```

A small test shows that this works:

```
(set-register-contents! count-leaves-rec
  'tree '(1 (3 4) 5 (6 (7 3) 9)))
(set-register-contents! count-leaves-rec 'retval 0)

(start count-leaves-rec)

(printf ":~a~%" (get-register-contents count-leaves-rec 'retval))

=>

:8
```

b.

In this version of `count-leaves`, there are also two recursive calls, but one of them is*tail-recursive* and can be replaced by a loop:

```
(define count-leaves
  (make-machine
    '(tree n retval temp1 temp2 retaddr)
    `((= ,=) (+ ,+) (car ,car) (cdr ,cdr)
      (not ,not) (null? ,null?) (pair? ,pair?))
    '(
        (goto (label machine-start))

      ;;; procedure count-leaves-iter
      count-leaves-iter
        (restore retaddr)       ; return address
        (restore temp1)         ; argument 'tree'
        (restore temp2)         ; argument 'n'
        (save tree)             ; save caller's regs
        (save n)
        (save retaddr)
        (assign tree (reg temp1))
        (assign n (reg temp2))

      count-leaves-iter-loop
        (test (op null?) (reg tree))
        (branch (label count-leaves-null))
        (test (op pair?) (reg tree))
        (branch (label count-leaves-pair))
        (goto (label count-leaves-atom))

      count-leaves-pair
        ; Recursive call, push arguments &
        ; return address, and execute call.
        ;
        (save n)
        (assign temp1 (op car) (reg tree))
        (save temp1)
        (assign retaddr (label after-recursive-return))
        (save retaddr)
        (goto (label count-leaves-iter))

      after-recursive-return
        (assign n (reg retval))
        (assign tree (op cdr) (reg tree))
        (goto (label count-leaves-iter-loop))

      count-leaves-null
        (assign retval (reg n))
        (goto (label count-leaves-end))

      count-leaves-atom
        (assign retval (op +) (reg n) (const 1))
        (goto (label count-leaves-end))

      count-leaves-end
        (restore retaddr)       ; restore caller's regs
        (restore n)
        (restore tree)
        (goto (reg retaddr))    ; return to caller
      ;;; end procedure count-leaves-iter

      machine-start
        (assign n (const 0))
        (save n)
        (save tree)
        (assign retaddr (label machine-end))
        (save retaddr)
        (goto (label count-leaves-iter))

      machine-end
      )))
```

Exercise 5.22

Here is `append`:

```
(define append-rec
  (make-machine
    '(x y retval temp1 temp2 retaddr)
    `((car ,car) (cdr ,cdr) (cons ,cons)
      (not ,not) (null? ,null?) (pair? ,pair?))
    '(
        (goto (label machine-start))

      ;;; procedure append-rec
      append-rec
        (restore retaddr)       ; return address
        (restore temp2)         ; argument 'y'
        (restore temp1)         ; argument 'x'
        (save x)                ; save caller's regs
        (save y)
        (save retaddr)
        (assign x (reg temp1))
        (assign y (reg temp2))

        (test (op null?) (reg x))
        (branch (label append-null-x))
        (goto (label append-not-null-x))

      append-null-x
        (assign retval (reg y))
        (goto (label append-end))

      append-not-null-x
        ; Execute recursive call
        (assign temp1 (op cdr) (reg x))
        (save temp1)
        (save y)
        (assign retaddr (label after-recursive-return))
        (save retaddr)
        (goto (label append-rec))

      after-recursive-return
        (assign temp1 (op car) (reg x))
        (assign retval (op cons) (reg temp1) (reg retval))

      append-end
        (restore retaddr)       ; restore caller's regs
        (restore y)
        (restore x)
        (goto (reg retaddr))    ; return to caller
      ;;; end procedure append-end

      machine-start
        (save x)
        (save y)
        (assign retaddr (label machine-end))
        (save retaddr)
        (goto (label append-rec))

      machine-end
      )))
```

And a test:

```
(set-register-contents! append-rec 'x '(1 2 3))
(set-register-contents! append-rec 'y '(8 9))
(set-register-contents! append-rec 'retval '())

(start append-rec)
(printf ":~a~%" (get-register-contents append-rec 'retval))
=>
:(1 2 3 8 9)
```

And this is `append!`:

```scheme
(define append!
  (make-machine
    '(x y retval temp1 temp2 retaddr)
    `((car ,car) (cdr ,cdr) (cons ,cons)
      (set-cdr! ,set-cdr!)
      (not ,not) (null? ,null?) (pair? ,pair?))
    '(
        (goto (label machine-start))

      ;;; procedure append!
      append!
        (restore retaddr)
        (restore y)
        (restore x)

        ; Prepare arguments and call last-pair
        (save x)
        (assign temp2 (label after-last-pair))
        (save temp2)
        (goto (label last-pair))

      after-last-pair
        (assign temp2 (reg retval))
        (perform (op set-cdr!) (reg temp2) (reg y))
        (assign retval (reg x))
        (goto (reg retaddr))
      ;;; end procedure append!

      ;;; procedure last-pair
      last-pair
        (restore temp2)         ; argument 'retaddr'
        (restore temp1)         ; argument 'x'
        (save x)
        (save retaddr)
        (assign x (reg temp1))
        (assign retaddr (reg temp2))

      last-pair-loop
        (assign temp1 (op cdr) (reg x))
        (test (op null?) (reg temp1))
        (branch (label last-pair-null))
        (assign x (op cdr) (reg x))
        (goto (label last-pair-loop))

      last-pair-null
        (assign retval (reg x))
        (assign temp2 (reg retaddr))
        (restore retaddr)
        (restore x)
        (goto (reg temp2))    ; return
      ;;; end procedure last-pair

      machine-start
        (save x)
        (save y)
        (assign retaddr (label machine-end))
        (save retaddr)
        (goto (label append!))

      machine-end
      )))
```

And a test:

```
(set-register-contents! append! 'x '(1 2 3))
(set-register-contents! append! 'y '(8 9))

(start append!)
(printf ":~a~%" (get-register-contents append! 'x))
=>
:(1 2 3 8 9)
```

A word on programming style

I wrote about this before, but I want to emphasize this point again: the programming style I'm using here for writing the register machine code is not optimized for speed or minimal space. Rather, its main aim is to be systematic and easily understandable. It's definitely possible to optimize away a few statements in each machine I wrote in this section, especially in the handling of stack data, but IMHO that would hurt the clarity of this code. And clarity is my preference, at least here.

For comments, please send me an email.

© 2003-2025 Eli Bendersky

Back to top