Representing the Stack as a Vector

We have been representing the stack with Scheme lists. Lists are a great high-level representation for stacks, but they aren’t how machines actually represent the stack. In a computer, we just have memory to work with, and memory is really just an array. In Scheme, we call an array a *vector*, and we will use *vectors* to represent the stack.

We use \texttt{(make-vector \emph{n})} to make a vector of length \emph{n}. To set position \emph{pos} of vector \emph{v} to the value \emph{val}, we use \texttt{(vector-set! v pos val)}. To access an element of vector \emph{v}, we use \texttt{(vector-ref v pos)}.

We define the stack, registers, and structs as follows:

\begin{verbatim}
(define the-stack (make-vector 100))

(define =sp= 0)
(define =rl= 'dummy)
(define =tmp0= 'dummy)

(define-struct stk/mt ()
(define-struct stk/cons (fst))
\end{verbatim}

The \emph{=sp=} register is our *stack pointer*. The stack pointer represents one location past the top of the stack.\footnote{We could have easily declared that the stack pointer represent the top of the stack. The only requirement is that we use a convention *consistently*.}

The only change to our code for \emph{fil-pos} is that we no longer need to pass the stack around as an argument because the stack is now a global structure:

\begin{verbatim}
(define (fil-pos/stk)
  (cond
   [(empty? =rl=) (Pop empty)]
   [(cons? =rl=)]
\end{verbatim}
(cond
  [(positive? (first =r1=))
   (begin
     (set! =tmp0= (first =r1=))
     (set! =r1= (rest =r1=))
     (Push (make-stk/cons =tmp0=))
     (fil-pos/stk))]
  [else
   (begin
     (set! =r1= (rest =r1=))
     (fil-pos/stk))])
)

We now update our Push and Pop functions in a straightforward manner:

(define (Push frame)
  (begin
    (vector-set! the-stack =sp= frame)
    (set! =sp= (add1 =sp=))))

(define (Pop v)
  (let ([1st (vector-ref the-stack (sub1 =sp=))])
    (begin
      (set! =sp= (sub1 =sp=))
      (cond
       [(stk/mt? 1st) v]
       [(stk/cons? 1st)
        (Pop (cons (stk/cons-fst 1st) v))])))
)

Finally, the function filp-os becomes:

(define (fil-pos l)
  (begin
    (set! =sp= 0)
    (Push (make-stk/mt))
    (set! =r1= l)
    (fil-pos/stk)))

Making the Heap Explicit

In Scheme, when we call cons, a cons cell is created on the heap. In order to hand compile our code to assembly, we will need to not rely on Scheme lists, and instead make use of the computer’s heap. We will use a vector to represent the heap, and we will store our own representation of cons cells on this heap representation.

We define the heap as follows:

(define the-heap (make-vector 1000))
(define heap-ptr 0)
We need a way of allocating a cons cell on the heap. The function `nempty/alloc` allocates an empty list at the next free spot on the heap and returns its location:

```
;; nempty/alloc: → location
(define (nempty/alloc)
  (begin
    (vector-set! the-heap heap-ptr 'empty)
    (set! heap-ptr (+ heap-ptr 1))
    (− heap-ptr 1)))
```

```
;; ncons/alloc: num × location → location
(define (ncons/alloc fst rst)
  (begin
    (vector-set! the-heap heap-ptr 'cons)
    (vector-set! the-heap (add1 heap-ptr) fst)
    (vector-set! the-heap (+ 2 heap-ptr) rst)
    (set! heap-ptr (+ heap-ptr 3))
    (− heap-ptr 3)))
```

The selectors and predicates for our list implementation are straightforward:

```
(define (nfirst/alloc addr)
  (vector-ref the-heap (add1 addr)))
```

```
(define (nrest/alloc addr)
  (vector-ref the-heap (+ 2 addr)))
```

```
(define (ncons?/alloc addr)
  (eq? (vector-ref the-heap addr) 'cons))
```

```
(define (nempty?/alloc addr)
  (eq? (vector-ref the-heap addr) 'empty))
```

Now we simply have to update the `fil-pos/stk` function to use our new list constructors, selectors, and predicates:

```
(define (fil-pos/stk)
  (cond
    [(nempty?/alloc =r1=) (Pop (nempty/alloc))]
    [(ncons?/alloc =r1=)]
    [(positive? (nfirst/alloc =r1=))
      (begin
        (set! tmp0= (nfirst/alloc =r1=))
        (set! =r1= (nrest/alloc =r1=))
        (Push (make-stk/cons =tmp0=))
        (fil-pos/stk))]
    [else
      (begin
```

3
The last line of \textit{Pop} function needs to be updated to use the new list constructor:

\begin{verbatim}
(define (Pop v)
  (let ((1st (vector-ref the-stack (sub1 =sp=))))
    (begin
      (set! =sp= (sub1 =sp=))
      (cond
        [(stk/mt? 1st) v]
        [(stk/cons? 1st)
         (Pop (ncons/alloc (stk/cons-fst 1st) v))])))
)

The input to \textit{fil-pos} now must be on the heap. The following code allocates a test list, and then sets the desired registers and stack state and then computes on that test list. This is clearly not a generic \textit{fil-pos} function as written (since it has the input built-in) but it illustrates the idea and is simple: \footnote{A more complicated approach would have been to describe how to get from the user input into appropriate list constructor calls—but this is not an interesting problem as far as this class is concerned.}

\begin{verbatim}
(define (fil-pos)
  (begin
    (let ((mem (ncons/alloc -5
             (ncons/alloc 4
              (ncons/alloc 17
                (nempty/alloc))))))
      (begin
        (set! =sp= 0)
        (Push (make-stk/mt))
        (set! =rl= mem)
        (printf "answer: ~s\nheap: ~s\n" (filpos/stk) the-heap))))

\end{verbatim}