6.821 Programming Languages Handout
Fall 2002

MASSACHVSETTS INSTITVTE OF TECHNOLOGY
Department of Electrical Engineering and Compvter Science

Problem Set 8

Problem 1: Concurrency

Dr. Brian Storm is working late one night when he realizes that most computers spend far more time waiting for commands from their users than actually carrying out the commands. Quickly, he decides to harness the vast power of these idle cycles by creating an implementation of PostFix that can execute in a distributed fashion.

In Dr. Storm's new language, a number of PostFix *processes* can execute simultaneously, each on a different computer. Two processes running on different machines can cooperate by exchanging messages over a network connecting all of the computers. Dr. Storm decides to call his new language POW! — PostFix Over Wires!.

Having taken 6.821 while a student at MIT, Dr. Storm knows that he should test his ideas out first by creating an operational semantics for his new language. He begins by deciding that a process will be represented by a triple consisting of a unique process identifier (pid), the commands of the process, and the stack of the process. Thus a configuration becomes a *set* of processes:

$$\begin{array}{rcccl} \pi & \in & \text{Pid} & = & \text{IntLit} \\ p & \in & \text{Process} & = & \text{Pid} \times \text{Commands} \times \text{Stack} \\ c & \in & \mathcal{C} & = & \{X \mid X \text{ is a set of processes with unique pids}\} \end{array}$$

Dr. Storm has no trouble adapting the rewrite rules for the usual Postfix commands to his new semantics. For example:

$$\begin{array}{rcl} c \cup \{\langle \pi, N.Q, S\rangle\} & \Rightarrow & c \cup \{\langle \pi, Q, N.S\rangle\} \\ c \cup \{\langle \pi, \texttt{dup}.Q, V.S\rangle\} & \Rightarrow & c \cup \{\langle \pi, Q, V.V.S\rangle\} \\ c \cup \{\langle \pi, \texttt{pop}.Q, V.S\rangle\} & \Rightarrow & c \cup \{\langle \pi, Q, S\rangle\} \\ c \cup \{\langle \pi, \texttt{swap}.Q, V_1.V_2.S\rangle\} & \Rightarrow & c \cup \{\langle \pi, Q, V_2.V_1.S\rangle\} \end{array}$$

Having worked through the night constructing the foundation of POW!, Dr. Storm leaves for home and some badly needed sleep. But before he leaves, he lets you in on his plans for POW! and asks you to fill in the details.

a. Dr. Storm decides that a POW! program will start off executing a single process with pid 0, which will "spawn" new processes as necessary. The value of the POW! program will be the value at the top of the stack of process 0 when it runs out of commands.

   Define the input function, the set of final configurations, and the output function for POW!. Beware that there is a certain amount of ambiguity in this question regarding final states and errors. In your solution you should point out these ambiguities and state how you resolve them. For instance, you should give a careful description of how you handle errors, and a precise definition of what you consider to be a "stuck state", if your description uses them.

b. Dr. Storm adds the following concurrency primitives to the grammar of Postfix:

$$C ::= \texttt{spawn} \mid \texttt{pid} \mid \texttt{channel} \mid \texttt{send} \mid \texttt{receive} \mid \ldots$$

   Here is an informal description of the behavior of the new commands:

- `spawn`: If the stack of the parent process executing the `spawn` command is of the form $Q.S$, then a new process is created, with a fresh pid, command sequence Q, and stack S. The stack of parent becomes S; both the parent and the child have the same stack after `spawn`.
- `pid`: The process executing the `pid` command pushes its process number onto its stack. (Although process ids are represented as IntLits, you are not allowed to add process ids to IntLits. All functions on IntLits have been changed to reflect this.)
- `channel`: The process executing the `channel` command pushes a fresh channel onto its stack. A channel is something over which messages can be exchanged (see `send` and `receive`). Dr. Storm has already added a domain equation for channels and extended the value domain to include channels and process ids (to make them first-class, of course... yeah! first-class!):

  $$\begin{array}{rcccl} W & \in & \text{Channel} & = & \text{IntLit} \\ V & \in & \text{Value} & = & \text{Channel} + \text{Pid} + \ldots \end{array}$$

  (Although channels are represented as IntLits, you are not allowed to add channels to IntLits or send messages over IntLits. As with Pids, functions on IntLits have been modified to reflect this restriction.)
- `send` and `receive`: These two commands are used to invoke synchronous message passing over a channel. In synchronous message passing, two processes must *rendezvous* in order to exchange a message: the message exchange occurs in one atomic step, and if a process tries to send when no other process is ready to receive, the sending process is blocked — it cannot proceed with the send and must wait. Similarly, a process that tries to receive when no process is ready to send cannot proceed.

  A process is *ready to send value V over channel W* if its first command is `send`, the top value on its stack is V, and the next-to-top value on its stack is W.

  A process is *ready to receive over channel W* if its first command is `receive` and the top value on its stack is W.

  If p_s is ready to send value V over channel W and p_r is ready to receive over channel W then the processes can exchange the message in a single step: V is popped from the stack of p_s and pushed onto the stack of p_r. The channel W is *not* popped off either stack. Note that if more than one process is ready to send or receive on the same channel, pair selection is non-deterministic. That is to say any sender/receiver pair may be selected from the set of candidate processes attempting to communicate on the channel.

Provide the rewrite rules for `spawn`, `pid`, `channel`, `send`, and `receive`.

Problem 2: Memory Management

Ben Bitdiddle has been called in by the Analog Equipment Corporation to consult on a difficult memory management problem. Analog uses Balsa, a programming language that stack allocates all continuations and environments. Since Balsa does not support a garbage collector, heap storage must be explicitly managed by programmers via calls to the procedures `malloc` and `free`:

E ::= `(malloc` E`)` | `(free` E`)` | ...

Here is the informal description of `malloc` and `free` from the Balsa ANSI Standard:

- `(malloc` E`)`: If the value of E is a positive integer n, returns a location for a block of storage that is $n+1$ words long. The first word of the returned block is a size header; the other n words are uninitialized.
- `(free` E`)`: If the value of E is a location, frees the storage at that location and returns an unspecified value.

Analog is having problems with a very large Balsa application (called "The Titanic" by the development staff) that eventually always either mysteriously crashes or runs out of heap space. Ben suspects that the programmers who wrote the application are not properly deallocating storage.

In order to debug Analog's problem Ben decides to write a standard stop-and-copy garbage collector for Balsa. He modifies `malloc` and `free` to keep track of the total amount of "busy" storage — `malloc` increments a `*busy*` counter with the number of words in the object it creates and `free` decrements the `*busy*` counter by the number of words in the object it frees. In Ben's system `free` does not actually free any storage. Instead, when storage is exhausted, the garbage collector runs and copies live storage (storage that is reachable from a root set) from old space into new space.

a. Let *live* be the number of words copied during a garbage collection and *busy* be the value of the `*busy*` counter at the time of the garbage collection. In each of the following situations encountered while executing the program in Ben's system *with* garbage collection, describe the implications for executing the original Balsa program *without* garbage collection:

   (i) $live < busy$

   (ii) $live > busy$

   (iii) $live = busy$

b. A *dangling reference* is a pointer to a freed block of memory that is still reachable from the root set. How could Ben modify his garbage collector to detect dangling references?

c. Ben tries his garbage collector out on another program from Analog Equipment Corporation. The development staff at AEC have used program verification techniques to prove that this program does not have a storage leak and in fact it runs just fine without garbage collection. However, when Ben runs the program with garbage collection, the garbage collector runs out of memory. What is going on? You should assume that the program verificiation techniques were valid and used correctly.

Problem 3: `funrec`

Chloe Jour has been experimenting with the TORTOISE compiler. She notes that the representation of procedures in TORTOISE is not very efficient, particularly for recursive procedures. Consider, for example, the following code:

```
(letrec ((even? (lambda (a) (if (= 0 a)
                                #t
                                (odd? (- a 1)))))
         (odd?  (lambda (b) (if (= 0 b)
                                #f
                                (even? (- b 1))))))
  E_body)
```

The procedures `even?` and `odd?` are represented by the structures depicted in Figure 1. In the figure, `even?` and `odd?` are pointers to closure blocks. Each closure block has four data slots: the first holds a pointer to the code of the procedure, and the remainder point to the free variables of the procedure (i.e., they constitute its environment). The cells in the picture "tie the knot" of the mutual recursion; they are introduced by a combination of the `letrec` desugaring and assignment conversion.

Chloe observes that it is possible to represent a set of mutually recursive procedure by a single closure that combines the code pointers and free variables of all the procedures. For example, she condenses the closures from Figure 1 into the single closure depicted in Figure 2. The merged data structure is a block with four data slots: the first data slot holds a pointer to the code for `even?`; the second holds a pointer to the code for `odd?`, and the remainder point to the free variables contained in the bodies of both procedures. The new data structure represents in 5 words of memory what used to take 14 words — a considerable space savings. Chloe's representation saves time as well; since it has no cells, the time overhead of indirecting through the cells is removed.

```
          +------------------------------------------------------+
          |                                                      |
          |      +---------+                                     |
even? ----+----->| closure |                                     |
                 -----------                                     |
                 |    *----|----> code for even?                 |
                 -----------                       +------+      |
                 |    *----|---------------------->| cell |      |
                 -----------                       +------+      |
                 |    *----|----> closure for =    |   *  |      |
                 -----------                       +---|--+      |
                 |    *----|----> closure for -        |         |
                 +---------+                           |         |
                                                       |         |
          +--------------------------------------------+         |
          |                                                      |
          |      +---------+                                     |
odd? -----+----->| closure |                                     |
                 -----------                                     |
                 |    *----|----> code for odd?                  |
                 -----------                       +------+      |
                 |    *----|---------------------->| cell |      |
                 -----------                       +------+      |
                 |    *----|----> closure for =    |   *--|------+
                 -----------                       +------+
                 |    *----|----> closure for -
                 +---------+
```

Figure 1: Closures and environments for the `letrec` example. We assume that the globalizing phase wraps the program with bindings for = and - so they are free in the procedures.

```
            +---------+
even? ----> | closure |
            +---------+
odd? -----> |   *-----|--------> code for even?
            +---------+
            |   *-----|--------> code for odd?
            +---------+
            |   *-----|--------> closure for =
            +---------+
            |   *-----|--------> closure for -
            +---------+
```

Figure 2: Chloe's merged closure structure for the `even?/odd?` example.

In Chloe's representation, each mutually recursive procedure does not have its own closure block. Instead, a single closure block is shared among all the procedures. Different procedures are represented by pointers to different parts of the closure block. In the example, `even?` points to the header of the closure block (as usual), but `odd?` points to the first data word of the block. This trick allows the new procedure representation to be called in exactly the same way as the old representation: in both representations, a procedure is a pointer to a word in memory just before the word containing the code pointer for the procedure. A call does not need to know which sort of procedure it is applying; it only needs to extract the code pointer from the representation. In either case, it can access the code pointer by loading the word just past that pointed to by the procedure representation.

To experiment with this new representation, Chloe adds a new form, `funrec`, to SCHEME. `funrec` is a specialized form of `letrec`, restricted to defining recursive procedures.

A `funrec` expression has the general form:

```
(funrec ((I_0 (lambda (I_0-arg*) E_0))
         (I_1 (lambda (I_1-arg*) E_1))
           ⋮
         (I_n (lambda (I_n-arg*) E_n)))
  E_body)
```

Each I_i is a procedure being defined by the `funrec`, with arguments I_{i-arg}* and body E_i. Every procedure I_i can be used in every body E_j as well as in E_{body}. Since Chloe's representation for the procedures does not employ any cells, she declares that it is illegal to `set!` the variables I_0, ..., I_n bound by the `funrec`. Using her new `funrec` construct, Chloe expresses the `even?/odd?` example as:

```
(funrec ((even? (lambda (a) (if (= 0 a)
                                #t
                                (odd? (- a 1)))))
         (odd? (lambda (b)  (if (= 0 b)
                                #f
                                (even? (- b 1))))))
  E_body)
```

Chloe extends several phases of the TORTOISE compiler to handle `funrec`: the desugarer, globalizer, assignment converter, and CPS converter. She is just starting to modify the closure converter when her brother Supta reminds her that they have a dinner engagement. She asks you to extend this phase of compilation to handle `funrec`.

Before she leaves, she gives you the following information:

- The (unmodified) TORTOISE compiler can be found in the course directory in the subdirectory `code/book/ch15:compilation`. This subdirectory contains numerous files; see the `README` file for documentation. The compiler can be loaded into Scheme by loading the file `load.scm`.
- As part of her extensions, Chloe has written two new files that should be loaded into Scheme *after* the original compiler. These can be found in the course directory in the subdirectory `code/ps8`:
  - `funrec.scm` contains the modifications that allow the early phases of the compiler to handle `funrec`.
  - `closurize.scm` contains a new closure converter that Chloe was writing before she was called away. This file *supersedes* the `closure-convert.scm` file from the original TORTOISE implementation. For convenience, the contents of this file are presented in Figure 3. The `closurize` procedure converts every `lambda` and `funrec` into a flat closure. The converter is complete except for the definition of `closurize-funrec`, which you are responsible for writing.
- Chloe has extended the set of closure primops to allow pointers into the middle of closures. As before, `closure` creates a closure block and returns a pointer to the header word of the block. But Chloe has added a new primop, `closure-shift`, that returns a pointer into a given closure at a specified offset. Given pointer (word address) p into a closure and a word offset o, `(primop closure-shift` p o`)`

returns the pointer $p + o$. The offset may be positive, zero, or negative, but the resulting pointer is required to be within the closure block; it is an error if this condition is not met. Chloe has also extended the primitives `closure-ref` and `closure-set!` to handle negative offsets, again with the restriction that all references must lie within the closure block.

For example, consider the following sequence of evaluations:

```
(define c (%closure 3 5 7))

(%closure-ref c 0) ⟶ 3
(%closure-ref c 1) ⟶ 5
(%closure-ref c 2) ⟶ 7

(define c+2 (%closure-shift c 2))

(%closure-ref c+2 -2) ⟶ 3
(%closure-ref c+2 -1) ⟶ 5
(%closure-ref c+2 0) ⟶ 7
(%closure-ref c+2 1) ⟶ error

(define c+1 (%closure-shift c+2 -1))

(%closure-ref c+1 -1) ⟶ 3
(%closure-ref c+1 0) ⟶ 5
(%closure-ref c+1 1) ⟶ 7
```

- Chloe emphasizes that all SCHEME constructs other than `lambdas` within a `funrec` will be compiled as before. In particular the code generated for a call of a procedure defined via `funrec` is the same as that for any other call. In other words, the *caller*'s half of the calling convention is unchanged. However, the code generated for `lambdas` within a `funrec` needs to be different than that generated for non-`funrec` lambdas.
- Chloe suggests you use the following composition of compiler phases to test your design:

```
(define ->closures/no-cps
  (cascade initialize
           desugar
           globals/wrap
           assignment-convert
           closurize
           abbreviate
           pp))
```

  This includes the desugaring, globalizing, assignment conversion, and closure conversion phases, as well as some extra phases (`initialize`, `abbreviate`, and `pp`) that make the output easier to read. The `cascade` function composes the phases so that they are performed in a left to right order. The set of phases does not include CPS conversion so that the output is easier to read. (Of course, the compilation process should still work if the CPS phase is inserted.)

After going on for what seems like forever, Chloe finally leaves and you set about your task:

a. Implement the `closurize-funrec` procedure. Be sure to provide an English explanation of your design and a few test cases including the `even?/odd?` example.

   For this part of the question, you should ignore any garbage collection issues that arise.

   Here are some questions to keep in mind:

- For the TORTOISE representation shown in figure 1, how does the code for the body of the `odd?` procedure access the values of the variables named `even?` and `=`?
- For Chloe's merged structure in figure 2, how does the code for the body of the `odd?` procedure access the values of the variables named `even?` and `=`?

b. Chloe's representation for mutually recursive procedures can indeed save time and space. However, it introduces a few prickly problems:

   (i) The `procedure?` predicate no longer works. Explain why, and describe how to fix it.

   (ii) Garbage collection will no longer work properly given Chloe's representation. Explain why, and describe how to modify the garbage collector for TORTOISE to fix the problem.

```scheme
(define (closurize node)
  (cond ((application-node? node) (closurize-application node))
        ((lambda-node? node) (closurize-lambda node))
        ((funrec-node? node) (closurize-funrec node))
        (else (subnode-map closurize node))))

(define (closurize-application node)
  (apply make-ccall
         (closurize (call-rator node))
         (map closurize (call-rands node))))

(define (closurize-lambda node)
  (let ((formals (lambda-formals node))
        (body (lambda-body node))
        (frees (free-vars node))
        (closure-var (make-var (fresh-name 'closure))))
    `(PRIMOP CLOSURE
             (LAMBDA (,closure-var ,@formals)
               ,(rewrite (list->set frees)
                         ;; Ref-rewriting procedure
                         (lambda (var)
                           (make-primop 'closure-ref
                                        (list closure-var
                                              ;; Need 1+ to pass over code
                                              (1+ (position var frees)))))
                         ;; SET!-rewriting procedure
                         (lambda (var body)
                           (make-primop 'closure-set!
                                        (list closure-var
                                              ;; Need 1+ to pass over code
                                              (1+ (position var frees))
                                              body)))
                         (closurize body)))
             ,@frees)))
```

Figure 3: The contents of Chloe's file `closurize.scm`.