

Homework 4

COSE212, Fall 2019

Hakjoo Oh

Due: 11/25, 23:59

Problem 1 (100pts) Let us design and implement an imperative language, called B, which is a subset of the C programming language. The syntax of B is as follows:

$e \rightarrow$	<code>unit</code>	unit
	<code>x := e</code>	assignment
	<code>e ; e</code>	sequence
	<code>if e then e else e</code>	branch
	<code>while e do e</code>	while loop
	<code>write e</code>	output
	<code>let x := e in e</code>	variable binding
	<code>let proc f(x₁, x₂, ..., x_n) = e in e</code>	procedure binding
	<code>f(e₁, e₂, ..., e_n)</code>	call by value
	<code>f<x₁, x₂, ..., x_n></code>	call by reference
	<code>n</code>	integer
	<code>true</code> <code>false</code>	boolean
	<code>{}</code> <code>{x₁:=e₁, x₂:=e₂, ..., x_n:=e_n}</code>	record (i.e., struct)
	<code>e.x</code>	record lookup
	<code>e.x := e</code>	record assignment
	<code>x</code>	identifier
	<code>e + e</code> <code>e - e</code> <code>e * e</code> <code>e / e</code>	arithmetic operation
	<code>e < e</code> <code>e = e</code> <code>not e</code>	boolean operation

A program is an expression. Expressions include unit, assignments, sequences, conditional expressions (branch), while loops, read, write, let expressions, let expressions for procedure binding, procedure calls (by either call-by-value or call-by-reference), integers, boolean constants, records (i.e., structs), record lookup, record assignment, identifier, arithmetic expressions, and boolean expressions. Note that procedures may have multiple arguments. The language manipulates the following values:

x, y	\in	Id	identifier (variable)
l	\in	$Addr$	address (memory location)
n	\in	\mathbb{Z}	integer
b	\in	$\mathbb{B} = \{true, false\}$	
r	\in	$Record = Id \rightarrow Addr$	
v	\in	$Val = \mathbb{Z} + \mathbb{B} + \{\cdot\} + Record$	
σ	\in	$Env = Id \rightarrow Addr + Procedure$	
M	\in	$Mem = Addr \rightarrow Val$	
		$Procedure = (Id \times Id \times \dots) \times Expression \times Env$	

A record (i.e., struct) is defined as a (finite) function from identifiers to memory addresses. A value is either an integer, boolean value, unit value (\cdot), or a record. An environment maps identifiers to memory addresses or procedure values. A memory is a finite function from addresses to values. Note that we design B in a way that procedures are not stored in memory, which means that procedures are not first-class values in B. The semantics of the language is defined as follows (Below, we write $\sigma\{x \mapsto l\}$ and $M\{l \mapsto v\}$ for the environment σ and memory M extended with the new entries):

$$\begin{array}{c}
\text{TRUE} \frac{}{\sigma, M \vdash \mathbf{true} \Rightarrow true, M} \quad \text{FALSE} \frac{}{\sigma, M \vdash \mathbf{false} \Rightarrow false, M} \\
\\
\text{NUM} \frac{}{\sigma, M \vdash \mathbf{n} \Rightarrow n, M} \quad \text{UNIT} \frac{}{\sigma, M \vdash \mathbf{unit} \Rightarrow \cdot, M} \\
\\
\text{VAR} \frac{}{\sigma, M \vdash x \Rightarrow M(\sigma(x)), M} \quad \text{RECF} \frac{}{\sigma, M \vdash \{ \} \Rightarrow \cdot, M} \\
\\
\text{RECT} \frac{\begin{array}{c} \sigma, M \vdash e_1 \Rightarrow v_1, M_1 \\ \sigma, M_1 \vdash e_2 \Rightarrow v_2, M_2 \\ \vdots \\ \sigma, M_{n-1} \vdash e_n \Rightarrow v_n, M_n \end{array}}{\sigma, M \vdash \{x_1 := e_1, \dots, x_n := e_n\} \Rightarrow \{x_1 \mapsto l_1, \dots, x_n \mapsto l_n\}, M_n\{l_1 \mapsto v_1, \dots, l_n \mapsto v_n\}} \quad \forall i. l_i \notin \text{Dom}(M_n) \\
\\
\text{ADD} \frac{\sigma, M \vdash e_1 \Rightarrow n_1, M' \quad \sigma, M' \vdash e_2 \Rightarrow n_2, M''}{\sigma, M \vdash e_1 + e_2 \Rightarrow n_1 + n_2, M''} \\
\\
\text{SUB} \frac{\sigma, M \vdash e_1 \Rightarrow n_1, M' \quad \sigma, M' \vdash e_2 \Rightarrow n_2, M''}{\sigma, M \vdash e_1 - e_2 \Rightarrow n_1 - n_2, M''} \\
\\
\text{MUL} \frac{\sigma, M \vdash e_1 \Rightarrow n_1, M' \quad \sigma, M' \vdash e_2 \Rightarrow n_2, M''}{\sigma, M \vdash e_1 * e_2 \Rightarrow n_1 * n_2, M''} \\
\\
\text{DIV} \frac{\sigma, M \vdash e_1 \Rightarrow n_1, M' \quad \sigma, M' \vdash e_2 \Rightarrow n_2, M''}{\sigma, M \vdash e_1 / e_2 \Rightarrow n_1/n_2, M''}
\end{array}$$

$$\begin{array}{c}
\text{EQUALT} \frac{\sigma, M \vdash e_1 \Rightarrow v_1, M' \quad \sigma, M' \vdash e_2 \Rightarrow v_2, M''}{\sigma, M \vdash e_1 = e_2 \Rightarrow \mathbf{true}, M''} \quad \begin{array}{l} v_1 = v_2 = n \\ \vee v_1 = v_2 = b \\ \vee v_1 = v_2 = \cdot \end{array} \\
\\
\text{EQUALF} \frac{\sigma, M \vdash e_1 \Rightarrow v_1, M' \quad \sigma, M' \vdash e_2 \Rightarrow v_2, M''}{\sigma, M \vdash e_1 = e_2 \Rightarrow \mathbf{false}, M''} \quad \text{otherwise} \\
\\
\text{LESS} \frac{\sigma, M \vdash e_1 \Rightarrow n_1, M' \quad \sigma, M' \vdash e_2 \Rightarrow n_2, M''}{\sigma, M \vdash e_1 < e_2 \Rightarrow n_1 < n_2, M''} \\
\\
\text{NOT} \frac{\sigma, M \vdash e \Rightarrow b, M'}{\sigma, M \vdash \mathbf{not} \ e \Rightarrow \mathbf{not} \ b, M'} \\
\\
\text{ASSIGN} \frac{\sigma, M \vdash e \Rightarrow v, M'}{\sigma, M \vdash x := e \Rightarrow v, M' \{ \sigma(x) \mapsto v \}} \\
\\
\text{RECASSIGN} \frac{\sigma, M \vdash e_1 \Rightarrow r, M_1 \quad \sigma, M_1 \vdash e_2 \Rightarrow v, M_2}{\sigma, M \vdash e_1 . x := e_2 \Rightarrow v, M_2 \{ r(x) \mapsto v \}} \\
\\
\text{RECLOOKUP} \frac{\sigma, M \vdash e \Rightarrow r, M'}{\sigma, M \vdash e . x \Rightarrow M'(r(x)), M'} \\
\\
\text{SEQ} \frac{\sigma, M \vdash e_1 \Rightarrow v_1, M' \quad \sigma, M' \vdash e_2 \Rightarrow v_2, M''}{\sigma, M \vdash e_1 ; e_2 \Rightarrow v_2, M''} \\
\\
\text{IFT} \frac{\sigma, M \vdash e \Rightarrow \mathbf{true}, M' \quad \sigma, M' \vdash e_1 \Rightarrow v, M''}{\sigma, M \vdash \mathbf{if} \ e \ \mathbf{then} \ e_1 \ \mathbf{else} \ e_2 \Rightarrow v, M''} \\
\\
\text{IFF} \frac{\sigma, M \vdash e \Rightarrow \mathbf{false}, M' \quad \sigma, M' \vdash e_2 \Rightarrow v, M''}{\sigma, M \vdash \mathbf{if} \ e \ \mathbf{then} \ e_1 \ \mathbf{else} \ e_2 \Rightarrow v, M''} \\
\\
\text{WHILEF} \frac{\sigma, M \vdash e_1 \Rightarrow \mathbf{false}, M'}{\sigma, M \vdash \mathbf{while} \ e_1 \ \mathbf{do} \ e_2 \Rightarrow \cdot, M'} \\
\\
\text{WHILET} \frac{\sigma, M \vdash e_1 \Rightarrow \mathbf{true}, M' \quad \sigma, M_1 \vdash \mathbf{while} \ e_1 \ \mathbf{do} \ e_2 \Rightarrow v_2, M_2}{\sigma, M \vdash \mathbf{while} \ e_1 \ \mathbf{do} \ e_2 \Rightarrow v_2, M_2}
\end{array}$$

$$\begin{array}{c}
\text{LETV} \frac{\sigma, M \vdash e_1 \Rightarrow v, M'}{\sigma \{x \mapsto l\}, M' \{l \mapsto v\} \vdash e_2 \Rightarrow v', M''} \quad l \notin \text{Dom}(M') \\
\\
\text{LETF} \frac{\sigma \{f \mapsto \langle (x_1, \dots, x_n), e_1, \sigma \rangle\}, M \vdash e_2 \Rightarrow v, M'}{\sigma, M \vdash \text{let proc } f(x_1, \dots, x_n) = e_1 \text{ in } e_2 \Rightarrow v, M'} \\
\\
\begin{array}{c}
\sigma, M \vdash e_1 \Rightarrow v_1, M_1 \\
\sigma, M_1 \vdash e_2 \Rightarrow v_2, M_2 \\
\vdots \\
\sigma, M_{n-1} \vdash e_n \Rightarrow v_n, M_n
\end{array} \\
\text{CALLV} \frac{\sigma' \{x_1 \mapsto l_1\} \cdots \{x_n \mapsto l_n\} \{f \mapsto \langle (x_1, \dots, x_n), e', \sigma' \rangle\}, \quad M_n \{l_1 \mapsto v_1\} \cdots \{l_n \mapsto v_n\} \vdash e' \Rightarrow v', M'}{\sigma, M \vdash f(e_1, \dots, e_n) \Rightarrow v', M'} \quad \begin{array}{l} \sigma(f) = \langle (x_1, \dots, x_n), e', \sigma' \rangle \\ \forall i. l_i \notin \text{Dom}(M_n) \end{array} \\
\\
\text{CALLR} \frac{\sigma' \{x_1 \mapsto \sigma(y_1)\} \cdots \{x_n \mapsto \sigma(y_n)\} \{f \mapsto \langle (x_1, \dots, x_n), e, \sigma' \rangle\}, \quad M \vdash e \Rightarrow v, M'}{\sigma, M \vdash f \langle y_1, \dots, y_n \rangle \Rightarrow v, M'} \quad \sigma(f) = \langle (x_1, \dots, x_n), e, \sigma' \rangle \\
\\
\text{WRITE} \frac{\sigma, M \vdash e \Rightarrow n, M'}{\sigma, M \vdash \text{write } e \Rightarrow n, M'}
\end{array}$$

In OCaml, the language and values can be defined as follows:

```

type exp =
| NUM of int | TRUE | FALSE | UNIT
| VAR of id
| ADD of exp * exp
| SUB of exp * exp
| MUL of exp * exp
| DIV of exp * exp
| EQUAL of exp * exp
| LESS of exp * exp
| NOT of exp
| SEQ of exp * exp (* sequence *)
| IF of exp * exp * exp (* if-then-else *)
| WHILE of exp * exp (* while loop *)
| LETV of id * exp * exp (* variable binding *)
| LETF of id * id list * exp * exp (* procedure binding *)
| CALLV of id * exp list (* call by value *)
| CALLR of id * id list (* call by referenece *)
| RECORD of (id * exp) list (* record construction *)
| FIELD of exp * id (* access record field *)
| ASSIGN of id * exp (* assign to variable *)
| ASSIGNF of exp * id * exp (* assign to record field *)

```

```

    | WRITE of exp
and id = string

type loc = int
type value =
| Num of int
| Bool of bool
| Unit
| Record of record
and record = (id * loc) list
type memory = (loc * value) list
type env = binding list
and binding = LocBind of id * loc | ProcBind of id * proc
and proc = id list * exp * env

```

1. (70pts) Implemente the function `runb`:

```
runb : exp -> value
```

which takes a program expression and computes its value. Whenever the semantics is undefined, raise the exception `UndefinedSemantics`.

Examples:

- The program

```

let ret = 1 in
let n = 5 in
while (0 < n) {
  ret := ret * n;
  n := n - 1;
};
ret

```

is represented by

```

LETV ("ret", NUM 1,
      LETV ("n", NUM 5,
            SEQ (
              WHILE (LESS (NUM 0, VAR "n"),
                    SEQ (
                      ASSIGN ("ret", MUL (VAR "ret", VAR "n")),
                      ASSIGN ("n", SUB (VAR "n", NUM 1))
                    )
                ),
              VAR "ret")))

```

and produces 120.

- The program

```

let proc f (x1, x2) =
  x1 := 3;
  x2 := 3;
in
let x1 = 1 in
let x2 = 1 in
f <x1, x2>;
x1 + x2

```

is represented by

```

LETF ("f", ["x1"; "x2"],
  SEQ (
    ASSIGN ("x1", NUM 3),
    ASSIGN ("x2", NUM 3)
  ),
  LETV("x1", NUM 1,
    LETV("x2", NUM 1,
      SEQ(
        CALLR ("f", ["x1"; "x2"]),
        ADD(VAR "x1", VAR "x2")))))

```

and produces 6.

- The program

```

let f = {x := 10, y := 13} in
let proc swap (a, b) =
  let temp = a in
  a := b;
  b := temp
in
swap (f.x, f.y);
f.x

```

is represented by

```

LETV ("f", RECORD ([("x", NUM 10); ("y", NUM 13)]),
  LETF ("swap", ["a"; "b"],
    LETV ("temp", VAR "a",
      SEQ (
        ASSIGN ("a", VAR "b"),
        ASSIGN ("b", VAR "temp"))),
    SEQ (
      CALLV("swap", [FIELD (VAR "f", "x"); FIELD (VAR "f", "y")]),
      FIELD (VAR "f", "x")
    )
  )
)

```

and produces 10.

2. (30pts) Implement the function

```
gc: env -> memory -> memory
```

which takes an environment and a memory, and performs garbage collection (i.e. recycling memory locations that will not be used in the remaining program execution).