HW 6 **Prof. Caldwell**
COSC 4780

1 Semantics

The semantics for the core language are given by the following equations.

For Command :

$[\![L{:=}E : \mathit{comm}]\!](s) = \mathit{update}([\![L : \mathit{intloc}]\!], [\![E : \mathit{intexp}]\!](s), s)$
$[\![C_1 ; C_2 : \mathit{comm}]\!](s) = [\![C_2 : \mathit{comm}]\!]([\![C_1 : \mathit{comm}]\!](s))$
$[\![\textbf{if } E \textbf{ then } C_1 \textbf{ else } C_2 \textbf{ fi} : \mathit{comm}]\!](s) = \mathit{if}([\![E : \mathit{boolexp}]\!](s), [\![C_1 : \mathit{comm}]\!](s), [\![C_2 : \mathit{comm}]\!](s))$
$[\![\textbf{while } E \textbf{ do } C \textbf{ od} : \mathit{comm}]\!](s) = w(s)$
    where $w(s) = \mathit{if}([\![E : \mathit{boolexp}]\!](s), w([\![C : \mathit{comm}]\!](s)), s)$
$[\![\textbf{skip} : \mathit{comm}]\!](s) = s$

For Expression :

$[\![k : \mathit{intexp}]\!](s) = k$
$[\![@L : \mathit{intexp}]\!](s) = \mathit{lookup}([\![L : \mathit{intloc}]\!], s)$
$[\![E_1 + E_2 : \mathit{intexp}]\!](s) = \mathit{plus}([\![E_1 : \mathit{intexp}]\!](s), [\![E_2 : \mathit{intexp}]\!](s))$
$[\![\neg E : \mathit{boolexp}]\!](s) = \mathit{not}([\![E : \mathit{boolexp}]\!](s))$
$[\![E_1 = E_2 : \mathit{boolexp}]\!](s) = \mathit{equalbool}([\![E_1 : \mathit{boolexp}]\!](s), [\![E_2 : \mathit{boolexp}]\!](s))$
$[\![E_1 = E_2 : \mathit{boolexp}]\!](s) = \mathit{equalint}([\![E_1 : \mathit{intexp}]\!](s), [\![E_2 : \mathit{intexp}]\!](s))$

For Location :

$[\![\mathit{loc}_i : \mathit{intloc}]\!] = \mathit{loc}_i$

When P is a syntactic class and τ is a type, the semantic functions are denoted by Scott brackets as $[\![P : \tau]\!]$. These functions have the following types[1]:

$$\begin{array}{l}
(\lambda C.\ [\![C : \mathit{comm}]\!]) : \mathit{comm} \to \mathit{store} \to \mathit{store} \\
(\lambda E.\ [\![E : \mathit{intexp}]\!]) : \mathit{expression} \to \mathit{store} \to \mathit{int} \\
(\lambda E.\ [\![E : \mathit{boolexp}]\!]) : \mathit{expression} \to \mathit{store} \to \mathit{bool} \\
(\lambda L.\ [\![L : \mathit{inloc}]\!]) : \mathit{inloc} \to \mathit{inloc}
\end{array}$$

The corresponding OCaml functions and their types are as follows:

```
meaning_of_command : command → store → store
```

[1]The notation $[\![P : \tau]\!]$ builds the argument P into the function, we have lambda-abstracted the embedded argument out to show that the meaning function takes the syntactic phrase P as an argument as well.

```
meaning_of_intexp : expression → store → int
meaning_of_boolexp : expression → store → bool
meaning_of_inloc : inloc → inloc
```

We can use these semantics as rewrite rules ina left to right fashion to "evaluate" a program in a store.

So, for example,

$[\![\, loc_1{:=}1; loc_2{:=}@loc_1 + 1 : comm \,]\!]\langle 0,1,0\rangle$
$\quad\langle\langle$by the sequencing rule (;)$\rangle\rangle$
$=[\![\, loc_2{:=}@loc_1 + 1 : comm \,]\!]([\![\, loc_1{:=}1 : comm \,]\!]\langle 0,1,0\rangle)$
$\quad\langle\langle$by the assignment rule.$\rangle\rangle$
$= [\![\, loc_2{:=}@loc_1 + 1 : comm \,]\!](update([\![\, loc_1 : intloc \,]\!], [\![\, 1 : intexp \,]\!]\langle 0,1,0\rangle, \langle 0,1,0\rangle))$
$\quad\langle\langle$by the rule for locations.$\rangle\rangle$
$= [\![\, loc_2{:=}@loc_1 + 1 : comm \,]\!](update(loc_1, [\![\, 1 : intexp \,]\!]\langle 0,1,0\rangle, \langle 0,1,0\rangle))$
$\quad\langle\langle$by the rule for integer expressions.$\rangle\rangle$
$= [\![\, loc_2{:=}@loc_1 + 1 : comm \,]\!](update(loc_1, 1, \langle 0,1,0\rangle))$
$\quad\langle\langle$by the definition of *update.*$\rangle\rangle$
$= [\![\, loc_2{:=}@loc_1 + 1 : comm \,]\!]\langle 1,1,0\rangle$
$\quad\langle\langle$by the assignment rule.$\rangle\rangle$
$= update([\![\, loc_2 : loc \,]\!], [\![\, @loc_1 + 1 : intexp \,]\!], \langle 1,1,0\rangle, \langle 1,1,0\rangle)$
$\quad\langle\langle$by the rule for locations.$\rangle\rangle$
$= update(loc_2, [\![\, @loc_1 + 1 : intexp \,]\!], \langle 1,1,0\rangle, \langle 1,1,0\rangle)$
$\quad\langle\langle$by the rule for Plus.$\rangle\rangle$
$= update(loc_2, Plus([\![\, @loc_1 : intexp \,]\!]\langle 1,1,0\rangle, [\![\, 1 : intexp \,]\!]\langle 1,1,0\rangle), \langle 1,1,0\rangle)$
$\quad\langle\langle$by the rule for dereferencing.$\rangle\rangle$
$= update(loc_2, Plus(lookup([\![\, loc_1 : loc \,]\!], \langle 1,1,0\rangle), [\![\, 1 : intexp \,]\!]\langle 1,1,0\rangle), \langle 1,1,0\rangle)$
$\quad\langle\langle$by the definition of *lookup.*$\rangle\rangle$
$= update(loc_2, Plus(1, [\![\, 1 : intexp \,]\!]\langle 1,1,0\rangle), \langle 1,1,0\rangle)$
$\quad\langle\langle$by the rule for integers.$\rangle\rangle$
$= update(loc_2, Plus(1,1), \langle 1,1,0\rangle)$
$\quad\langle\langle$by the definition of *Plus.*$\rangle\rangle$
$= update(loc_2, 2, \langle 1,1,0\rangle)$
$\quad\langle\langle$by the definition of *update.*$\rangle\rangle$
$\langle 1,2,0\rangle)$

This is a tedious calculation, which is why it's nice to have an Ocaml program that can do it for us.

2 Semantics of Side Effecting Expressions

Notice that no expression has an effect on the store, this can be seen by examining the equations and noticing that the store is not part of the return value. We say: *There are no side-effecting expressions in the language.* Lots of languages have side-effecting expressions; a classic example is the $i++$ $(++i)$notation of C. The value of the expression $i++$ is the value of i and the expression has the side effect of incrementing the value stored in the memory location referenced by i. Here's an idiomatic usage in the C-code computing the length of a string.

```
  int strlen(cp)
    char *cp
 {
      int count = 0;
      while (*cp++) count++;
      return count;
 }
```

The variable `cp` is a pointer to a string. `*cp` dereferences the pointer to the string. Recall that strings are terminated by numm (0) so, as soon as the end of the string is reached, the value of the expression `*cp++` is `0`, which is interpreted as `false` in C and C++ - so the while loop stops.

If we add a side-effecting operation into the expressions in the core language, we need to modify the semantics. We need to modify the return type of `meaning_of_intexp` and `meaning_of_boolexp` to account for the new state – as well as the value of the expression. We'll add a side effecting expression of the form $L \leftarrow E$. We extend the expression type in OCaml as follows:

```
type expression =
     Num of int
   | Deref of loc
   | Plus of expression * expression
   | Not of expression
   | Eq of expression * expression
   | AssignE of loc * expression
;;
```

The typing rule for the new expression is as follows:

$$\frac{L : \mathit{intloc} \qquad E : \mathit{intexp}}{L \leftarrow E : \mathit{intexp}}$$

To account for side-effects, we must modify the type of the meaning function to return a pair. In the mathematical presentation, we will write $\langle x, y\rangle$ to denote the pair containing x and y. On OCaml this is simply written `(x,y)`. Here are the types of the new meaning functions.

$$(\lambda E.\ [\![\, E : \mathit{intexp}\,]\!]) : \mathit{expression} \to \mathit{store} \to (\mathit{int} \times \mathit{store})$$
$$(\lambda E.\ [\![\, E : \mathit{boolexp}\,]\!]) : \mathit{expression} \to \mathit{store} \to (\mathit{bool} \times \mathit{store})$$

In the OCaml code we will have the following.

```
meaning_of_intexp : expression → store → (int * store)
meaning_of_boolexp : expression → store → (bool * store)
```

The semantics for the new rule is given as:

$$\begin{array}{lcl} [\![\, L \leftarrow E : \mathit{intexp}\,]\!](s) & = & \mathit{let}\ \langle i, s_1\rangle = [\![\, E : \mathit{intexp}\,]\!](s)\ \mathit{in} \\ & & \mathit{let}\ s_2 = \mathit{update}([\![\, L : \mathit{intloc}\,]\!], i, s_1)\ \mathit{in} \\ & & \quad \langle i, s_2\rangle \end{array}$$

Thus, the expression E is evaluated (in the current store s) to the value $\langle i, s_1\rangle$ where i is the value of the expression and s_1 is the store that results. Note that E may have side effects itself. This new store s_1 is updated so that the value i is stored in location L. Note that these expressions could be nested.

```
loc2 ← ((loc1 ← (@loc1 + @loc2)) + @loc1)
```

If we evaluate the left subexpression for the + operator first, this expression will give a different answer than if we evaluate the right subexpression first. This means addition is no longer commutative! A similar situation arises in C++, consider the expression `((i++) + (++i))` and `((++i) + (i++))`. A naive programmer may reason as follows: "Since, a+b = b+a, I should always be able to swap addends around."

To add side-effecting expressions, we need to modify all the semantic equations (even for the non side-effecting expressions) so that they pass the store around. The rules for addition and Boolean equality need to account for the order of evaluation of the sub-expressions.

$$\begin{array}{rcl}
[\![\, k : \mathit{intexp} \,]\!](s) & = & \langle k, s\rangle \\
[\![\, @L : \mathit{intexp} \,]\!](s) & = & \langle \mathit{lookup}([\![\, L : \mathit{intloc} \,]\!], s), s\rangle \\
[\![\, E_1 + E_2 : \mathit{intexp} \,]\!](s) & = & \mathit{let}\langle i_1, s_1\rangle = [\![\, E_1 : \mathit{intexp} \,]\!](s)\ \mathit{in} \\
 & & \mathit{let}\langle i_2, s_2\rangle = [\![\, E_2 : \mathit{intexp} \,]\!](s_1)\ \ \mathit{in} \\
 & & \quad \langle i_1 + i_2, s_2\rangle \\
[\![\, \neg E : \mathit{boolexp} \,]\!](s) & = & \mathit{let}\langle b, s'\rangle = [\![\, E : \mathit{boolexp} \,]\!](s))\ \mathit{in} \\
 & & \quad \langle \mathit{not}\ b, s'\rangle \\
[\![\, E_1 = E_2 : \mathit{boolexp} \,]\!](s) & = & \mathit{let}\langle b_1, s_1\rangle = [\![\, E_1 : \mathit{boolexp} \,]\!](s)\ \mathit{in} \\
 & & \mathit{let}\langle b_2, s_2\rangle = [\![\, E_2 : \mathit{boolexp} \,]\!](s_1)\ \ \mathit{in} \\
 & & \quad \langle \mathit{equalbool}(b_1, b_2), s_2\rangle \\
[\![\, E_1 = E_2 : \mathit{boolexp} \,]\!](s) & = & \mathit{let}\langle i_1, s_1\rangle = [\![\, E_1 : \mathit{intexp} \,]\!](s)\ \mathit{in} \\
 & & \mathit{let}\langle i_2, s_2\rangle = [\![\, E_2 : \mathit{intexp} \,]\!](s_1)\ \ \mathit{in} \\
 & & \quad \langle \mathit{equalint}(i_1, i_2), s_2\rangle
\end{array}$$

For the semantics of commands, we need to update the meaning functions for those commands that use the meanings of expressions to account for their own meanings so that they use the pairs properly. We do not have to change the type of the meaning of command function.

$$\begin{array}{rcl}
[\![\, L{:=}E : \mathit{comm} \,]\!](s) & = & \mathit{let}(i, s_1) = [\![\, E : \mathit{intexp} \,]\!](s)\ \mathit{in} \\
 & & \quad \mathit{update}([\![\, L : \mathit{intloc} \,]\!], i, s_1) \\
[\![\, C_1 ; C_2 : \mathit{comm} \,]\!](s) & = & [\![\, C_2 : \mathit{comm} \,]\!]([\![\, C_1 : \mathit{comm} \,]\!](s)) \\
[\![\, \textbf{if}\ E\ \textbf{then}\ C_1\ \textbf{else}\ C_2\ \textbf{fi} : \mathit{comm} \,]\!](s) & = & \mathit{let}\langle b, s_1\rangle = [\![\, E : \mathit{boolexp} \,]\!](s)\ \mathit{in} \\
 & & \quad \mathit{if}(b, [\![\, C_1 : \mathit{comm} \,]\!](s_1), [\![\, C_2 : \mathit{comm} \,]\!](s_1)) \\
[\![\, \textbf{while}\ E\ \textbf{do}\ C\ \textbf{od} : \mathit{comm} \,]\!](s) & = & \mathit{let}\ w(s) = \\
 & & \quad \mathit{let}\langle b, s_1\rangle = [\![\, E : \mathit{boolexp} \,]\!](s)\ \mathit{in} \\
 & & \qquad \mathit{if}(b, w([\![\, C : \mathit{comm} \,]\!](s_1)), s) \\
 & & \mathit{in} \\
 & & \quad w(s) \\
[\![\, \textbf{skip} : \mathit{comm} \,]\!](s) & = & s
\end{array}$$

Problem 2.1. I have provided code to implement the semantic functions for the core language. Your assignment is to add the new constructor to the expression type and to update the functions `meaning_of_intexp`, `meaning_of_boolexp`, and `meaning_of_command` so that they implement these new semantics.