1 Expressions

This section revises the implementation of evaluating expression.

1.1 Syntax

We have the following syntax:

$$e ::= n \mid x(e_1, e_2) \mid x$$

In Racket, we define the syntax above as follows:

```
(struct e:number (value) #:transparent)      ; n
(struct e:binop (op lhs rhs) #:transparent)  ; x(e1, e2)
(struct e:variable (name) #:transparent)     ; x
```

For instance, here are a few example expressions:

```
(e:number 0)  ; 0
(e:number 1)  ; 1
; result * variable
(e:binop
  (e:variable '*)
  (e:variable 'result)
  (e:variable 'factor))
```

1.2 Semantics

The semantics of evaluating expressions is defined below. We write $e \Downarrow_E n$ to signify that expression e evaluates down to number n while using an environment E. Here, expression e and environment E are input parameters, while number n is the value being returned (output parameter).

$$n \Downarrow_E n \qquad x \Downarrow_E E(x) \qquad \frac{x \Downarrow_E f \qquad e_1 \Downarrow_E n_1 \qquad e_2 \Downarrow_E n_2 \qquad n = f(n_1, n_2)}{x(e_1, e_2) \Downarrow n}$$

Notice how each rule evaluates a different kind of expression. In programming language theory, we call such rules *syntax directed*.

- A number n evaluates down to itself.
- A variable x evaluates down to the value assigned to x in environment E.
- To evaluate a binary expression $x(e_1, e_2) \Downarrow n$: we evaluate the operator $x \Downarrow_E f$, where f represents a binary function, we evaluate the first operand $e_1 \Downarrow_E n_1$, we evaluate the second operand, $e_2 \Downarrow_E n_2$, and establish the output result n to be $f(n_1, n_2)$.

We implement each rule of the operational as a branch of conditional. The first rule is defined for numbers:

$$n \Downarrow_E n$$

We check if the input expression `e` is a number with `(e:number? e)`, and then return the number n stored inside the *struct* with `(e:number-value e)`.

```
[(e:number? e) (e:number-value e)]
```

The second rule is defined for variables:

$$x \Downarrow_E E(x)$$

We check if the input expression `e` is a variable with `(e:variable? e)`. We return $E(x)$ which holds the contents of variable x in environment x. We implement the environment as a hash-table (`hash`) where the keys are variables of type `e:variable` and the values are either numbers or Racket-functions that take two numbers and return a number. To lookup the environment we use function `hash-ref`, so $E(x)$ is implemented as `(hash-ref env e)`.

```
[(e:variable? e) (hash-ref env e)]
```

The third rule is defined for binary operators.

$$\frac{x \Downarrow_E f \qquad e_1 \Downarrow_E n_1 \qquad e_2 \Downarrow_E n_2 \qquad n = f(n_1, n_2)}{x(e_1, e_2) \Downarrow n}$$

We check if the input expression `e` is a binary operator with `(e:binop? e)`. We call each term above the fraction *pre-conditions*; in practice each pre-condition is implemented as a variable definition in Racket. Note that in our formalism the expression is given as $x(e_1, e_2)$, while in Racket we have a single variable `e`. In Racket we need to unpack the various fields from `e`: x is `(e:binop-op e)`, e_1 is `(e:binop-lhs e)`, e_2 is `(e:binop-rhs e)`. Formula $x \Downarrow_E f$ is a recursive evaluation of the operator (which is encoded as a variable), so we write the following Racket code:

```
(define f (e:eval env (e:binop-op e)))
```

Formula $e_1 \Downarrow_E n_1$ declares variable `n1` by recursively evaluating `(e:binop-lhs e)`, which represents e_1.

```
(define n1 (e:eval env (e:binop-lhs e)))
```

Similarly, formula $e_2 \Downarrow_E n_2$ can be implemented as

(define n2 (e:eval env (e:binop-rhs e)))

```
We implement formula~$n = f(n_1, n_2)$ with the following Racket code:
\begin{Racket}
        (define n (f n1 n2))
```

We are now ready to return `n` as we have n in the output parameter of $x(e_1, e_2) \Downarrow n$.

Below, you can find the full Racket code:

```
(define (e:eval env e)
  (cond ; n ⇓E  n
        [(e:number? e) (e:number-value e)]
        ; x ⇓E  E(e)
        [(e:variable? e) (hash-ref env e)]
        [(e:binop? e)
          ; x ⇓E  f
          (define f (e:eval env (e:binop-op e)))
          ; e1 ⇓E  n1
          (define n1 (e:eval env (e:binop-lhs e)))
          ; e2 ⇓E  n2
          (define n2 (e:eval env (e:binop-rhs e)))
          ; n = f(n1, n2)
          (define n (f n1 n2))
          n]))
```

2 Programs

2.1 Syntax

We define the syntax of our programs as a list of instructions (assignment or loop). This language has no concept of variable scoping, so all variables are global. Instruction $x = e$ stores the result of evaluating expression e in variable x. Instruction `while` e $\{p\}$ runs the list of instructions p while condition e evaluates down to a non-zero number.

$$
\begin{aligned}
i &::= x = e \mid \texttt{while}\ (e)\ \{p\} && \textit{instructions} \\
p &::= i :: p \mid [] && \textit{programs}
\end{aligned}
$$

For instance, the following program computes the factorial of 4, the input of the factorial is given in `factor` and the output of the the factorial is stored in variable `result`:

```
factor = 4
result = 1
while (factor) {
  result = result * factor
  factor = factor - 1
}
```

We can implement the AST with the following Racket:

```
(struct i:assign (var exp) #:transparent)
(struct i:while (cond body) #:transparent)
```

We can encode the AST of the factorial of 4 as follows:

```
(list
  ; factor = 4
  (i:assign (e:variable 'factor) (e:number 4))
  ; result = 1
  (i:assign (e:variable 'result) (e:number 1))
  ; while (factor) {
  (i:while (e:variable 'factor)
    (list
      ; result = result * factor
      (i:assign (e:variable 'result)
                (e:binop (e:variable '*)
                         (e:variable 'result) (e:variable 'factor)))
      ; factor = factor - 1
      (i:assign (e:variable 'factor)
                (e:binop (e:variable '-)
                         (e:variable 'factor) (e:number 1)))
    )
  ) ; }
)
```

2.2 Semantics

We first define the semantics of evaluating a single instruction, and then a list of instructions. The evaluation $i \Downarrow_E E'$ takes an input instruction i and an input environment E and outputs an environment E'.

$$\frac{e \Downarrow_E n}{x := e \Downarrow_E E[x := n]} \qquad \frac{e \Downarrow_E 0}{\texttt{while}\ (e)\ \{p\} \Downarrow_E E}$$

$$\frac{e \Downarrow_{E_1} n \qquad n \neq 0 \qquad p \Downarrow_{E_1} E_2 \qquad \texttt{while}\ (e)\ \{p\} \Downarrow_{E2} E_3}{\texttt{while}\ (e)\ \{p\} \Downarrow_{E_1} E_3}$$

The three evaluation rules are as follows:

- If we can evaluate expression e down to a number n, with $e \Downarrow_E n$, we can update environment E with by assigning x to n.
- If the condition e evaluates down to 0, then the loop terminates and returns the input environment E.
- If the condition e evaluates down to a number other than 0, then we run the body of the loop p which yields an environment E_2, we then take E_2 to execute the loop $\texttt{while}\ (e)\ \{p\}$ one more time, returning the output environment E_3.

We now present the semantics of lists of instructions p:

$$[] \Downarrow_E E \qquad \frac{i \Downarrow_{E_1} E_2 \qquad p \Downarrow_{E_2} E_3}{i :: p \Downarrow_{E_1} E_3}$$

The two rules are straightforward:

- If the list is empty there are no instructions to evaluate, so we return the input environment E.
- If there is at least one instruction i to evaluate, then we evaluate instruction i and get environment E_2, we feed environment E_2 to evaluate the rest of the list p and obtain environment E_3. We return environment E_3.

We implement the following rule in Racket.

$$\frac{e \Downarrow_E n}{x := e \Downarrow_E E[x := n]}$$

First, we check if the input instruction i is an assignment with `(i:assign? i)`. Second, we implement the pre-condition $e \Downarrow_E n$ as `(define v (e:eval env (i:assign-exp i)))`, since expression e corresponds to `(i:assign-exp i)`. Third, we return $E[x := n]$, which in Racket we can write as `(hash-set env (i:assign-var i) v)` since variable x is `(i:assign-var i)`.

```
[(i:assign? i) ; x := e
  ; e ⇓E n
  (define n (e:eval env (i:assign-exp i)))
  ; E [x:= n]
  (hash-set env (i:assign-var i) n)]
```

Now, we implement the two remaining rules in one branch. We highlight the evaluation of the condition with a yellow background to emphasize the similarity of these pre-conditions.

$$\frac{e \Downarrow_E 0}{\texttt{while}\ (e)\ \{p\} \Downarrow_E E}$$

$$\frac{e \Downarrow_{E_1} n \qquad n \neq 0 \qquad p \Downarrow_{E_1} E_2 \qquad \texttt{while}\ (e)\ \{p\} \Downarrow_{E2} E_3}{\texttt{while}\ (e)\ \{p\} \Downarrow_{E_1} E_3}$$

In Racket, that means that, after checking that the instruction is a while-loop with `(i:assign? i)`, then we must run $e \Downarrow_E n$, which we implement as

```
(define n (e:eval env (i:assign-exp i)))
```

where expression e is `(i:assign-exp i)`. Now, if $n = 0$, then we are in the presence of the first rule, and therefore should return the input environment `env`. Otherwise, $n \neq 0$, and we are in the presence of the second rule. There are two more pre-conditions to implement. First, we implement formula $p \Downarrow_{E_1} E_2$ as `(define E2 (p:eval env (i:while-body i)))` — recall that `env` represents E_1 in the second rule, an that `(i:while-body i)` represents the loop body p. We present function `p:eval` which evaluates a list of instructions after we conclude the evaluation of an instruction. Second, we implement formula `while` $(e)\ \{p\} \Downarrow_{E2} E_3$ as `(define E3 (i:eval E2 i))`, where expression `i` is the loop itself and environment `E2` represents E_2 that results from the the previous evaluation. We return `E3` since that is the output of evaluating the loop.

```
(define (i:eval env i)
  (cond
    [(i:assign? i) ; x := e
      ; e ⇓E n
      (define n (e:eval env (i:assign-exp i)))
      ; E [x:= n]
      (hash-set env (i:assign-var i) n)]
    [(i:while? i) ; while (e) { p }
      ; e ⇓E n
      (define v (e:eval env (i:while-cond i)))
      (cond [(equal? v 0) env] ; n = 0
            [else ; n ≠ 0
              ; p ⇓E1  E2
              (define E2 (p:eval env (i:while-body i)))
              ; while (e) {p} ⇓E2  E3
              (define E3 (i:eval E2 i))
              E3])]))
```

Finally, we present the implementation of the evaluation of a list of instructions.

$$[] \Downarrow_E E \qquad \frac{i \Downarrow_{E_1} E_2 \qquad p \Downarrow_{E_2} E_3}{i :: p \Downarrow_{E_1} E_3}$$

The first rule $[] \Downarrow_E E$ returns the environment `env` when the list is empty, implemented as `(empty? p)`. In the second rule, we have two pre-conditions, so we write two definitions. We evaluate the first instruction, $i \Downarrow_{E_1} E_2$, with `(define E2 (i:eval env (first p)))`, where `(first p)` represents i and environment `env` represents E_1. Notice that we evaluate an instruction with `i:eval` not with `p:eval`. We evaluate the rest of the instructions, $p \Downarrow_{E_2} E_3$, with

```
(define E3 (p:eval E2 (rest p)))
```

where `E2` results from the evaluation of the previous instruction, formally as E_2, and `(rest p)` represents p.

```
(define (p:eval env p)
  (cond [(empty? p) env] ; []
        [else ; i :: p
          ; i ⇓E1  E2
          (define E2 (i:eval env (first p)))
          ; p ⇓E2  E3
          (define E3 (p:eval E2 (rest p)))
          E3]))
```