Introducing the $\lambda_D$
We highlight in red an operation that produces a side effect: *mutating an environment*. 

\[
\begin{align*}
& e \Downarrow_E v \\
\Rightarrow & \quad E \leftarrow [x := v] \\
\Rightarrow & \quad (\text{define } x e) \Downarrow_E \text{ void} \\
& \quad (E-\text{def}) \\
\end{align*}
\]

\[
\begin{align*}
& t_1 \Downarrow_E v_1 \\
\quad & \quad t_2 \Downarrow_E v_2 \\
\Rightarrow & \quad t_1; t_2 \Downarrow_E v_2 \\
& \quad (E-\text{seq})
\end{align*}
\]
Language $\lambda_D$: Expressions

Because we have side-effects, the order in which we evaluate each sub-expression is important.

$v \Downarrow_E v \quad \text{(E-val)}$

$x \Downarrow_E E(x) \quad \text{(E-var)}$

$\lambda x.t \Downarrow_E (E, \lambda x.t) \quad \text{(E-lam)}$

$e_f \Downarrow_E (E_f, \lambda x.t_b) \quad e_a \Downarrow_E v_a \quad E_b \leftarrow E_f + [x := v_a] \quad t_b \Downarrow_{E_b} v_b \quad \text{(E-app)}$

Can you explain why the order is important?
Because we have side-effects, the order in which we evaluate each sub-expression is important.

\[
\begin{align*}
v & \Downarrow_E v \quad \text{(E-val)} \\
x & \Downarrow_E E(x) \quad \text{(E-var)} \\
\lambda x.t & \Downarrow_E (E, \lambda x.t) \quad \text{(E-lam)} \\
e_f & \Downarrow_E (E_f, \lambda x.t_b) \quad e_a & \Downarrow_E v_a \quad E_b & \leftarrow E_f + [x := v_a] \quad t_b & \Downarrow_{E_b} v_b \quad \text{(E-app)} \\
(e_f & e_a) \Downarrow_E v_b
\end{align*}
\]

Can you explain why the order is important? Otherwise, we might evaluate the body of the function \( e_b \) without observing the assignment \( x := v_a \) in \( E_b \).
Mutable operations on environments
Mutable operations on environments

Put

\[ E \leftarrow [x := v] \]

Take a reference to an environment \( E \) and mutate its contents, by adding a new binding.

Push

\[ E \leftarrow E' + [x := v] \]

Create a new environment referenced by \( E \) which copies the elements of \( E' \) and also adds a new binding.
Making side-effects explicit
Mutation as a side-effect

Let us use a triangle $\triangleright$ to represent the order of side-effects.

\[
\frac{e \downarrow_E v \quad \triangleright \quad E \leftarrow [x := v]}{(\text{define } x \ e) \downarrow_E \text{ void}} \quad (\text{E-def})
\]

\[
\frac{t_1 \downarrow_E v_1 \quad \triangleright \quad t_2 \downarrow_E v_2}{t_1; t_2 \downarrow_E v_2} \quad (\text{E-seq})
\]

\[
\frac{e_f \downarrow_E (E_f, \lambda x.t_b) \quad \triangleright \quad e_a \downarrow_E v_a \quad \triangleright \quad E_b \leftarrow E_f + [x := v_a] \quad \triangleright \quad t_b \downarrow_{E_b} v_b}{(e_f e_a) \downarrow_E v_b} \quad (\text{E-app})
\]
Implementing side-effect mutation

Making the heap explicit

We can annotate each triangle with a heap, to make explicit which how the global heap should be passed from one operation to the next. In this example, defining a variable takes an input global heap $H$ and produces an output global heap $H_2$.

\[
\text{\begin{array}{c}
\triangleright H \quad e \downarrow_E v \quad \triangleright H_1

\hline
\triangleright H \quad (\text{define } x \leftarrow e) \downarrow_E \text{ void} \quad \triangleright H_2
\end{array}}
\]

\[(E\text{-def})\]
Let us use our rule sheet!

\[
\frac{e \Downarrow_E v}{(E\text{-def})}
\]

\[
\frac{(\text{define } x e) \Downarrow_E \text{void}}{}
\]

\[
\frac{t_1 \Downarrow_E v_1 \quad t_2 \Downarrow_E v_2}{t_1; t_2 \Downarrow_E v_2 (E\text{-seq})}
\]

\[
\frac{e_f \Downarrow_E (E_f, \lambda x . t_b) \quad e_a \Downarrow_E v_a \quad E_b \leftarrow E_f + [x := v_a] \quad t_b \Downarrow_E v_b}{(e_f e_a) \Downarrow_E v_b (E\text{-app})}
\]

\[
\frac{v \Downarrow_E v}{(E\text{-val})}
\]

\[
\frac{x \Downarrow_E E(x)}{(E\text{-var})}
\]

\[
\frac{\lambda x . t \Downarrow_E (E, \lambda x . t)}{(E\text{-lam})}
\]
Examples
Evaluating Example 2

\[(\text{define } b \ (\text{lambda } (x) \ a))\]
\[(\text{define } a \ 20)\]
\[(b \ 1)\]

Input

\[E_0: []\]
---
Env: E_0
Term: \( (\text{define } b \ (\text{lambda } (y) \ a)) \)
Evaluating Example 2

\[
\begin{align*}
\text{Input} \\
(\text{define } b \ (\text{lambda} \ (x) \ a)) \\
(\text{define } a \ 20) \\
(b \ 1)
\end{align*}
\]

\[
\begin{align*}
\text{Output} \\
E0: [ \ ] \\
\text{Env: E0} \\
\text{Term: } (\text{define } b \ (\text{lambda} \ (y) \ a)) \\
\end{align*}
\]

\[
\begin{align*}
\lambda y.a \Downarrow_{E_0} (E_0, \lambda y.a) \\
\Rightarrow \\
(\text{define } b \ \lambda y.a) \Downarrow_{E_0} \text{ void}
\end{align*}
\]

\[
\begin{align*}
E0: [ \ ] \\
\text{Env: E0} \\
\text{Term: } (\text{define } b \ (\text{lambda} \ (y) \ a)) \\
\text{Value: } #<\text{void}>
\end{align*}
\]
Example 2: step 2

Input

\[
E_0: [ \\
   (b \cdot (\text{closure } E_0 (\lambda y \ a))) \\
]\]

---

Env: \(E_0\)
Term: \((\text{define } a \ 20)\)
Example 2: step 2

Input

\[
E_0: \begin{array}{c}
(b \cdot (\text{closure } E_0 (\lambda y \ a)))
\end{array}
\]

---

Env: E_0
Term: (define a 20)

Output

\[
E_0: \begin{array}{c}
(a \cdot 20)
(b \cdot (\text{closure } E_0 (\lambda y \ a)))
\end{array}
\]

Value: #<void>

\[
\begin{array}{c}
20 \downarrow_{E_0} 20
\end{array}
\]

\[
\begin{array}{c}
(\text{define } a 20) \downarrow_{E_0} \text{void}
\end{array}
\]
Example 2: step 3

Input

\[
E_0: \left[ \begin{array}{l}
(a \cdot 20) \\
(b \cdot (\text{closure } E_0 (\lambda (y) a)))
\end{array} \right]
\]

---

Env: \( E_0 \)
Term: \((b \ 1)\)
Example 2: step 3

**Input**

\[ E_0: [ \]
\[ (a . 20) \]
\[ (b . (\text{closure } E_0 (\lambda y. a))) \]
\[ ] \]

---

Env: E0
Term: (b 1)

**Output**

\[ E_0: [ \]
\[ (a . 20) \]
\[ (b . (\text{closure } E_0 (\lambda y. a))) \]
\[ ] \]

\[ E_1: [ E0 \]
\[ (y . 1) \]
\[ ] \]

Value: 20

\[ b \downarrow_{E_0} (E_0, \lambda y.a) \quad \Rightarrow \quad 1 \downarrow_{E_0} 1 \quad \Rightarrow \quad E_1 \leftarrow E_0 + [y := 1] \quad \Rightarrow \quad a \downarrow_{E_1} 20 \]

\[ (b 1) \downarrow_{E_0} 20 \]
Example 3

```
(define (f x) (lambda (y) x))
(f 10)
```

Input

```
E₀: []
---
Env: E₀
Term: (define (f x) (lambda (y) x))
```
Example 3

Input

```
(define (f x) (lambda (y) x))
(f 10)
```

Output

```
E0: []
---
Env: E0
Term: (define (f x) (lambda (y) x))
E0: [
    (f . (closure E0
            (lambda (x) (lambda (y) x))))]
Value: void
```
Example 3

Input

\[(\text{define } (f \ x) (\text{lambda } (y) \ x))\]
\[(f \ 10)\]

Output

\[E_0: [\] \]
---
Env: E0
Term: \[(\text{define } (f \ x) (\text{lambda } (y) \ x))\]

\[\lambda x. \lambda y. x \Downarrow_{E_0} (E_0, \lambda x. \lambda y. x)\]
\[(\text{define } f \ \lambda x. \lambda y. x) \Downarrow_{E_0} \text{void}\]
Example 3

\[(\text{define } (\text{f } x) (\text{lambda } (y) x))\]
\[(\text{f } 10)\]

Input

\[E_0: []\]
---
Env: E0
Term: \((\text{define } (f \ x) (\text{lambda } (y) \ x))\)

Output

\[E_0: [(f . (\text{closure } E0 (\text{lambda } (x) (\text{lambda } (y) x)))))]\]
Value: void

\[\lambda x.\lambda y.x \downarrow_{E_0} (E_0, \lambda x.\lambda y.x) \quad \Rightarrow \quad E_0 \leftarrow [f := (E_0, \lambda x.\lambda y.x)]\]

\[(\text{define } f \ \lambda x.\lambda y.x) \downarrow_{E_0} \text{void}\]
Example 3

Input

E₀: [ (f . (closure E₀ (lambda (x) (lambda (y) x)))) ]

Env: E₀
Term: (f 10)
Example 3

Input

\[
E_0: \[
(f . (\text{closure } E_0 \\
\quad (\text{lambda} (x) (\text{lambda} (y) x))))
\]

---
Env: E_0
Term: (f 10)
\]

Output

\[
E_0: \[
(f . (\text{closure } E_0 \\
\quad (\text{lambda} (x) (\text{lambda} (y) x))))
\]

E_1: [ E_0 (x . 10) ]
Value: (\text{closure } E_1 (\text{lambda} (y) x))
\]
Example 3

Input

\[ E_0: [\]
\[ (f . (\text{closure } E_0 (\lambda x (\lambda y x))))] \]

---

Env: E0

Term: (f 10)

Output

\[ E_0: [\]
\[ (f . (\text{closure } E_0 (\lambda x (\lambda y x))))] \]

\[ E_1: [ E_0 (x . 10)] \]

Value: (\text{closure } E_1 (\lambda y x))

\[ E_0(f) = (E_0, \lambda x. \lambda y. x) \]

\[ f \Downarrow_{E_0} (E_0, \lambda x. \lambda y. x) \]

\[ 10 \Downarrow_{E_0} 10 \]

\[ E_1 \leftarrow E_0 + [x := 10] \]

\[ \lambda y. x \Downarrow_{E_1} (E_1, \lambda y. x) \]

\[ (f 10) \Downarrow_{E_0} (E_1, \lambda y. x) \]
How to implement mutation without mutable constructs?
Motivating example

- Calling function $b$ must somehow access variable $a$ which is defined after its creation.

```scheme
(define b (lambda (x) a))
(define a 20)
(b 1)
```

Shared "mutable" state with immutable data-structures
Why immutability?

Benefits

- A necessity if we use a language without mutation (such as Haskell)
- Parallelism: A great way to implement fast and safe data-structures in concurrent code (look up copy-on-write)
- Development: Controlled mutation improves code maintainability
- Memory management: counters the problem of circular references (notably, useful in C++ and Rust, see example)

- Encoding shared mutable state with immutable data-structures is a great skill to have.
Heap

We want to design a data-structure that represents a heap (a shared memory buffer) that allows us to: allocate a new memory cell, load the contents of a memory cell, and update the contents of a memory cell.

Constructors

- `empty-heap` returns an empty heap
- `(heap-alloc h v)` creates a new memory cell in heap `h` whose contents are value `v`
- `(heap-put h r v)` updates the contents of memory handle `r` with value `v` in heap `h`

Selectors

- `(heap-get h r)` returns the contents of memory handle `r` in heap `h`
Heap usage

(define h empty-heap) ; h is an empty heap
(define r (heap-alloc h "foo")); stores "foo" in a new memory cell

What should the return value of heap-alloc?

- Should heap-alloc return a copy of h extended with "foo"? How do we access the memory cell pointing to "foo"?
- Should heap-alloc return a handle to the new memory cell? How can we access the new heap?
Heap usage

(define h empty-heap) ; h is an empty heap
(define r (heap-alloc h "foo")) ; stores "foo" in a new memory cell

What should the return value of heap-alloc?

- Should heap-alloc return a copy of h extended with "foo"? How do we access the memory cell pointing to "foo"?
- Should heap-alloc return a handle to the new memory cell? How can we access the new heap?

Function heap-alloc must return a pair eff that contains the new heap and the memory handle.

(struct eff (state result) #:transparent)
Heap usage example

Spec

```
(define h1 empty-heap) ; h is an empty heap
(define r (heap-alloc h1 "foo")); stores "foo" in a new memory cell
(define h2 (eff-state r))
(define x (eff-result r));
(check-equal? "foo" (heap-get h2 x)); checks that "foo" is in x
(define h3 (heap-put h2 x "bar")); stores "bar" in x
(check-equal? "bar" (heap-get h3 x)); checks that "bar" is in x
```
Handles must be unique

We want to ensure that the handles we create are **unique**, otherwise allocation could overwrite existing data, which is undesirable.

Spec

```
(define h1 empty-heap) ; h is an empty heap
(define r1 (heap-alloc h1 "foo")); stores "foo" in a new memory cell
(define h2 (eff-state r1))
(define x (eff-result r1))
(define r2 (heap-alloc h2 "bar")); stores "foo" in a new memory cell
(define h3 (eff-state r2))
(define y (eff-result r2))
(check-not-equal? x y) ; Ensures that x ≠ y
(check-equal? "foo" (heap-get h3 x))
(check-equal? "bar" (heap-get h3 y))
```
How can we implement a memory handle?
A simple heap implementation

- Let a handle be an integer
- Recall that the heap only grows (no deletions)
- A handle matches the number of elements already present in the heap
- When the heap is empty, the first handle is 0, the second handle is 1, and so on.
Heap implementation

- We use a hash-table to represent the heap because it has a faster random-access than a linked-list (where lookup is linear on the size of the list).
- We wrap the hash-table in a struct, and the handle (which is a number) in a struct, for better error messages. And because it helps maintaining the code.

```scheme
(struct heap (data) #:transparent)
(define empty-heap (heap (hash)))
(struct handle (id) #:transparent)
(struct eff (state result) #:transparent)
(define (heap-alloc h v)
  (define data (heap-data h))
  (define new-id (handle (hash-count data)))
  (define new-heap (heap (hash-set data new-id v)))
  (eff new-heap new-id))
(define (heap-get h k)
  (hash-ref (heap-data h) k))
(define (heap-put h k v)
  (define data (heap-data h))
  (cond
   [(hash-has-key? data k) (heap (hash-set data k v))]
   [else (error "Unknown handle!")]))
```
Contracts
Contracts

Adding some sanity to highly dynamic code.

- Design-by-contract: idea pioneered by Bertrand Meyer and pushed in the programming language Eiffel, which was recognized by ACM with the Software System Award in 2006.
- Contracts are pre- and post-conditions each unit of code must satisfy (e.g., a function)
- In some languages, notably F* and Dafny, pre- and post-conditions are checked at compile time!

Bibliography

Contracts in Racket

Use **define/contract** rather than **define** to test the validity of each parameter and the return value.

- The **→** operator takes a predicate for each argument and one predicate for the return value.
  For instance: `(→ symbol? real? string?)` declares that the first parameter is a symbol, the second parameter is numeric, and the return value is a string.

Example

```racket
(define/contract (f x y)
  ; Defines the contract
  (→ symbol? real? string?)
  (format "(~a, ~a)"))
```
Contracts examples

Read up on Racket's manual entry on: data-structure contracts

- real? for numbers
- any/c for any value
- list? for a list
- listof number? for a list that contains numbers
- cons? for a pair
- (or/c integer? boolean?) either an integer or a boolean
- (and/c integer? even?) an integer that is an even number
- (cons/c number? string?) a pair with a number and a string
- (hash/c symbol? number?) a hash-table where the keys are symbols and the keys are numbers