Today we will...

- Why should we care about functional programming?
- Implement environments using heaps and frames
- Review some usage examples
Why learn the Structure of Higher Level Languages?
Structure of Higher Level Languages

I postponed this discussion, because I felt that you are now better suited to understand and relate to the points being made.

- Why learn the fundamental concepts in all programming languages?
- Why learn different languages?
- Why focus on functional programming?
- Why use Racket?

Disclaimer

- Most of these claims are opinions
- These will be mostly informal claims
- We are not trying to find the best language (or programming model)
Overview

- Languages are just tools, learn which language is amenable to what context
- The best programming language does not exist (theoretically most languages are equivalent)
- Different languages have different characteristics that favour different domains: for instance, functional languages being used in Programming Language research, C/Fortran in scientific/high-performance computing
- A programming language is a computing interface: it is crucial to understand its meaning
- The importance of first-class functions and avoiding mutation
Semantics and idioms

Why should we care about language semantics?

- **A language is a computing user interface.**
  We are learning reusable, cross-cutting patterns.

- **The semantics must be unambiguous and precise.**
  It is not a matter of personal opinion how a conditional expression works. Language features must be described unambiguously to users.

- **The semantics defines a software contract.**
  Is the bug in the client's bug, or is it in our code?

- **Language idioms (patterns) are transferrable knowledge.**
  Understanding idioms (patterns) teaches you something that can be applied across languages and technologies.
How are all languages similar?
How are all languages the same?

- **Theoretical:** Any input-output behavior implementable in language X is implementable in language Y (Church-Turing thesis), and **equivalent to the $\lambda$-calculus without numbers**.
- **Practical:** Reoccurring fundamentals: variables, abstraction, recursive definitions.
How are languages different?
Languages are not slow/fast

- A language *implementation* is fast/slow, not the language itself
- Certain languages computational models are more amenable to implement efficiently
- Languages are user interfaces of computational models

How different languages behave in different contexts?
Why is C faster than all other languages?

Is it because C is "close to the metal?" That is, is C fast because its semantics matches the processor's semantics?
Why is C faster than all other languages?

Is it because C is "close to the metal?" That is, is C fast because its semantics matches the processor's semantics? No!

- Which processor? How could it match the semantics of all processors?
- Which compiler? The key of C's success lays in having good compilers.
- C is fast because it is **old and its interface remains stable**!
- C compilers are just **really** good at optimizing the target language.
- There is a set of good practices to write optimizer-ready C code

Take away

The facts above make C quite successful in High Performance Computing (large scale scientific codes).

Source: *C Is Not a Low-level Language: Your computer is not a fast PDP-11*. David Chisnall. ACM Queue vol. 16, no. 2. 2018
Why is Python slow multithreading?

- CPython (the main implementation of Python) is conditioned by the GIL (the Global Interpreter Lock) which effectively serializes parallel execution.
- To parallelize code we must run multiple processes, where shared memory is especially slow, which, in turn, slows down compute-bound programs.

Take away

- Avoid running compute-bound parallel codes in Python. Maybe choose C?

Constraint language programming

We solve the equation \( SEND + MORE = MONEY \) where each letter represents a digit in Prolog using a constraint language programming module:

```prolog
sendmore(Digits) :-
    Digits = [S,E,N,D,M,O,R,E],  % Create variables
    Digits ins 0..9,           % Associate domains to variables
    S #\= 0,                  % Constraint: S must be different from 0
    M #\= 0,
    all_different(Digits),    % all the elements must take different values
    1000*S + 100*E + 10*N + D  % Other constraints
     + 1000*M + 100*O + 10*R + E
    #=> 10000*M + 1000*O + 100*N + 10*E + Y,
    label(Digits).            % Start the search
```

Take away

Some problems are more amenable to certain programming languages.
How are languages different?

1. **The implementation matters:** A language implementation may be conditioned (faster/slower) in certain contexts.
2. **The model matters:** Certain problems are simpler/more efficient to write in specific languages.
3. **The domain matters:** A technology your business needs may only be available in some language (say TensorFlow in Python).
Why learn different languages?

Learn at least one new language every year.


Why should you care

- Deeper understanding of the differences and the similarities between languages
- Learn different approaches to the same problems
- More job opportunities
- Better technology choices (some technologies are only available in specific languages)
Why functional programming?
What is functional programming?

- Mutation is discouraged
- Higher-order functions serve as a generalization device

Why should we care?

- These features help designing correct, elegant, and efficient software
- Functional programming languages are heavily favoured by PL researchers, which means they serve as a test bed for PL design. Functional programming is close(r) to math formalism, thus implementation is usually simpler in functional programming languages.
- **Functional programming is trendy!** C++/Java/C#/Python/JavaScript are all incorporating functional programming idioms.
Why should we discourage mutation?

- Simpler to reason about: no surprises passing a data-structure to functions/objects
- Concurrency-ready: read-only means no race conditions (and no locks), which leads to simpler, faster code

Who is using it?

- immutable.js for JavaScript by Facebook
- vavr, PCollections, the Scala runtime, and the Closure runtime for Java
- immer for C++
- immutable collections for .NET
Why should we use higher-order functions?

- Simpler interface than objects (which method? which order?)
- Can be combined effectively (frameworks on combining functions)
Most programming languages features started out in functional programming languages.

- Garbage collection (LISP, 1959)
- Higher-order functions (lambda expressions in C++, C#, Java, Python) introduced in LISP (1959) and in ISWIM (1966)
- Type inference, e.g., auto in C++, var in C# (Hindley-Milner-Damas)
- Algebraic-data types and pattern matching (1970s in Hope)
- Recursion
A new wave of languages

- Swift: next-generation programming language for Apple systems
- Rust: functional programming meets system programming
- F#: an ML derivate for the .NET ecosystem
- Elixir: highly-available distributed system
- Clojure: a LISP-influenced language for the JVM and the web
How are we using functional programming?

- **OCaml**: web development (Facebook), distributed systems (Docker), finance (Bloomberg, Aesthetic Integration), hardware virtualization (Citrix)
- **Haskell**: verification (Facebook), distributed systems (Google), compilers (Intel), distributed systems (Microsoft)
- **Erlang**: communication (WhatsApp), ads (AddRoll), web backend (Bet365), finance (Goldman Sachs)
- **Elixir**: spam prevention (Pinterest), micro services (Lonely Planet)
- **F#**: data analysis (Kaggle), trading (Credit Suisse), gaming backend (GameSys)
- **Racket**: game scripting (Naughty Dog), image processing (YouPatch)
- **Scala**: middleware (Twitter), database (Netflix), microservices (Tumblr), web (The Guardian)

Honorable mentions

- ReasonML, Elm, PureScript, ClojureScript
Mutable environments
Today we implement a mutable environment.

Constructors

- **Empty**: The empty, root environment.
- **Put**: $E \leftarrow [x := v]$ updates an existing environment $E$ upon defining a variable. Returns the same frame, and updates the heap.
- **Push**: $E_2 \leftarrow E_1 + [x := v]$ creates a new environment $E_2$ by extending environment $E_1$ with one binding $x = v$. Returns the new environment.

Selectors

- **Variable Lookup**: $E(x)$ Looks up variable $x$ in the bindings of the current frame, otherwise recursively looks up the parent frame.
Figure 3.1: A simple environment structure.

Source: SICP book Section 3.2
Environment example

Environment visualization

Environment operations

\[
E_0 \leftarrow [x := 3]
\]
\[
E_0 \leftarrow [y := 5]
\]
\[
E_1 \leftarrow E_0 + [z := 6]
\]
\[
E_1 \leftarrow [x := 7]
\]
\[
E_2 \leftarrow E_0 + [m := 1]
\]
\[
E_2 \leftarrow [y := 2]
\]

Figure 3.1: A simple environment structure.

Source: SICP book Section 3.2
Constructors: Root

The root environment

```
(define root-alloc (heap-alloc empty-heap root-frame))
(define root-environ (eff-result root-alloc))
(define root-mem (eff-state root-alloc))
```
Constructors: Put

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

\[
(\text{define} \ (\text{environ-put} \ \text{mem} \ \text{env} \ \text{var} \ \text{val}) \\
(\text{define} \ \text{new-frm} \ ((\text{frame-put} \ ((\text{heap-get} \ \text{mem} \ \text{env}) \ \text{var} \ \text{val}))) \\
(\text{heap-put} \ \text{mem} \ \text{env} \ \text{new-frm}))
\]

Example

\[
E_0 \leftarrow [x := 3] \\
E_0 \leftarrow [y := 5]
\]

In Racket
Constructors: Put

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

\[(\text{define} \ (\text{environ-put} \ \text{mem} \ \text{env} \ \text{var} \ \text{val})
\begin{align*}
(\text{define} \ \text{new-frm} \ (\text{frame-put} \ (\text{heap-get} \ \text{mem} \ \text{env}) \ \text{var} \ \text{val}))
(\text{heap-put} \ \text{mem} \ \text{env} \ \text{new-frm}))
\end{align*}\]

Example

\[
E_0 \leftarrow [x := 3]
E_0 \leftarrow [y := 5]
\]

In Racket

\[
(\text{define} \ E_0 \ \text{root-environ})
(\text{define} \ m_1
(\text{environ-put}
(\text{environ-put} \ \text{root-heap} \ E_0 \ (\text{d:variable} \ 'x) \ (\text{d:number} \ 3);
E_0 \ (\text{d:variable} \ 'y) \ (\text{d:number} \ 5))))
\]
Constructors: Push

\[ E_2 \leftarrow E_1 + [x := v] \]

\[
\begin{align*}
\text{(define (environ-push mem env var val)} & \\
\text{\quad (define new-frame (frame env (hash var val)))} & \\
\text{\quad (heap-alloc mem new-frame))}
\end{align*}
\]

Example

\[ E_1 \leftarrow E_0 + [z := 6] \]

\[ E_1 \leftarrow [x := 7] \]
Constructors: Push

\[ E_2 \leftarrow E_1 + [x := v] \]

Example

\[
\begin{align*}
E_1 & \leftarrow E_0 + [z := 6] \\
E_1 & \leftarrow [x := 7]
\end{align*}
\]

In Racket

```
(define (environ-push mem env var val)
  (define new-frame (frame env (hash var val)))
  (heap-alloc mem new-frame))
```

```
(define e1-m2 (environ-push m1 E0 (d:variable 'z) (d:number 6)))
(define E1 (eff-result e1-m2))
(define m2 (eff-state e1-m2))
(define m3 (environ-put m2 E1 (d:variable 'x) (d:number 7)))
```
Continuing the example

Example

\[
\begin{align*}
E0 & \leftarrow \left[ x := 3 \right] \\
E0 & \leftarrow \left[ y := 5 \right] \\
E1 & \leftarrow E0 + \left[ z := 6 \right] \\
E1 & \leftarrow \left[ x := 7 \right] \\
E2 & \leftarrow E0 + \left[ m := 1 \right] \\
E2 & \leftarrow \left[ y := 2 \right]
\end{align*}
\]

In Racket

```
E0 \leftarrow \left[ x := 3 \right] \\
E0 \leftarrow \left[ y := 5 \right] \\
E1 \leftarrow E0 + \left[ z := 6 \right] \\
E1 \leftarrow \left[ x := 7 \right] \\
E2 \leftarrow E0 + \left[ m := 1 \right] \\
E2 \leftarrow \left[ y := 2 \right]
```
## Continuing the example

### Example

| E₀ ← [x := 3] | (define E₀ root-environ) |
| E₀ ← [y := 5] | (define m₁ |
| E₁ ← E₀ + [z := 6] | (environ-put root-heap E₀ (d:variable 'x) (d:number 3)) |
| E₁ ← [x := 7] | E₀ (d:variable 'y) (d:number 5)) |
| E₂ ← E₀ + [m := 1] | (define e₁-m₂ (environ-push m₁ E₀ (d:variable 'z) (d:number 6))) |
| E₂ ← [y := 2] | (define E₁ (eff-result e₁-m₂)) |

### In Racket

```racket
define E₀ root-environ
(define m₁
  (environ-put
    (environ-put root-heap E₀ (d:variable 'x) (d:number 3))
    E₀ (d:variable 'y) (d:number 5)))
(define e₁-m₂ (environ-push m₁ E₀ (d:variable 'z) (d:number 6)))
(define E₁ (eff-result e₁-m₂))
(define m₂ (eff-state e₁-m₂))
(define m₃ (environ-put m₂ E₁ (d:variable 'x) (d:number 7)))
(define e₂-m₄ (environ-push m₃ E₀ (d:variable 'm) (d:number 1)))
(define E₂ (eff-result e₂-m₄))
(define m₄ (eff-state e₂-m₄))
(define m₅ (environ-put m₄ E₂ (d:variable 'y) (d:number 2)))
```
Selector: Variable lookup

\[ E(x) \]

```
(define (environ-get mem env var)
    (define frm (heap-get mem env))  ;; Load the current frame
    (define parent (frame-parent frm)) ;; Load the parent
    (define result (frame-get frm var)) ;; Lookup locally
    (cond
        [result result] ;; Result is defined, then return it
        [parent (environ-get mem parent var)] ; If parent exists, recurse
        [else (error (format "Variable ~a is not defined" var))]])
```

Example

```
(check-equal? (environ-get m5 E2 (d:variable 'y)) (d:number 2))
(check-equal? (environ-get m5 E2 (d:variable 'm)) (d:number 1))
(check-equal? (environ-get m5 E2 (d:variable 'x)) (d:number 3))
```
A language of environments

Environment visualization

Figure 3.1: A simple environment structure.
Source: SICP book Section 3.2

(define parsed-m5
  (parse-mem
    '([E0 . ([x . 3] [y . 5]])
     [E1 . (E0 [x . 7] [z . 6]])
     [E2 . (E0 [m . 1] [y . 2]])])))

; Which is the same as creating the following data-structure
(heap
  (hash
    (frame #f
      (hash (d:variable 'y) (d:number 5) (d:variable 'x) (d:number 3))
      (handle 2)
      (frame (handle 0)
        (hash (d:variable 'y) (d:number 2) (d:variable 'm) (d:number 1))
        (handle 1)
        (frame (handle 0)
          (hash (d:variable 'z) (d:number 6) (d:variable 'x) (d:number 7))
          (check-equal? parsed-m5 m5)
        ))
    )))