Introduction to Scheme

Luis Diogo Couto

AARHUS UNIVERSITY
DEPARTMENT OF ENGINEERING

25 April 2013

Prelude
- A (very) brief history lesson
- Scheme: An Example
- Key Concepts

Intro to Scheme
- Hope you like syntax...

Summary

Prelude
- A (very) brief history lesson
- Scheme: An Example
- Key Concepts

Intro to Scheme
- Hope you like syntax...

Summary

A (very) brief history lesson

- LISP
  - A family of programming languages
  - Long and notable history (1958)
  - Pioneered many features of modern languages
- Scheme
  - One of the main dialects of LISP
  - Invented in 1975
  - Minimalist design (small core + extensions)
  - Your reference is R5RS standard from 1998

Scheme: An Example

```
; This procedure computes the factorial
; of its parameter.
(define factorial
  (lambda (x)
    (if (= x 1)
        x
        (* (factorial (- x 1)) x))))
```

Scheme: An Example

```
; This procedure computes the factorial
; of its parameter.
(define factorial
  (lambda (x)
    (if (= x 1)
        x
        (* (factorial (- x 1)) x))))
```

- Note the parentheses
- Very different notation
- Experiment: `http://racket-lang.org/`

Key Concepts

- Everything has a value
  - `x`
    has the value of `x`, of course
  - `43`, `'symbol`
    evaluate to themselves
  - `(- x 1)` and `(= x 0)`
    have the value of the evaluated expression
  - `(if (= x 0) x (- x 1))`
    is similar to C the ternary operator (`?:`)
  - `(lambda (x) ...)`
    has a procedure value

Key Concepts

- Everything has a value
    - `x`
      has the value of `x`, of course
    - `43`, `'symbol`
      evaluate to themselves
    - `(- x 1)` and `(= x 0)`
      have the value of the evaluated expression
    - `(if (= x 0) x (- x 1))`
      is similar to C the ternary operator (`?:`)
    - `(lambda (x) ...)`
      has a procedure value
- How to evaluate? Substitution model
    - More on this later
    - For now... rewrite expressions until value is reached

Prelude

A (very) brief history lesson

Scheme: An Example

Key Concepts

Intro to Scheme

Hope you like syntax...

Summary

Define

Introduce new definitions

syntax

`(define` $\langle\text{variable}\rangle$ $\langle\text{expression}\rangle$`)`

- $\langle\text{variable}\rangle$ can be any variable name

Define

Introduce new definitions

syntax

(`define` $\langle$variable$\rangle$ $\langle$expression$\rangle$)

- $\langle$variable$\rangle$ can be any variable name

```
(define x
    3)
```

Lambda

Construct a procedure

syntax

(`lambda` ($\langle$formals$\rangle$) $\langle$body$\rangle$)

- $\langle$body$\rangle$ is a sequence of expressions
- $\langle$formals$\rangle$ are parameter variables
- Remember, procedures are values

Lambda

Construct a procedure

syntax

(lambda ($\langle$formals$\rangle$) $\langle$body$\rangle$)

- $\langle$body$\rangle$ is a sequence of expressions
- $\langle$formals$\rangle$ are parameter variables
- Remember, procedures are values

```
(lambda (x) (+ x 1))
```

Let

Introduce temporary bindings

syntax

`(let` $\langle$bindings$\rangle$ $\langle$body$\rangle$`)`
`(let*` $\langle$bindings$\rangle$ $\langle$body$\rangle$`)`
$\langle$bindings$\rangle$ are of form: `((`$\langle$variable$_1\rangle$ $\langle$expression$_1\rangle$`) ...)`

Let

Introduce temporary bindings

syntax

`(let` $\langle\text{bindings}\rangle$ $\langle\text{body}\rangle$`)`
`(let*` $\langle\text{bindings}\rangle$ $\langle\text{body}\rangle$`)`
$\langle\text{bindings}\rangle$ are of form: `((`$\langle\text{variable}_1\rangle$ $\langle\text{expression}_1\rangle$`) ...)`

- returns value of body
- `let*` allows for $\langle\text{expression}_n\rangle$ to use $\langle\text{variable}_{n-1}\rangle$

```
(let ((x 1) (y 2))
     (+ x y))
```

Pairs

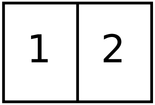
Structured datatype with 2 fields

syntax

construct: `(cons` ⟨expression⟩ ⟨expression⟩ `)`
car field: `(car` ⟨pair⟩ `)`
cdr field: `(cdr` ⟨pair⟩ `)`

- `car` and `cdr` similar to left, right, 1st, 2nd...
- historical names, do not worry about it

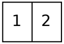
```
(1 . 2)
```



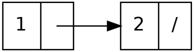
Lists

- A crucial datatype
- Defined as
  - empty list
  - pair with a list in `cdr`

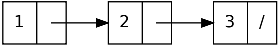
```
(1 . 2)
```




```
(1 2)
```




```
(1 2 3)
```



Map

Apply a procedure to each element of list(s)

syntax

(map ⟨procedure⟩ ⟨list-expression⟩ ...)

- lists must be of same length
- number of lists must match procedure arguments
- returns list of results

Map

Apply a procedure to each element of list(s)

syntax

(map ⟨procedure⟩ ⟨list-expression⟩ ...)

- lists must be of same length
- number of lists must match procedure arguments
- returns list of results

```
(map abs (list 1 -2 3))
```

Map

Apply a procedure to each element of list(s)

syntax

(map ⟨procedure⟩ ⟨list-expression⟩ ...)

- lists must be of same length
- number of lists must match procedure arguments
- returns list of results

```
(map abs (list 1 -2 3))
```

- Classic higher order programming
- Not special. We can implement it

Recursion

- Classic recursion computes recursive call first
- Takes return value of recursive call and calculates result
- Final result only calculated after all recursive calls return

Classic Recursion Factorial

```
(define (fact n)
  (if (= n 1)
    1
    (* n (fact (- n 1)))))
```

```
   (fact 4)
→ ((lambda (n) (if (= n 1)) 1 (* n (fact (- n 1)))) 4)
→ (if (= 4 1) 1 (* n (fact (- 4 1)))) 4)
→ (* 4 (fact (- 4 1)))
→ (* 4 (fact 3))
→ (* 4 (* 3 (fact 2)))
→ (* 4 (* 3 (* 2 (fact 1))))
→ (* 4 (* 3 (* 2 1)))
→ 24
```

Tail Recursion

- Why care?
- Not very efficient
  - stack space
- We can adress this with tail recursion
  - Allow optimisation
  - Unbounded number of calls
- Typically, done with a helper function and an argument to carry results

Tail Recursion Factorial

```
(define fact-tl
  (lambda (n r)
    (if (= n 1)
      r
      (fact-tl (- n 1) (* r n)))))
```

```
  (fact-tl 4)
→ (fact-tl 4 1)
→ (if (= n 1) r (fact-tl (- n 1) (* r n)))
→ (if (= 4 1) r (fact-tl (- 4 1) (* 1 4)))
→ (fact-tl (- 4 1) (* 1 4))
→ (fact-tl 3 4)
→ (fact-tl 2 12)
→ 24
```

Tail Recursion Factorial

```
(define fact-tl
  (lambda (n r)
    (if (= n 1)
      r
      (fact-tl (- n 1) (* r n)))))
```

```
  (fact-tl 4)
→ (fact-tl 4 1)
→ (if (= n 1) r (fact-tl (- n 1) (* r n)))
```

```
(define fact
    (lambda (n) (fact-tl n 1)))
```

```
→ (fact-tl 3 4)
→ (fact-tl 2 12)
→ 24
```

Set!

(side-effect) assignment

syntax

`(set!` $\langle$variable$\rangle$ $\langle$expression$\rangle$`)`

- Scheme is not purely functional
- But you **cannot** use it in the first 2 hand-ins
- Value of the expression itself is undefined

Set!

(side-effect) assignment

syntax

`(set!` ⟨variable⟩ ⟨expression⟩`)`

- Scheme is not purely functional
- But you **cannot** use it in the first 2 hand-ins
- Value of the expression itself is undefined

```
(set! x 5)
```

Prelude
A (very) brief history lesson
Scheme: An Example
Key Concepts

Intro to Scheme
Hope you like syntax...

Summary

Summary

- So Far...
  - Initial Scheme concepts
  - Crash course on syntax

Summary

- So Far...
  - Initial Scheme concepts
  - Crash course on syntax
- Next
  - FP Assignment 1
  - Substitution Model
  - Higher Order Programming (?)