Functional programming and the Scheme Programming Language

Programs are functions

Scheme programs are built out of small functions, each performing a single well-defined task. These functions are used to build higher-level functions until the “top-level” behavior is defined.

referential transparancy

- a function can be replaced by its equivalent value
- no ‘side effects’

higher-order functions / first-class functions

- passing functions as arguments to other functions,
- returning them as values from other functions, and
- assigning them to variables or storing them in data structures

No variables and no assignment statements

value semantics

- for an object, only its value counts, not its identity
- immutable objects have value semantics trivially

No loops (follows from above)

recursion for iteration

Hello World in Scheme

```scheme
(display "Hello World!")
```

Factorial in Scheme

```scheme
(define factorial
  (lambda(n)
    (if (= n 0)
        1
        (* n (factorial (- n 1))))))


(factorial 4)
```

Absolute Value in Scheme

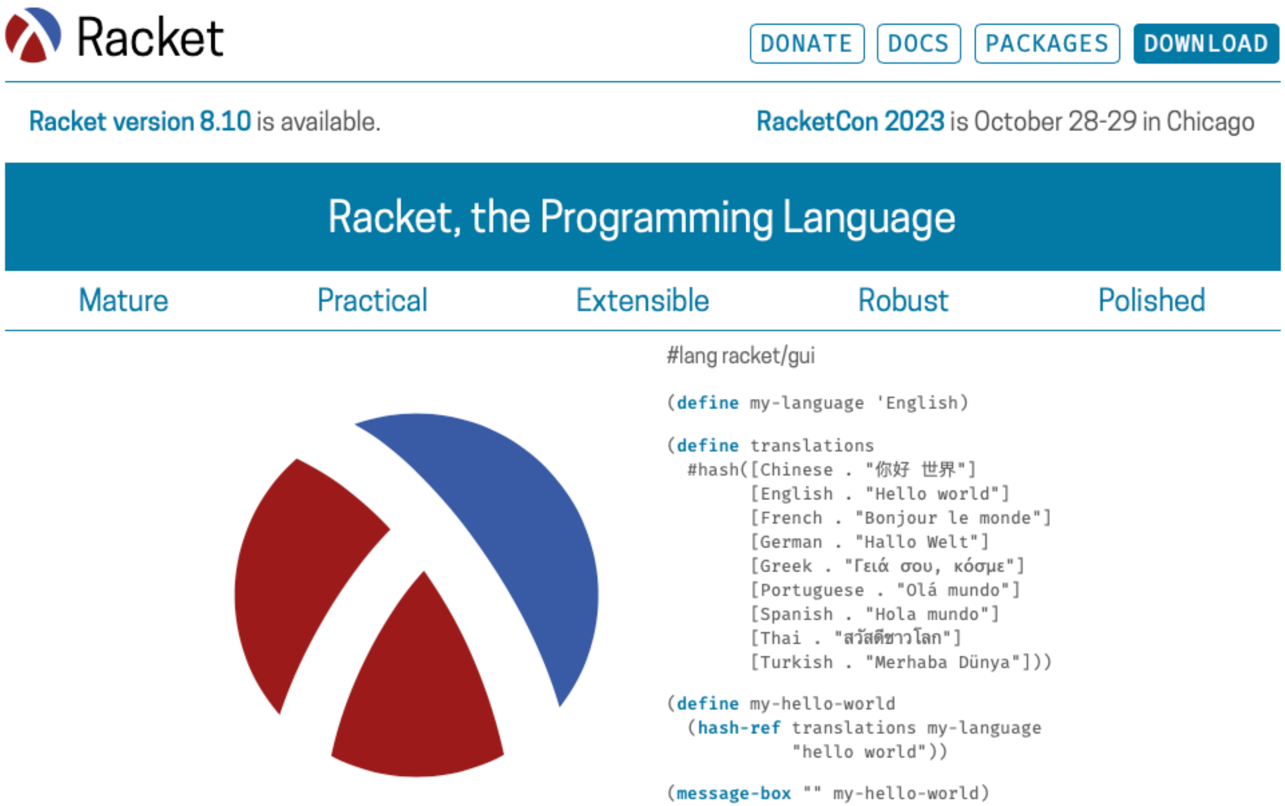
```scheme
(define absolute-value
  (lambda (x)
    (if (negative? x)
        (- x)
        x)))
```

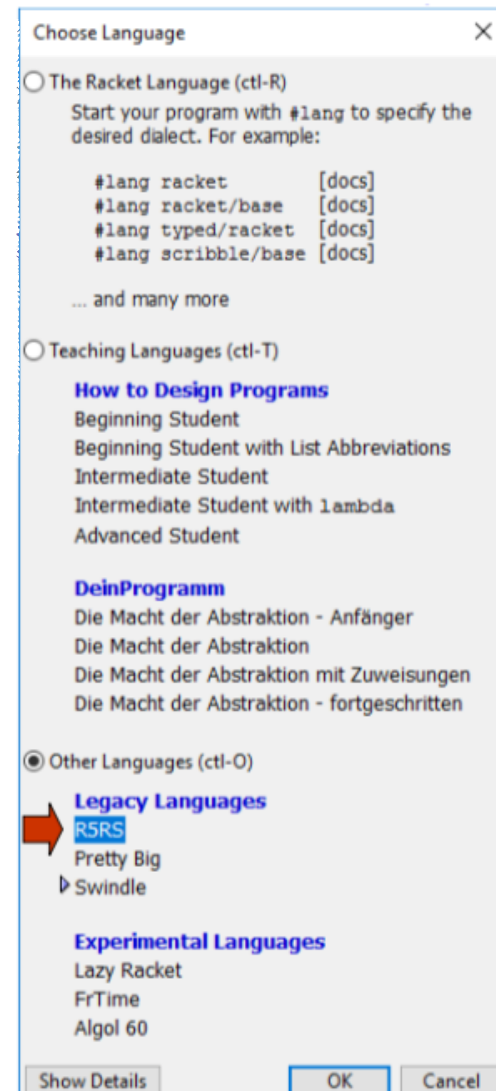
```scheme
(absolute-value -5)
```

Scheme: Standardization

- ANSI/IEEE standard
- *The Revised^5 Report on the Algorithmic Language Scheme* **[R^{5}RS]**

https://racket-lang.org





A read–eval–print loop (REPL), also termed an interactive toplevel or language shell, is a simple interactive computer programming environment that takes single user inputs, executes them, and returns the result.

The Scheme REPL

- Scheme **prompt**:

  >

The Scheme REPL

- Scheme **prompt**:

  ```
  > (* 2 (cos 0) (+ 4 6))
  ```

The Scheme REPL

- Scheme **prompt**:

```
> (* 2 (cos 0) (+ 4 6))
20
```

Scheme Syntax

- Let's examine the example in detail

  `> (* 2 (cos 0) (+ 4 6))`
  `20`

  constants: `0`, `2`, `4`, `6`

  identifiers: `cos`, `+`, `*`

  structured forms: `(...)`

  - Brackets are not for grouping!
    - In C/C++, "**`((1))`**" is the same as "**`1`**"
    - In Scheme, "**`((1))`**" is **NOT** "**`1`**"
    - Brackets are for delimiting structured forms.

  procedure application: `(`*proc* arg_1 `...` arg_n`)`

  "+": name of the **addition** procedure

  case-insensitive: `(cos 0)` $\equiv$ `(COS 0)`

Order of Evaluation

Order of Evaluation: When there are multiple subexpressions, which subexpression should be evaluated first?

Applicative Order:

(*proc* arg_1 **. . .** arg_n **)**

1. evaluate $arg_1, \ldots, arg_n$
2. evaluate *proc*
3. apply *proc* to $arg_1, \ldots, arg_n$

Numeric Procedures

Procedure	Meaning
`(+` x_1 `...` x_n`)`	The sum of $x_1, \ldots, x_n$
`(*` x_1 `...` x_n`)`	The product of $x_1, \ldots, x_n$
`(-` x y`)`	Subtract y from x
`(-` x`)`	Minus x
`(/` x y`)`	Divide x by y
`(max` x_1 `...` x_n`)`	The maximum of $x_1, \ldots, x_n$
`(min` x_1 `...` x_n`)`	The minimum of $x_1, \ldots, x_n$
`(sqrt` x`)`	The square root of x
`(abs` x`)`	The absolute value of x

Scheme Programs

`lambda` Expressions

- An **anonymous** procedure:

  ```
  (lambda (x) (* x 2))
  ```

  - `(lambda ...)` - a “name-less” procedure
  - `(x)` - parameter list
  - `(* x 2)` - procedure body

- Applying the procedure:

  ```
  > ((lambda (x) (* x 2)) 3)
  6
  ```

- The value of a procedure application is obtained by
  1. substituting the actual arguments for the formal parameters in the procedure body
  2. evaluating the procedure body

Top-Level Definitions

```
> (define y 3)
```

```
We use 'define' to bind a symbol to a value.

You can define numbers, characters, lists
and functions with this operator.
```

Top-Level Definitions

```
> (define y 3)
> y
3
```

Top-Level Definitions

```
> (define y 3)
> y
3
> (define double
    (lambda (x)
      (* x 2)))
```

‘double’ is a function

‘lambda’ can be omitted for brevity

Top-Level Definitions

```
> (define y 3)
> y
3
> (define double
    (lambda (x)
      (* x 2)))
> (double 3)        <-- call the function 'double'
6
```

Top-Level Definitions

```
> (define y 3)
> y
3
> (define double
    (lambda (x)
      (* x 2)))
> (double 3)
6
> (double (double 3))
12
```

Storing Programs in Files

- Demonstration
  - Edit program
  - Syntax-check program
  - Run program

Conditional Expressions

Boolean Values

- **Boolean values**: `#t`, `#f`
  - Anything that is not `#f` is considered true.
- A procedure that returns a boolean value is called a **predicate**.

Example: `absolute-value`

$$\text{abs}(x) = \begin{cases} -x & \text{if } x < 0 \\ x & \text{otherwise} \end{cases}$$

```
(define absolute-value
  (lambda (x)
    (if (negative? x)
      (- x)
      x)))
```

Conditional Form `if`

```
(if test-expression
  true-expression
  false-expression)
```

1. Evaluate *test-expression*.
2. If *test-expression* evaluates to true then evaluate *true-expression* and return its value.
3. Otherwise, evaluate *false-expression* and return its value.

Numeric Predicates

Predicate	Meaning
`(zero?` x`)`	x is zero
`(positive?` x`)`	x is positive
`(negative?` x`)`	x is negative
`(even?` x`)`	x is even
`(odd?` x`)`	x is odd

`some functions make predicates; such functions have names that end with '?'`

Relational Predicates

Predicate	Meaning
(= x y)	x is equal to y
(< x y)	x is less than y
(< x y)	x is greater than y
(<= x y)	x is no greater than y
(>= x y)	x is no less than y

Type Predicates

Predicate	Meaning
`(number?` x**`)`**	x is a number
`(boolean?` x**`)`**	x is a Boolean value (i.e., **`#t`**, **`#f`**)

Example: `sign`

$$\text{sign}(x) = \begin{cases} 1 & \text{if } x > 0 \\ 0 & \text{if } x = 0 \\ -1 & \text{otherwise} \end{cases}$$

```
(define sign
  (lambda (x)
    (cond
      ((positive? x)
        1)
      ((zero? x)
        0)
      (else
        -1))))
```

Conditional Form cond

```
(cond
  (test1  expr1)
  (test1  expr2)
  ...
  (testn  exprn)
  (else  default))
```

1. Evaluate *test*$_1$, *test*$_2$, . . . , *test*$_n$ in the order they appear.
2. If some *test*$_i$ evaluates to true, then stop evaluating the rest of the test expressions, but instead evaluate *expr*$_i$ and return its value.
3. Otherwise, evaluate *default* and return its value.

More Conditional Forms

Forms	**Meaning**
`(or` x_1 `...` x_n`)`	Logical or
`(and` x_1 `...` x_n`)`	Logical and
`(not` x`)`	Logical negation

- These are not procedures.

  Example: `(or` x_1 `...` x_n`)`
  - Evaluate $x_1, \ldots, x_n$ in the order they appear
  - If any of the arguments evaluates to true, return true rightaway without evaluating the rest of the arguments.
  - Otherwise, return `#f`.

Exercise

- Rewrite **`(not`** x**`)`** into an equivalent expression using only the **`if`** form.
- Rewrite **`(and`** x_1 **`...`** x_n**`)`** into an equivalent expression using only the **`cond`** form.
- The definition of **`and`** in terms of **`cond`** is only an approximation.

The `let` Form

```
(let ((var1  exp1)
      (var2  exp2)
      ...
      (varn  expn))
  body)
```

1. All expressions exp_1, exp_2, ... exp_n are evaluated first.
2. The results are then bound to newly created local variables var_1, var_2, ..., var_n.
3. With the local variables defined, evaluate *body* and return its result.

Equational Reasoning

Equational Reasoning

- We grow up knowing **equational reasoning** by heart:

$$\begin{aligned} & (1+2) \times (3-4) \\ = \quad & 3 \times (3-4) \\ = \quad & 3 \times -1 \\ = \quad & -3 \end{aligned}$$

- Because of referential transparency, the behavior of functional programs can be understood using equational reasoning!

Example: `double`

```
   (double (double 3))
=  (double ((lambda (x) (* x 2)) 3))
=  (double (* 3 2))
=  (double 6)
=  ((lambda (x) (* x 2)) 6)
=  (* 6 2)
=  12
```

Example: `absolute-value`

```
    (absolute-value -3)
=   ((lambda (x) (if (negative? x) (- x) x)) -3)
=   (if (negative? -3) (- -3) -3)
=   (if #t (- -3) -3)
=   (- -3)
=   3
```

Recursion

Recursion vs Iteration

- In imperative programming languages, repetition can be achieved by iterative constructs such as `for` or `while`.
- In functional programming languages, repetition is mainly achieved by **recursion**.

Example: `triangular` (1)

$$
\begin{aligned}
\text{triangular}(n) &= 1 + 2 + \ldots + n \\
&= \begin{cases} 1 & \text{if } n = 1 \\ n + \text{triangular}(n-1) & \text{otherwise} \end{cases}
\end{aligned}
$$

```
(define triangular
  (lambda (n)
    (if (= n 1)
      1
      (+ n (triangular (- n 1))))))
```

Example: `triangular` (2)

```
> (trace triangular)
> (triangular 3)
|(triangular 3)
| (triangular 2)
| |(triangular 1)
| |1
| 3
|6
6
> (untrace triangular)
```

Example: `triangular` (3)

```
   (triangular 3)
= ((lambda (n)
      (if (= n 1) 1 (+ n (triangular (- n 1)))))
    3)
= (if (= 3 1) 1 (+ 3 (triangular (- 3 1))))
= (if #f 1 (+ 3 (triangular (- 3 1))))
= (+ 3 (triangular (- 3 1)))
= (+ 3 (triangular 2))
= (+ 3 (if (= 2 1) 1 (+ 2 (triangular (- 2 1)))))
= (+ 3 (+ 2 (triangular (- 2 1))))
= (+ 3 (+ 2 (triangular 1)))
= (+ 3 (+ 2 (if (= 1 1) 1 (+ 1 (triangular (- 1 1))))))
= (+ 3 (+ 2 1))
= (+ 3 3)
= 6
```

Example: power (1)

$$x^n = \begin{cases} 1 & \text{if } n = 0 \\ x \times x^{n-1} & \text{if } n > 0 \end{cases}$$

```
(define power
  (lambda (x n)
    (if (zero? n)
      1
      (* x (power x (- n 1))))))
```

Example: `power` (2)

```
> (power 2 4)
|(power 2 4)
| (power 2 3)
| |(power 2 2)
| | (power 2 1)
| | |(power 2 0)
| | |1
| | 2
| |4
| 8
|16
16
```

Exercise

- Use equational reasoning to trace the execution of `(power 2 4)`.