Chapter 11

Scheme

Scheme - Functional Language

- Historical context
- Problem area addresses
- Data structures & how implemented
- Built-in items & libraries
- Sample code
- Translation process
- New features
- Strengths & weaknesses
- Conclusion

Historical Context – Functional Languages

- 1940's computers programed using machine language
- FORTRAN, first high-level programming language (1954)
- Lisp, first functional language (1958)
- Also in 1958
  - Flow-Matic precursor to COBOL
  - ALGOL - standard method of describing algorithms, influencer of all imperative languages

Historical Context – Scheme

- Scheme general semantics & syntax came from Lisp, lexical (static) scoping and block structure came from ALGOL
- Developed from MIT AI Memos known as the Lambda Papers (1975-1980)
- Versions in 1975, 1978, 1984, standardized in 1990, revised 1998, 2006, 2007
- Influenced R, Rust and Swift

Problem Area Addressed

LISP follows the mathematical paradigm of algorithms much more than Fortran

Fortran defined for number crunching

LISP for symbolic manipulation

Why use a Functional Language

Functional languages are useful for :

- game AI portion – nice list manipulations
- mathematical computations – easy to read
- concurrency – since no state
- want to prove correctness

Problem Area Addressed by Scheme

Contrary to Lisp, Scheme is functional yet:

- Uses lexical scoping
- Includes imperative features

Scheme Data Structures

Scheme (and Lisp) only have two data structures

- Atom – symbol
- Lists

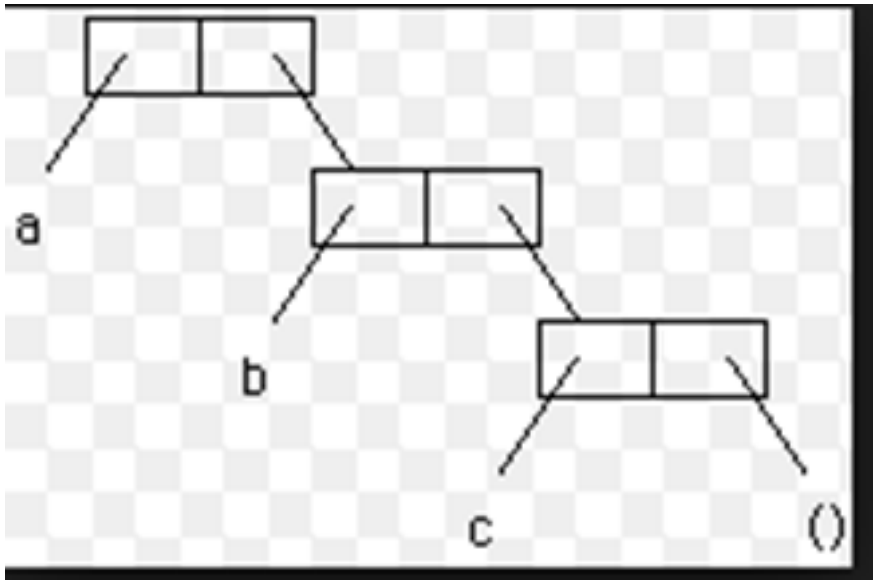
Atoms: X, hello, #t, #f, 0.566, ½, 3.5 + 4i

Lists: (A B C D) , (A (B C) D (E (F G)))

Internal Implementation

Data structure:

(a b c)



Major Built-In Procedures: car and cdr

car – takes a list are returns the first element in the list

(car '(1 2 3)) => 1

cdr – takes a list and returns the rest of the list

(cdr '(1 2 3)) => '(2 3)

Built-In Procedures

cons – takes an element and a list and inserts the element into the first position of the list

list – takes any number of elements and constructs a list from them

append – takes two lists and combines them into a single list

length – takes a list and returns it's length

Built-In Procedures

define – takes a name and a literal and binds the name to the literal

lambda – takes a list of operands and a function definition and optional a list of values, and creates a function

Example:

```
(define x '(a b c))                ; x is a list
(define x (lambda                  ; x is a procedure
          (n1 n2) (+ n1 n2)
)
```

Built-In Predicates

Predicate a procedure that returns #t or #f. By convention, predicates end with '?'.

null?

list?

boolean?
number?

equal?
procedure?

zero? - expects a number

even? - expects an integer

odd? - expects an integer

=, <, <=, >, >= - expect numbers

Scheme Data Libraries

Racket libraries:

- Draw - basic drawing tools, including drawing contexts such as bitmaps and PostScript files.
- Gui - GUI widgets such as windows, buttons, checkboxes, and text fields. The library also includes a sophisticated and extensible text editor.
- Pict - functional abstraction layer over draw, useful for creating slide presentations and images

Sample Code

Take a start and end and return a list of the numbers between, inclusive.

Example (myCount 1 5) returns (1 2 3 4 5)

```
(define myCount
    (lambda (start stop)
        (if (<=  start  stop)
         (cons  start  (myCount (+ start 1) stop))
         '())
         )
)
```

Scheme Translation

Scheme is interpreted

Scheme interpreter cycles continuously through

- Read
- Evaluate
- Print

New Features of Functional Programs

Ability to create functions “on the fly” – lambda

Higher order functions

Higher Order Functions

Higher order functions are functions that takes functions as arguments

Example: map

```
(define square (lambda (x) (* x x)))

(map square '(2 3 4))    returns '(4 9 16)
```

Create Functions on-the-fly

From before:

(define square (lambda (x) (* x x)))

(map square '(2 3 4)) returns '(4 9 16)

Equivalently,

(map (lambda (x) (* x x))) '(2 3 4))

Strengths of Functional Languages

Without understanding functional programming, you can't invent MapReduce, the algorithm that makes Google so massively scalable. The terms Map and Reduce come from Lisp and functional programming.

Purely functional programs have no side effects and are thus trivially parallelizable which is good for concurrency

Weaknesses of Functional Languages

Performance - Using only immutable values (no side-effects) and recursion can lead to performance problems – high RAM use and speed

Debugging - Writing pure functions is easy, but when combining them, debugging gets hard

Pure functions and I/O don't mix

For some, recursion isn't natural

Conclusion – Why Study Scheme?

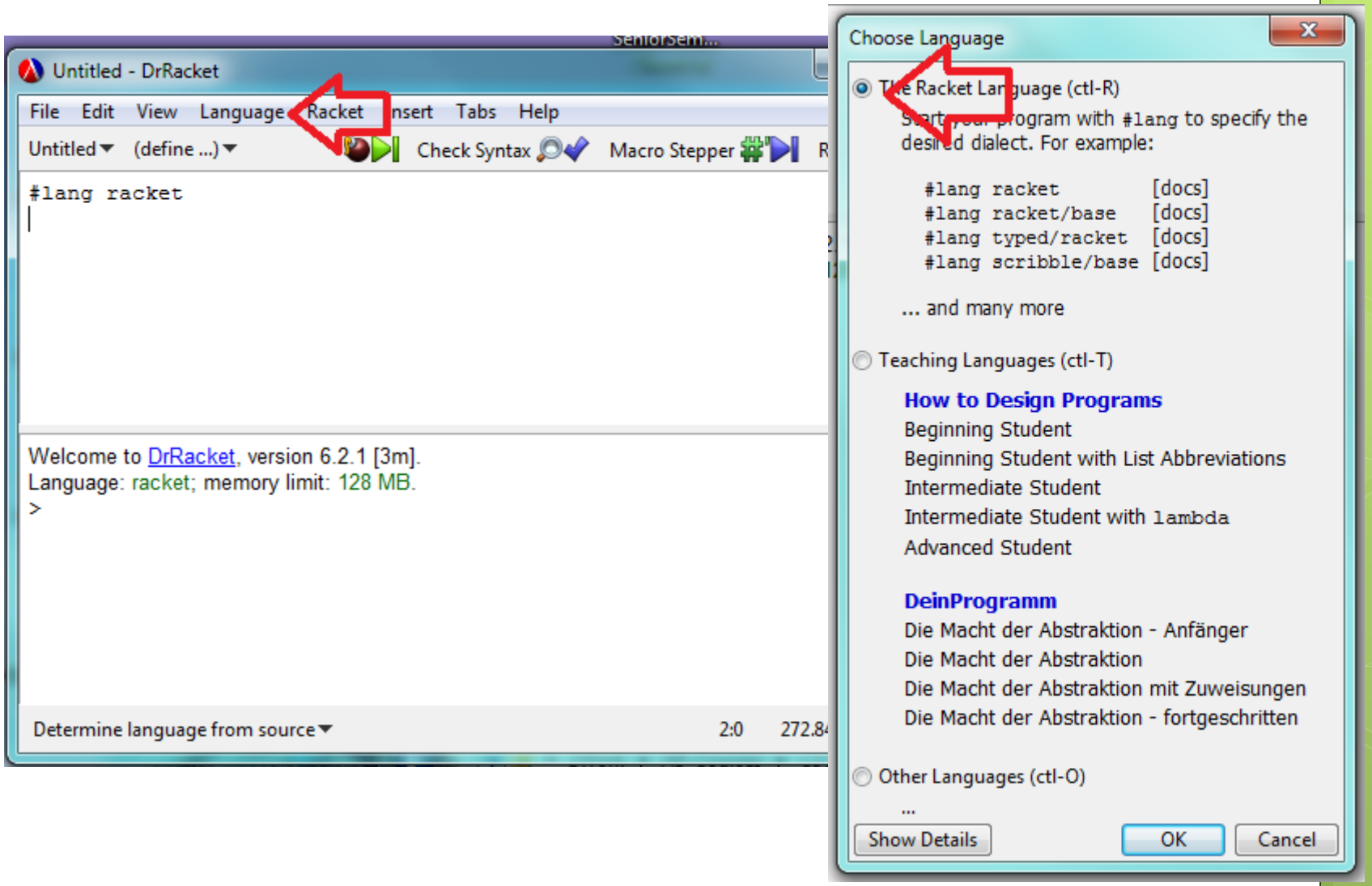
- Writing programs in Scheme helps you think recursively
- Recursion is a powerful problem solving skill
- Good for demonstrating certain language features such as higher order functions

Conclusion

Experience programming in Scheme will make you a stronger programmer and more knowledgeable about programming languages in general.

Scheme via Dr. Racket on Windows

Dr. Racket


Untitled - DrRacket
File Edit View Language Racket Insert Tabs Help
Untitled (define ...)
Check Syntax
Macro Stepper
#lang racket
Welcome to DrRacket, version 6.2.1 [3m].
Language: racket; memory limit: 128 MB.
>
Determine language from source
2:0
Choose Language
The Racket Language (ctl-R)
Start your program with #lang to specify the desired dialect. For example:
#lang racket [docs]
#lang racket/base [docs]
#lang typed/racket [docs]
#lang scribble/base [docs]
... and many more
Teaching Languages (ctl-T)
How to Design Programs
Beginning Student
Beginning Student with List Abbreviations
Intermediate Student
Intermediate Student with lambda
Advanced Student
DeinProgramm
Die Macht der Abstraktion - Anfänger
Die Macht der Abstraktion
Die Macht der Abstraktion mit Zuweisungen
Die Macht der Abstraktion - fortgeschritten
Other Languages (ctl-O)
...
Show Details
OK
Cancel

Scheme on katie

On kaite: mzscheme located at /usr/bin/mzscheme.

To invoke: mzscheme

```
Welcome to Racket v5.2.1.
>

>(load "filename.mz")

> (exit)
```

Read-Evaluate-Print

Scheme interpreter cycles continuously through

- Read
- Evaluate
- Print

Sometimes called REPL for Read-Evaluate-Print Loop (pronounced REP-ple)

Items are automatically evaluated.

Single quote - “don’t evaluate.”

```
> a        =>   a: unbound identifier
> ‘a       =>   a
```

Basic Types

Literals (numbers, Booleans, strings, characters) evaluate to themselves. For example:

```
>     23          =>  23

>#t               => #t

> “hello”         => “hello”

> #\c             => #\c
```

Scheme – Don't Evaluate

- Single quote lists since don't want evaluation to take place
- Once a list is quoted, don't quote elements within it

Example:

> (1 2 3) => application: not a procedure;
expected procedure, given: 1

> '(1 2 3) => '(1 2 3)

Built-In Procedures

if – takes a predicate, then-expression, and else-expression and evaluates one of the expressions based on the predicate

when – takes a predicate and expression to eval

unless – takes a predicate and expression to evaluate when the predicate is false

Built-In Procedures

cond – takes any number of pairs where each pair consists of a predicate followed by an expression. It searches down the list of pairs, and evaluates the expression associated with the first predicate that is true. The final predicate can be the word else.

Built-In Procedure - map

map - takes an function and a list. It applies the function to each element of the list and returns the resulting list.

Example:

```
(map (lambda (x) (* x 2)) '(1 2 3 4))
        => '(2 4 6 8)
```

Built-In Procedure - member

member (or **memq**) - takes an element and a list. It searches the list and if it finds the element, it returns the tail of the list beginning with that element. Otherwise it returns #f

Example:

(member 'a '(1 b a d 3)) => (a d 3)

(member 'a '(1 b d 3)) => #f

Built-In Procedure - assoc

assoc - takes an element and a list of pairs. It looks through the list and returns the first pair for which the first element matches

Example:

(assoc ‘a ‘((c 4) (b 3) (a 5) (d 6)) => (a 5)

(assoc ‘(q2 0) ‘((q2 4) q3) ((q3 1) q2) ((q2 0) q5)) =>
((q2 0) q5)

(assoc ‘a ‘((c 4) (b 3)) => #f

Equality

= only for numbers.

equal? for any item

eq? most discriminating

Example:

```
(define a '(1 2 3))
(equal? a   '(1 2 3))  => #t
(eq? a '(1 2 3))  => #f
```

Built-In Procedure - let

let - takes two arguments, a list of pairs and an expression. Each pair in the first list consists of an element and an expression. The 2nd argument of let is evaluated using the constant assignments given in the first argument. In other words, the variables are bound by the let for the duration of the expression.

Example:

```
(let (                  ; first argument
        (a 10)
        (b 15)
    )
    (+ a b)             ; second argument
)
```

Built-In Procedure - let

let - the initial values are computed and results are bound to variables

let* - from left to right, compute initial value and bind to variable, so earlier bindings can be used in later bindings

Example:

```
(let* ((a 2) (b 3 ) (c (+ a b)))
        (begin
                (display "c is ")
                (display c)
                (newline)
        )
)
```

I/O Scheme Functions

I/O Scheme functions:

read – read one Scheme object from the standard input and return it (thus, this is a function with a side effect , a new constant exists in the environment)

(define x (read))

write – print a representation of its argument to standard output so data could be read back in

(write (+ 3 5)) => 8

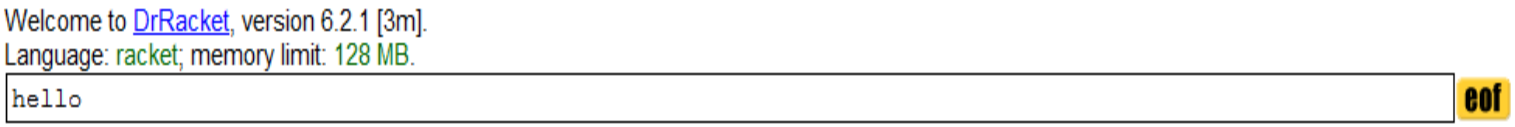
display – similar to write but datatypes are written as raw bytes or characters (not meant to be written back in)

Scheme Function read

read – read one Scheme object

Example:

```
(define x (read))
x
```



Returns ‘hello

Scheme Function read

read – read one Scheme object

Example:

```
(define x (read))
x
```

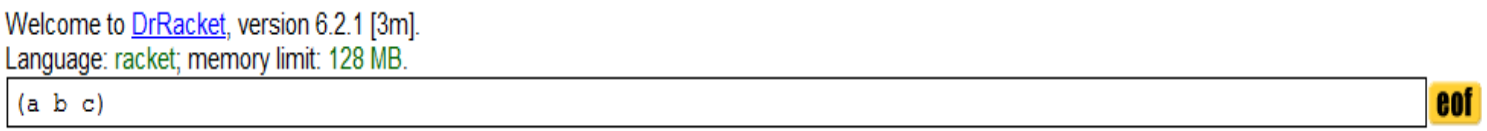
Returns ‘(a b c)

Scheme Function read-char

read-char – read a single character

Example:

```
(define x (read-char))
x
```



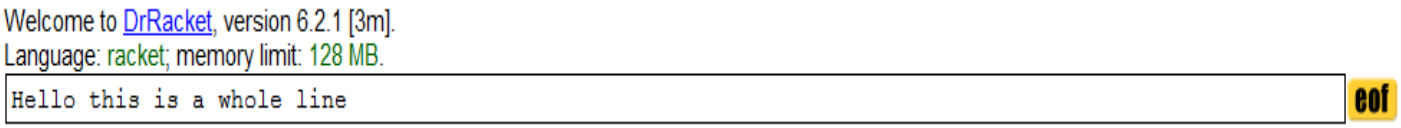
Returns #\(

Scheme Function read-line

read-line – read up to end of file

Example:

```
(define x (read-line))
x
```



Returns "Hello this is a whole line"

I/O Scheme File Functions

I/O Scheme functions:

open-input-file – as expected

Example: (open-input-file “table.csv”)

Example:

(read-line (open-input-file “table.csv”))

eof-object? – as expected

Example:

```
(if (eof-object? (read-line
                  (open-input-file “table.csv”)))
```

Ports

Have the usual ports – input, output and error

open-input-file – takes a string pathName, opens the file for input and returns an input-port, or triggers *exn:fail:filesystem* exception

open-output-file – similarly, but will create a file, so it can't already exist

close-input-port
close-output-port
custodian-shutdown-all