An Alternative Introduction to Programming

Read: A Tutorial of the Scheme Programming language

Shen Zheyu

October 6, 2017

Table of Contents I

Introduction

- Shen Zheyu, sophomore ECE student
- My GitHub: `http://github.com/arsdragonfly/`
- SSTIA projects `https://github.com/SSTIA`

Overview

This seminar is intended to provide a different introduction to programming from VG101 (and arguably many other courses). In a (hopefully) friendly way, you'll learn many useful things unlikely to be found in other introductory material.

Overview

This seminar is intended to provide a different introduction to programming from VG101 (and arguably many other courses). In a (hopefully) friendly way, you'll learn many useful things unlikely to be found in other introductory material.
Much of the content is adapted from *The Little Schemer, Fourth Edition* by D. P. Friedman and M. Felleisen. If you're very interested, You may also want to read *Structure and Interpretation of Computer Programs* by H. Abelson and G. J. Sussman.

Setup

To begin with this seminar, you need to have a Scheme interpreter up and running. I personally recommend Racket (`www.racket-lang.org`), though other programs may also work (MIT Scheme, Guile, Chez Scheme, etc.).

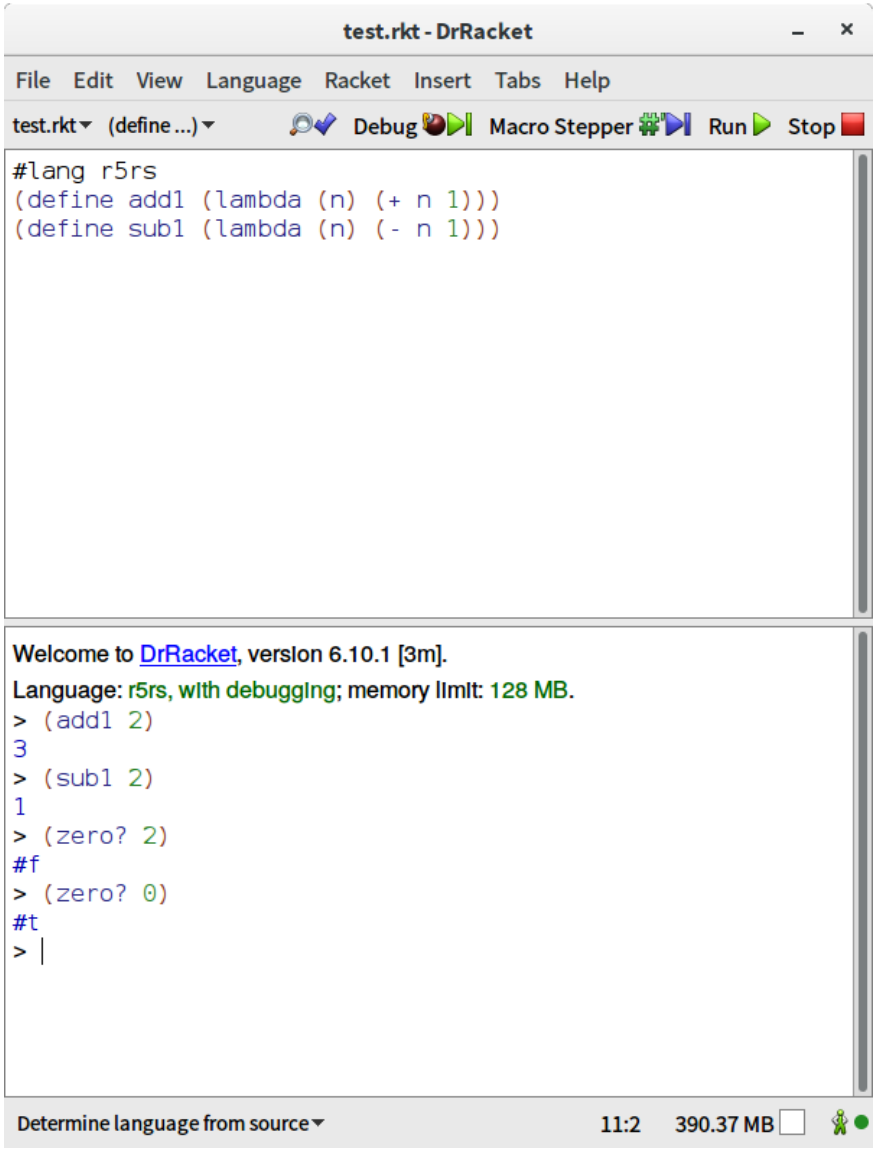
Setup

Before we start, we need to make sure that every program contains the following definitions of primitive functions:

```
(define add1
  (lambda (n)
    (+ n 1)))
(define sub1
  (lambda (n)
    (- n 1)))
```

It's recommended for this seminar now that you write your program in a single source code file, so that latter definitions of functions can build upon previous already-written ones.

Setup


test.rkt - DrRacket
File Edit View Language Racket Insert Tabs Help
test.rkt (define ...) Debug Macro Stepper Run Stop
#lang r5rs
(define add1 (lambda (n) (+ n 1)))
(define sub1 (lambda (n) (- n 1)))
Welcome to DrRacket, version 6.10.1 [3m].
Language: r5rs, with debugging; memory limit: 128 MB.
> (add1 2)
3
> (sub1 2)
1
> (zero? 2)
#f
> (zero? 0)
#t
>
Determine language from source
11:2
390.37 MB

Arithmetic on Natural Numbers

- What's the answer of `(add1 0)`?
- *1*.

Arithmetic on Natural Numbers

- What's the answer of `(add1 0)`?
- *1*.
- What's the answer of `(sub1 3)`?
- *2*.

Arithmetic on Natural Numbers

- What's the answer of `(add1 0)`?
- *1.*
- What's the answer of `(sub1 3)`?
- *2.*
- What's the answer of `(add1 (add1 0))`?
- *It's the answer of `(add1 1)`.*

Arithmetic on Natural Numbers

- What's the answer of `(add1 0)`?
- *1.*
- What's the answer of `(sub1 3)`?
- *2.*
- What's the answer of `(add1 (add1 0))`?
- *It's the answer of `(add1 1)`.*
- What's the answer of `(add1 1)` then?
- *2.*

Arithmetic on Natural Numbers

- What's the answer of `(zero? 0)`?
- `#t`, *which means "true".*

Arithmetic on Natural Numbers

- What's the answer of `(zero? 0)`?
- `#t`, *which means "true".*
- What's the answer of `(zero? 810)`?
- `#f`, *which means "false".*

Arithmetic on Natural Numbers

- What's the answer of `(zero? 0)`?
- `#t`, *which means "true".*
- What's the answer of `(zero? 810)`?
- `#f`, *which means "false".*
- What's the answer of `(zero? (sub1 (sub1 2)))`?
- `#t`

λ ?

- What's the answer of `(lambda (x) (add1 (add1 x)))`?
- *A* $lambda\ expression$*, which is similar to a mathematical function.*

λ ?

- What's the answer of `(lambda (x) (add1 (add1 x)))`?
- *A* $lambda\ expression$*, which is similar to a mathematical function.*
- What's the answer of

```
(define add2 (lambda (x) (add1 (add1 x))))
(add2 (add1 3))
```

  ?
- *6.*

λ ?

- How does `(add2 (add1 3))` work?
- *We first ask the question: what is `(add1 3)`?*

λ ?

- How does `(add2 (add1 3))` work?
- *We first ask the question: what is `(add1 3)`?*
- What is the answer of `(add1 3)`?
- *4.*

λ ?

- How does `(add2 (add1 3))` work?
- *We first ask the question: what is `(add1 3)`?*
- What is the answer of `(add1 3)`?
- *4.*
- What's the answer of `(add2 4)` then?
- *It becomes `((lambda (x) (add1 (add1 x))) 4)`.*

λ ?

- How does `(add2 (add1 3))` work?
- *We first ask the question: what is `(add1 3)`?*
- What is the answer of `(add1 3)`?
- *4.*
- What's the answer of `(add2 4)` then?
- *It becomes `((lambda (x) (add1 (add1 x))) 4)`.*
- Then?
- *We substitute x for 4 in the $lambda\ expression$.*

λ ?

- How does `(add2 (add1 3))` work?
- *We first ask the question: what is `(add1 3)`?*
- What is the answer of `(add1 3)`?
- *4.*
- What's the answer of `(add2 4)` then?
- *It becomes `((lambda (x) (add1 (add1 x))) 4)`.*
- Then?
- *We substitute x for 4 in the $lambda\ expression$.*
- What will we get then?
- *`(add1 (add1 4))`*

λ ?

- How does `(add2 (add1 3))` work?
- *We first ask the question: what is `(add1 3)`?*
- What is the answer of `(add1 3)`?
- *4.*
- What's the answer of `(add2 4)` then?
- *It becomes `((lambda (x) (add1 (add1 x))) 4)`.*
- Then?
- *We substitute x for 4 in the $lambda\ expression$.*
- What will we get then?
- *`(add1 (add1 4))`*
- Is that how we get 6?
- *Yes.*

cond?

- How does the following definition work out?

```
(define one?
  (lambda (x)
    (cond
      ((zero? (sub1 x)) #t)
      (else #f))))
```

- *We'll figure it out soon™.*

cond?

- How does the following definition work out?

```
(define one?
  (lambda (x)
    (cond
      ((zero? (sub1 x)) #t)
      (else #f))))
```

- *We'll figure it out soon™.*
- What's the answer of `(one? 1)`?
- `#t`.

cond?

- How does the following definition work out?

```
(define one?
  (lambda (x)
    (cond
      ((zero? (sub1 x)) #t)
      (else #f))))
```

- *We'll figure it out soon™.*
- What's the answer of `(one? 1)`?
- `#t`.
- What's the answer of `(one? 2)`?
- `#f`.

cond?

```
(define one?
  (lambda (x)
    (cond
      ((zero? (sub1 x)) #t)
      (else #f))))
```

- How does the above definition work out?