Andrés Ricardo Garza Vela

# Lonnrot Scheme

SIGNATURE

JUNE 2, 2021

A report written for the course of Compiler Design at ITESM

# Contents

# About the Lonnrot Language

## Language Overview

Lonnrot Scheme is an implementation of a subset of Scheme written in Racket. As such, it can be described as a **strict**, **call-by-value** language with **dynamic typing**. It has *atomic* data such as **integers**, **chars** and **booleans**, *structured* data such as **strings**, **boxes** and **cons cells** and, last but not least, it has *first class functions* in the form of **lambdas**. Core forms such as `let` and `if`, as well as arithmetic primitives such as addition and substraction are provided to manipulate the aforementioned data types, and are discussed at greater lengths in the rest of the report.

## Errors in Lonnrot

Given Lonnrot Scheme's nature as a dynamic language and given that it targets x86, errors are mostly **runtime exceptions**.

There is an interesting exception with regards to strings: since as of the time of this writing the only way to generate a string is through a string literal, we can get away with a bit of static analysis when using `string-ref`: we can compute the length of the string at hand and as such an out-of-bounds error is technically static (which is, admittedly, a hack).

Type checking specifics are given in subsequent chapters, but suffice it to say that you may stumble upon one of these types of errors when using Lonnrot (broadly and obviously speaking):

- **Syntactic:** Some core forms expect a distinct sequence of lists. For example, `define` expects to have a list of symbols as its second element; `cond` expects a finalizing `else` clause, `letrec` expects its second element to be a list of 2-lists (i.e. the bindings).

- **Semantic:** Primitives such as addition and substraction expect integers, whereas others may require characters. Conditional forms (i.e. `if` and `cond`) are usually not a problem in this regard, everything that is not `#f` is considered **true**.

- **Memory:** Lonnrot does not implement garbage collection, so it is completely possible to have a segfault. That being said, Lonnrot does implement `tail-call optimization`, which greatly saves stack space. However, for example, if you had an enormous list, you’d eventually run out of memory.

# Part I

# Project Documentation

# 1
# The Front End

In this chapter we go over the basic structural elements of Lonnrot Scheme: its lexicon and its syntax. More importantly, we describe the way these concepts map to the actual implementation of the project, that is, we describe how we can express these items as **parser combinators** for use with `megaparsack`.

Although it is common to express lexemes in a language as regular expressions, an interesting insight of Chomsky's Hierarchy is the fact that given that Context Free Languages subsume Regular Languages we can also express these as grammars (albeit simple ones).

Indeed the idea of parser combinators, I find, is not that different in spirit from that of Thompson's Algorithm to convert regex to NFAs

This might seem a gratuitous observation, but it's not: it allows for expressing atomic data as **parser combinators**. Building a parser for a language is, consequently, building parsers for all types of simple, more easily discernible data, and putting these together. We present thus the core lexemes of our language as a grammar.

## 1.1 Lexicon

```
⟨datum⟩        ::= ⟨boolean⟩
                 | ⟨character⟩
                 | ⟨variable⟩
                 | ⟨string⟩
                 | ⟨integer⟩
                 | ⟨list⟩

⟨variable⟩     ::= ⟨initial⟩ ⟨subsequent⟩*

⟨initial⟩      ::= ⟨letter⟩ | '!' | '$' | '&' | '*' | '/' | ':' | '<' | '=' |
                   '>' | '?' | '~' | '_' | '^'

⟨subsequent⟩   ::= ⟨initial⟩ | ⟨digit⟩ | '.' | '+' | '-' | '@'

⟨letter⟩       ::= a | ... | z | A | ... | Z

⟨boolean⟩      ::= #t | #f

⟨digit⟩        ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

⟨integer⟩      ::= ⟨digit⟩+

⟨char⟩         ::= '#\' ⟨any-character⟩

⟨list⟩         ::= (⟨datum⟩*)
                 | (⟨datum⟩+ . ⟨datum⟩)
```

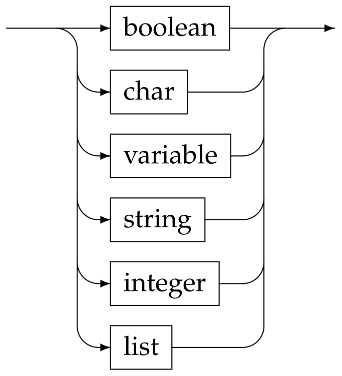
## 1.2 Syntax

| | | |
|---|---|---|
| ⟨*program*⟩ | ::= | ⟨*form*⟩* |
| ⟨*form*⟩ | ::= | ⟨*definition*⟩ \| ⟨*expression*⟩ |
| ⟨*definition*⟩ | ::= | (`define` ⟨*formals*⟩ ⟨*expression*⟩+) |
| ⟨*lambda*⟩ | ::= | (`lambda` ⟨*formals*⟩ ⟨*expression*⟩+) |
| ⟨*expression*⟩ | ::= | ⟨*constant*⟩ |
| | \| | ⟨*variable*⟩ |
| | \| | (`quote` ()) |
| | \| | (`if` ⟨*expression*⟩ ⟨*expression*⟩ ⟨*expression*⟩) |
| | \| | (`cond` ⟨*clause*⟩* ⟨*else-clause*⟩) |
| | \| | (`let` (⟨*binding*⟩) ⟨*expression*⟩+) |
| | \| | (`letrec` (⟨*fix-bind*⟩*) ⟨*expression*⟩+) |
| | \| | ⟨*lambda*⟩ |
| | \| | ⟨*application*⟩ |
| ⟨*clause*⟩ | ::= | (⟨*expression*⟩ ⟨*expression*⟩+) |
| ⟨*else-clause*⟩ | ::= | (**else** ⟨*expression*⟩+) |
| ⟨*binding*⟩ | ::= | (⟨*variable*⟩ ⟨*expression*⟩) |
| ⟨*fix-bind*⟩ | ::= | (⟨*variable*⟩ ⟨*lambda*⟩) |
| ⟨*constant*⟩ | ::= | ⟨*boolean*⟩ \| ⟨*number*⟩ \| ⟨*char*⟩ \| ⟨*string*⟩ \| ⟨*list*⟩ |
| ⟨*formals*⟩ | ::= | (⟨*variable*⟩+) |
| ⟨*application*⟩ | ::= | (⟨*variable*⟩ ⟨*expression*⟩*) |

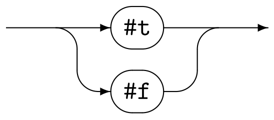
## 1.3 Syntax Diagrams

### 1.3.1 Tokens

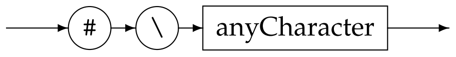
*datum*



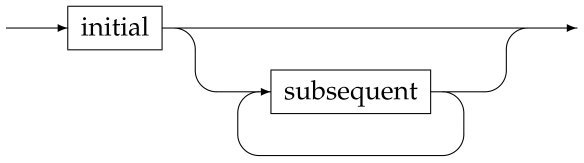
*boolean*



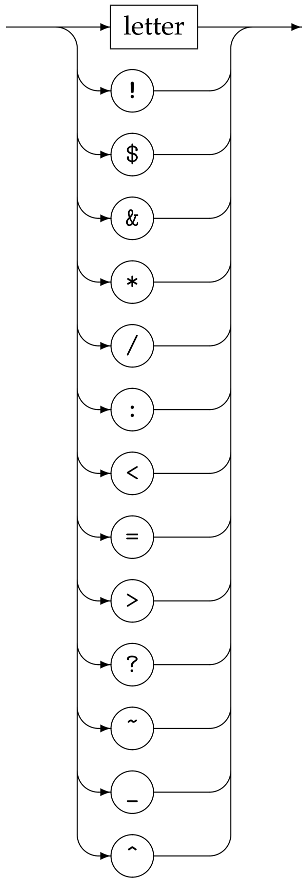
*char*



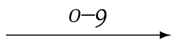
*variable*



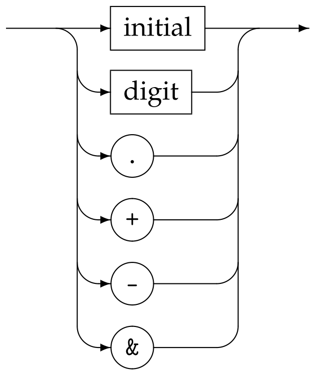
*initial*



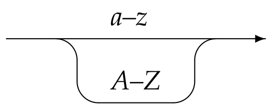
*digit*



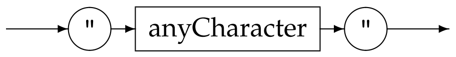
*subsequent*



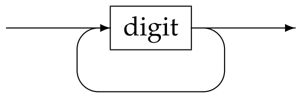
*letter*



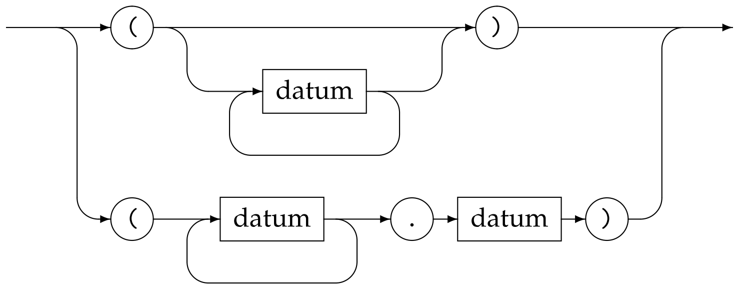
*string*



*integer*

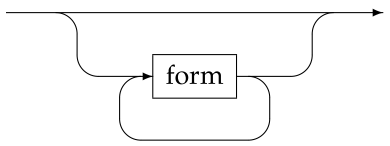


*list*

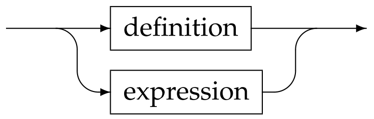


### 1.3.2 Grammar

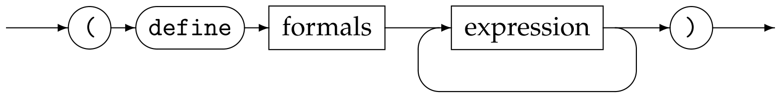
*program*



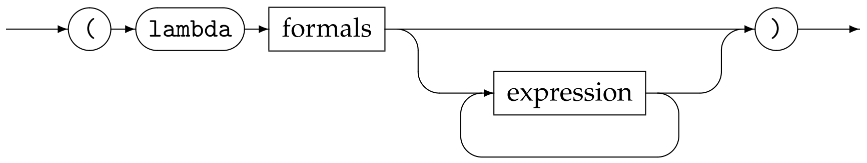
*form*



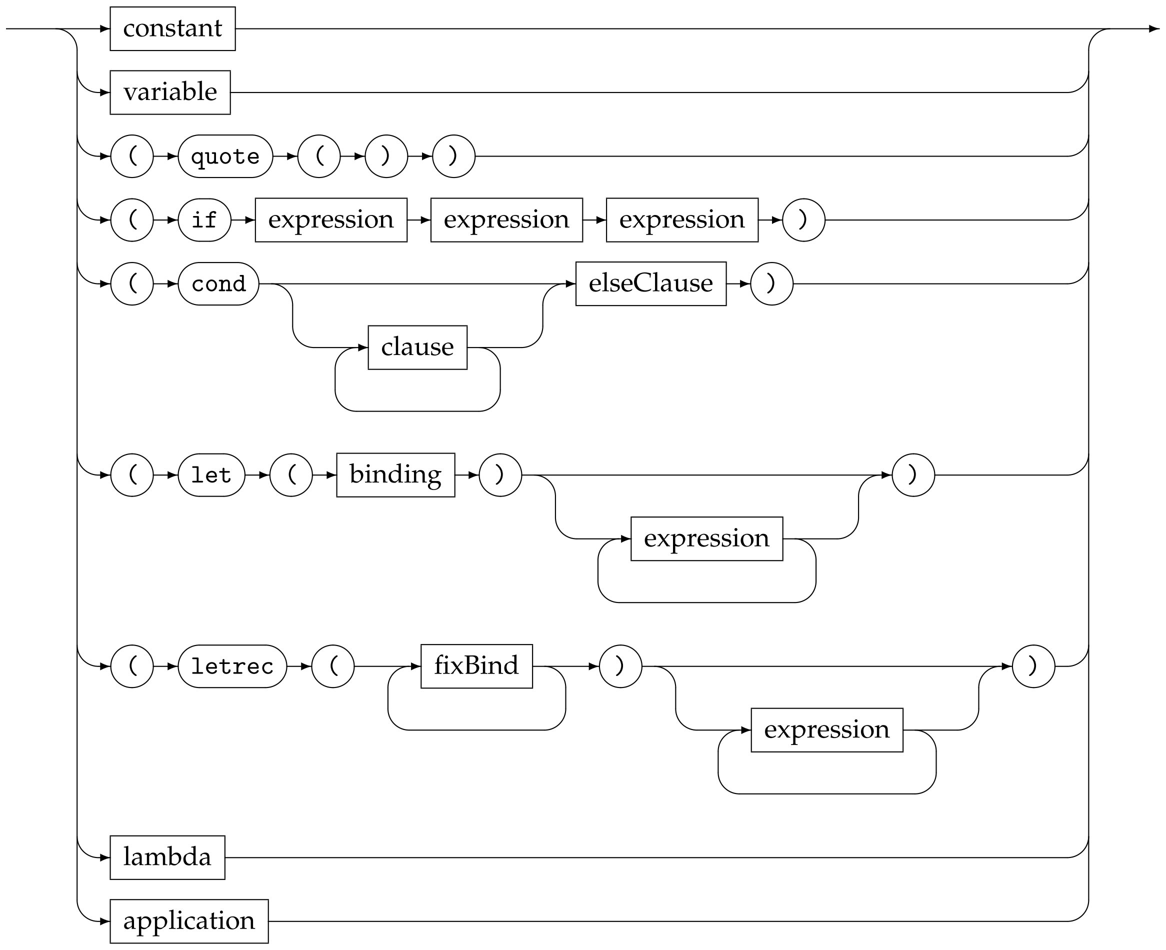
*definition*



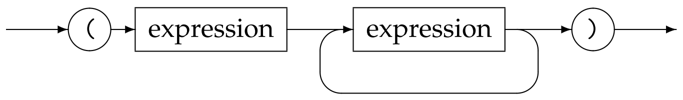
*lambda*



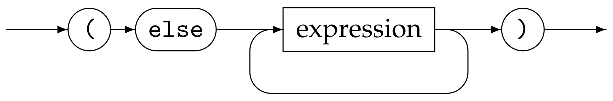
*expression*



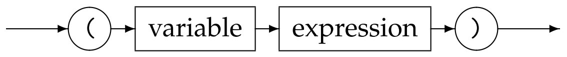
*clause*



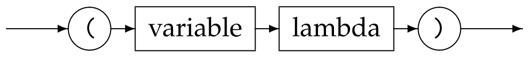
*elseClause*



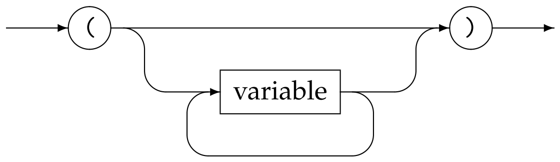
*binding*



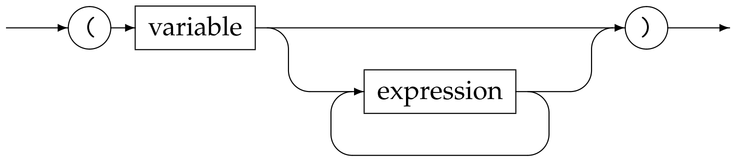
*fixBind*



*formals*



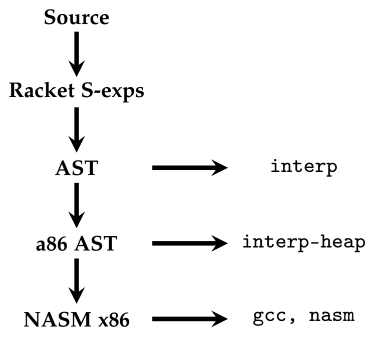
*application*



# 2

# Code Transformations

An overview of the transformations...



From parsing to compilation, Lonnrot Scheme code undergoes several transformations before being either evaluated or simply transformed into NASM x86 code. In this chapter, we explore the steps involved in each stage, the multiple **intermediate representations** used, their characteristics and how these map to the execution of a program.

## 2.1 Parsing

Mostly for pedagogical purposes

The first stage of compilation is **parsing**, which in the case of Lonnrot Scheme actually consists of two passes. First, the source (a *string*), is processed by a parser written in `megaparsack`. By this point, from Chapter 1, we know that the result of this is a singular *s-expression*: a `begin` form. What now?

The next step in the process, found in `parse.rkt`, takes that form and uses pattern-matching to recursively produce an AST representation. Possible nodes can be found in `ast.rkt`, but the key takeaway is this: a program is a `Prog` Racket struct, whose fields are a list of definitions (`Defs`) and a single expression.

## 2.2 Desugaring

Once we have our AST we could technically traverse it and evaluate the results, as is the case of many interpreters. But we can also traverse it to pick out certain nodes, or even *transform it*. The latter is usually done in service of *desugaring*, which simply means that we can transform certain nodes of our tree into **core forms**.

This technique is used in Lonnrot in several ways: we can desugar `cond` into a sequence of `if`s, we desugar `list` into a sequence of `cons` cells, but most importantly we can desugar function definitions into a big `letrec`, which gives Lonnrot Scheme the property of being able to have (1) mutually recursive functions and, generally (2) functions that refer to functions defined later in the source.

## 2.3 The Environment

In a typical Lisp interpreter, we can represent variable bindings through a structure called an *environment*, which is an **assoc-list**.. In some ways this may seem somewhat inefficient, but it plays to the **lexically scoped semantics** of the language: it allows *shadowing* since we extend the environment from the front. In other words, by traversing the environment, you'll reach the latest definition of a given variable. A variable lookup, naturally, consists of traversing the list, checking the `car` of each element.

An assoc-list, or association list, is simply a list of pairs, where the head (or car) of each element is a variable and the tail (or cdr, rest) of each element is the value associated to that variable

The big insight behind this behavior is that, during compilation, we can represent stack memory as a list. We do have to bear in mind a key difference: we don't know the value of variables during compilation. How can we associate a value to a variable in this case? Well, **we don't** because we **can't**; we lack information. What we do know is that we introduce bindings with `let`, and we do have the structure of the code at compile-time. Therefore, we can statically calculate the distance at which a variable's binding is supposed to be from the top of the stack at runtime. In that sense, while a `lookup` in an interpreter simply traverses an environment and returns a value, in compilation a `lookup` consists of traversing the environment and returning *an index*.

# 3
# Execution

Once we have translated code what is left is evaluating it. In this chapter, we discuss the implications of evaluating Lonnrot code: type checking, runtime operations and how what concessions have to be made to represent an environment in a setting such as x86 or a register-based virtual machine.

## 3.1 Semantics

### 3.1.1 Type Tags

To be able to perform type-checking we must have a mechanism to distinguish values at runtime. To that end, we use **type tags**. Under that premise, values are either **immediates** or **pointers**. The following table outlines the hierarchy and the corresponding tags used

Immediates are called that way because they represent data that is storable in a machine word.

| Immediates (end in 000) | |
|---|---|
| **Type** | **Tag** |
| Integer | 0000 0000 |
| Char | 0000 1000 |
| True | 0001 1000 |
| False | 0011 1000 |
| EOF | 0101 1000 |
| Void | 0111 1000 |
| Empty | 1001 1000 |

| Pointers (do not end in 000) | |
|---|---|
| **Type** | **Tag** |
| Box | 001 |
| Pair (Cons) | 010 |
| Strings | 011 |
| Functions | 100 |

To type check, then, is to emit code that:

- Moves a value to a scratch register (`r9`)
- `ands` this with a type mask (e.g. the type mask for an int is `1111`)
- Compares the result with the type tag
- Generates a `jne` code that jumps to the runtime exception handler.

Type tagging raises the issue of compiler correctness due to limitations in representation. In Lonnrot Scheme *there are integer overflows and underflows*, and the behavior is the same as that of, say, C. Likewise, it speaks to some of the characteristics of the x86 architecture. The most obvious example of this is pointer tagging: choosing to represent pointers as values with the last three bits set to things other than zero is not arbitrary. This is because x86 is *byte-addressable*, meaning that it is not possible for us to accidentally "step on" another address by fiddling with the last three bits. We have thus the ability to represent 8 different types of data with pointers.

Actually, the mere fact of storing integers within a machine word without any implementation of bignums also does this, but alas.

In any case, further information about which primitives accept which type of data is given in the Quick Reference 5.

### 3.1.2 Function Calling

Another important part of discussing the language's semantics is the issue of *how* functions are evaluated. Generally, there are three main issues at hand: (1) do we evaluate *all* function arguments before jumping to the function's code? (2) Which argument do we evaluate first? Should we do this left to right? (3) Do we check if the function exists before or after evaluating the arguments?

To cut straight to the chase, the answer to those questions is: Lonnrot Scheme evaluates all of the arguments before the jump, and it does so before checking if the function exists. Regarding question number two, there is no particularly defined way to do this in

This is called, quite appropriately, **eval-apply** semantics of function application

Scheme, generally. However Lonnrot Scheme evaluates arguments **left to right**.

## 3.2 Runtime and Memory

The runtime allows us to perform certain operations that would otherwise require lots of assembly code. The prime example of this is printing results, where we piggy-back on `printf` and a couple of helper functions that allow for special formatting depending on the type of data to be represented. This is especially handy because of type tagging: we can delegate the bit shifting required to produce an actual value to the runtime instead of emitting more code ourselves (even if it is possible to do so).

Equally important though is the fact that the runtime allows us to allocate heap memory through the use of `malloc`. This should make it apparent why the discussion on representing memory during compilation was so short: because we are already thinking in terms of the actual hardware from the beginning! When we generate code, we have to think about how we alter `rsp` and `rbx` (our heap pointer). So, in lieu of reproducing what can be found in tons of x86 manuals, a brief description of the core elements of the x86 architecture that come into play in our compiler are listed below:

| **Register** | **Role** |
|---|---|
| rax | Stores the result of intermediate and final computations |
| rbx | Stores the heap pointer |
| rcx | Scratch register, used to store the size of a closure |
| rdx | Used to pass the heap pointer to the runtime before finishing |
| r8 | Scratch register (used when compiling binary primitives) |
| r9 | Scratch register (used for intermediate values in assertions) |
| rsp | Stores the stack pointer |
| rdi | By SysV ABI convention, this is the first argument register |

# 4 Testing

Since we produce an executable, our output goes to `stdout`. This makes it somewhat roundabout to try and use a particular language to test the outputs. Rather, we use `bash` to check if the produced results are correct.

## 4.1 Example Tests

### Fibonacci with cond

```
(define (fib n)
  (cond ((zero? n) 0)
        ((= n 1)   1)
        (else
         (+ (fib (sub1 n))
            (fib (- n 2))))))

(fib 10)
```

### Tail-call Fibonacci

```
(letrec ((fib (lambda (n)
               (if (eq? n 0)
                   0
                   (if (eq? n 1)
                       1
                       (+ (fib (sub1 n))
                          (fib (- n 2)))))))

          (fib-tail
            (lambda (x f s)
              (if (eq? x 0)
                  f
                  (fib-tail (sub1 x) s (+ f s))))))

    (fib-tail 10 0 1))
```

## Normal Factorial

```
(define (! n)
  (if (zero? n)
      1
      (* n (! (sub1 n)))))

(! 5)
```

## Tail-call Factorial

```
;; !: Int -> Int
;; ! takes an integer and returns and computes its factorial
(define (!tc n acc)
 (if (zero? n)
     acc
     (!tc (sub1 n) (* acc n))))

(!tc 10 1)
```

## Find

```
(find 11 (list 1 2 3 10 4))
```

## List

```
(list)
```

## Map

```
(map (lambda (x) (+ x 4)) (list 1 2 3))
```

## Sort

```
(sort (list 8 3 6 5 0 2 9))
```

## Standard Library Test

```
(let ((l (append (list 1 2 3) (list 4 5 6))))
    (filter (lambda (x) (< x 5)) l))
```

## Mutually Recursive Lambdas

```
(define (even? n)
  (if (zero? n)
      #t
      (odd? (sub1 n))))

(define (odd? n)
  (if (zero? n)
      #f
      (even? (sub1 n))))

(even? 1001)
(odd? 1000)
```

## String Length

```
(string-length "hello␣world")
```

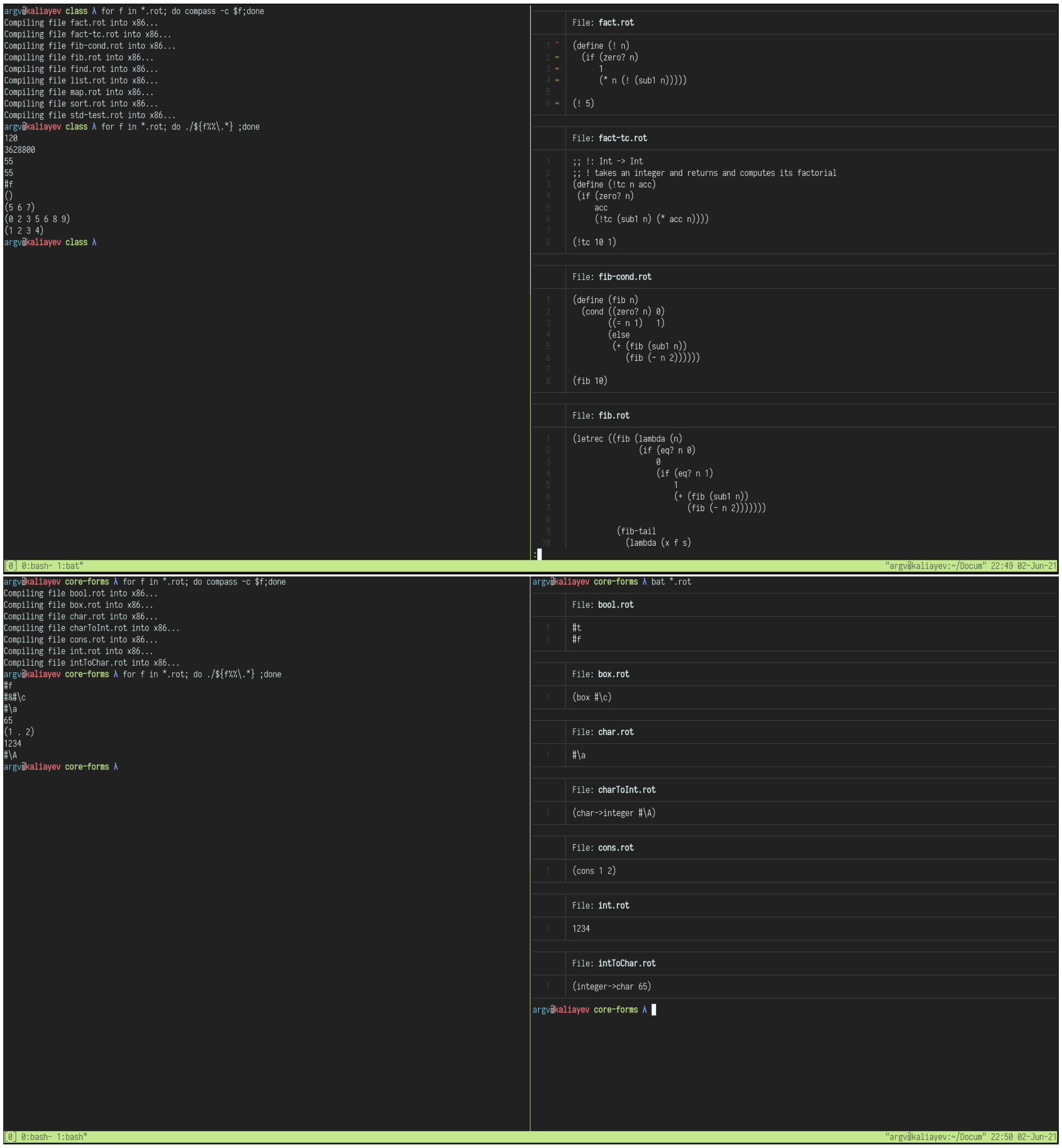
## String Ref

```
(displayln (string-ref 5 "hello"))
```

## Multiple relational ops with begin

```
(let ((t (char->integer #\t)))
 (let ((f (char->integer #\f)))
  (begin
    (if (< 1 2) (write-byte t)  (write-byte f))
    (if (< 2 1) (write-byte t)  (write-byte f))
    (if (> 4 3) (write-byte t) (write-byte f))
    (if (> 3 3) (write-byte t) (write-byte f))
    (if (= 3 3) (write-byte t) (write-byte f)))))
```

## 4.2 Example Test Outputs



```
argv@kaliayev class λ for f in *.rot; do compass -c $f;done
Compiling file fact.rot into x86...
Compiling file fact-tc.rot into x86...
Compiling file fib-cond.rot into x86...
Compiling file fib.rot into x86...
Compiling file find.rot into x86...
Compiling file list.rot into x86...
Compiling file map.rot into x86...
Compiling file sort.rot into x86...
Compiling file std-test.rot into x86...
argv@kaliayev class λ for f in *.rot; do ./${f%%\.*} ;done
120
3628800
55
55
#f
()
(5 6 7)
(0 2 3 5 6 8 9)
(1 2 3 4)
argv@kaliayev class λ
```

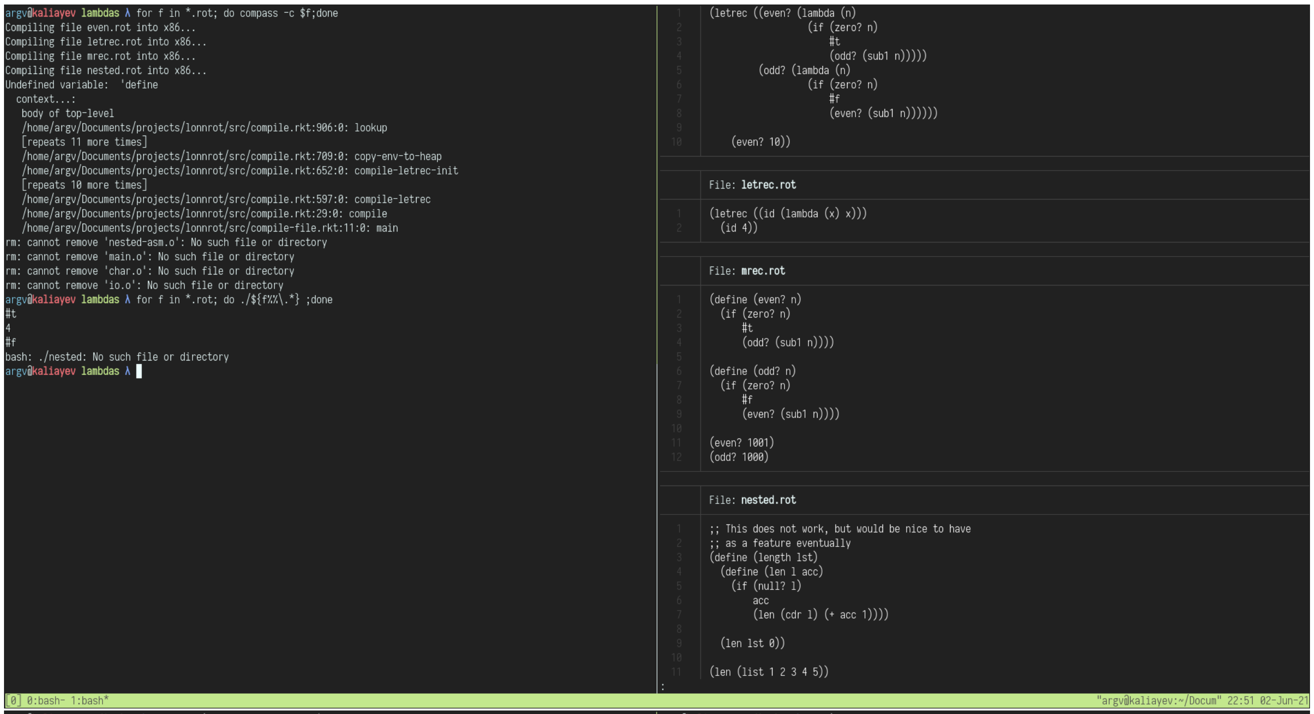
```
File: fact.rot

(define (! n)
  (if (zero? n)
      1
      (* n (! (sub1 n)))))

(! 5)

File: fact-tc.rot

;; !: Int -> Int
;; ! takes an integer and returns and computes its factorial
(define (!tc n acc)
 (if (zero? n)
     acc
     (!tc (sub1 n) (* acc n))))

(!tc 10 1)

File: fib-cond.rot

(define (fib n)
  (cond ((zero? n) 0)
        ((= n 1)   1)
        (else
         (+ (fib (sub1 n))
            (fib (- n 2))))))

(fib 10)

File: fib.rot

(letrec ((fib (lambda (n)
               (if (eq? n 0)
                   0
                   (if (eq? n 1)
                       1
                       (+ (fib (sub1 n))
                          (fib (- n 2)))))))

          (fib-tail
            (lambda (x f s)
:
```

```
[0] 0:bash- 1:bat*                                      "argv@kaliayev:~/Docum" 22:49 02-Jun-21
```

```
argv@kaliayev core-forms λ for f in *.rot; do compass -c $f;done
Compiling file bool.rot into x86...
Compiling file box.rot into x86...
Compiling file char.rot into x86...
Compiling file charToInt.rot into x86...
Compiling file cons.rot into x86...
Compiling file int.rot into x86...
Compiling file intToChar.rot into x86...
argv@kaliayev core-forms λ for f in *.rot; do ./${f%%\.*} ;done
#f
#&#\c
#\a
65
(1 . 2)
1234
#\A
argv@kaliayev core-forms λ
```

```
argv@kaliayev core-forms λ bat *.rot
File: bool.rot

#t
#f

File: box.rot

(box #\c)

File: char.rot

#\a

File: charToInt.rot

(char->integer #\A)

File: cons.rot

(cons 1 2)

File: int.rot

1234

File: intToChar.rot

(integer->char 65)

argv@kaliayev core-forms λ
```

```
[0] 0:bash- 1:bash*                                     "argv@kaliayev:~/Docum" 22:50 02-Jun-21
```



```
argv@kaliayev lambdas λ for f in *.rot; do compass -c $f;done
Compiling file even.rot into x86...
Compiling file letrec.rot into x86...
Compiling file mrec.rot into x86...
Compiling file nested.rot into x86...
Undefined variable:  'define
  context...:
   body of top-level
   /home/argv/Documents/projects/lonnrot/src/compile.rkt:906:0: lookup
   [repeats 11 more times]
   /home/argv/Documents/projects/lonnrot/src/compile.rkt:709:0: copy-env-to-heap
   /home/argv/Documents/projects/lonnrot/src/compile.rkt:652:0: compile-letrec-init
   [repeats 10 more times]
   /home/argv/Documents/projects/lonnrot/src/compile.rkt:597:0: compile-letrec
   /home/argv/Documents/projects/lonnrot/src/compile.rkt:29:0: compile
   /home/argv/Documents/projects/lonnrot/src/compile-file.rkt:11:0: main
rm: cannot remove 'nested-asm.o': No such file or directory
rm: cannot remove 'main.o': No such file or directory
rm: cannot remove 'char.o': No such file or directory
rm: cannot remove 'io.o': No such file or directory
argv@kaliayev lambdas λ for f in *.rot; do ./${f%%\.*} ;done
#t
4
#f
bash: ./nested: No such file or directory
argv@kaliayev lambdas λ
```

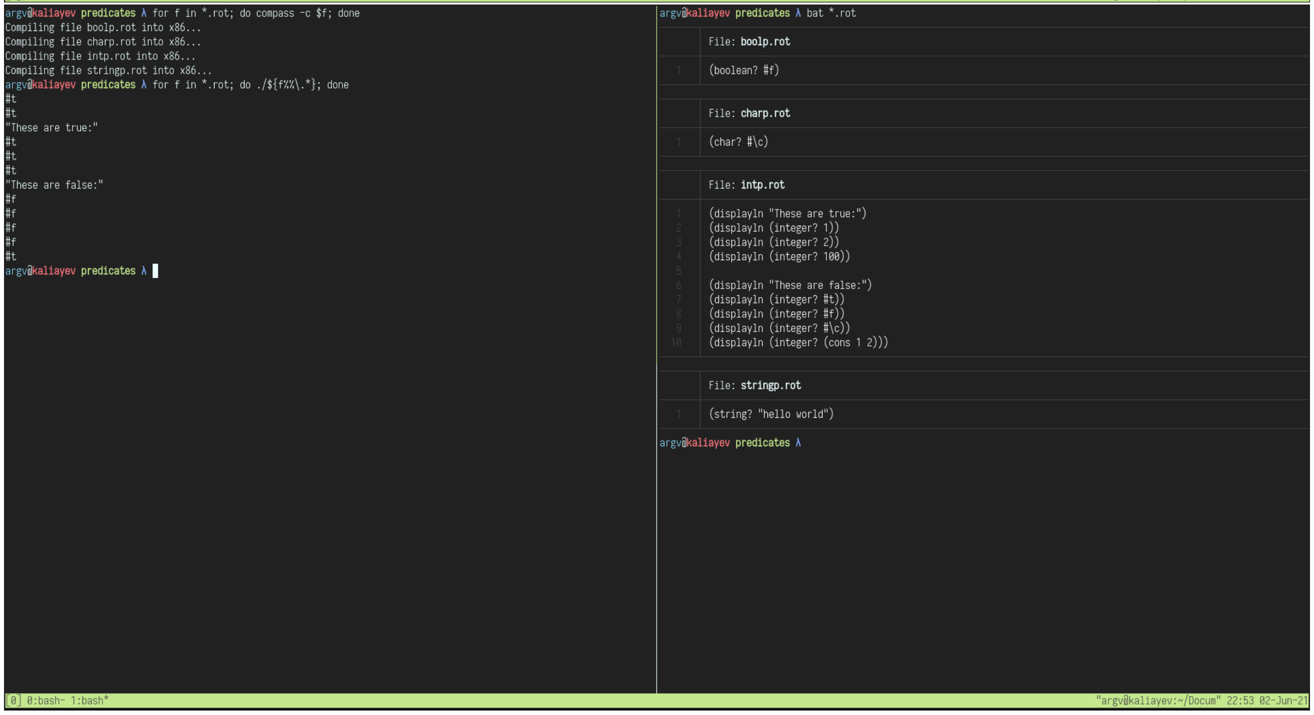
```
(letrec ((even? (lambda (n)
                  (if (zero? n)
                      #t
                      (odd? (sub1 n)))))
         (odd? (lambda (n)
                  (if (zero? n)
                      #f
                      (even? (sub1 n))))))

    (even? 10))
```

```
File: letrec.rot

(letrec ((id (lambda (x) x)))
  (id 4))
```

```
File: mrec.rot

(define (even? n)
  (if (zero? n)
      #t
      (odd? (sub1 n))))

(define (odd? n)
  (if (zero? n)
      #f
      (even? (sub1 n))))

(even? 1001)
(odd? 1000)
```

```
File: nested.rot

;; This does not work, but would be nice to have
;; as a feature eventually
(define (length lst)
  (define (len l acc)
    (if (null? l)
        acc
        (len (cdr l) (+ acc 1))))

  (len lst 0))

(len (list 1 2 3 4 5))
:
```

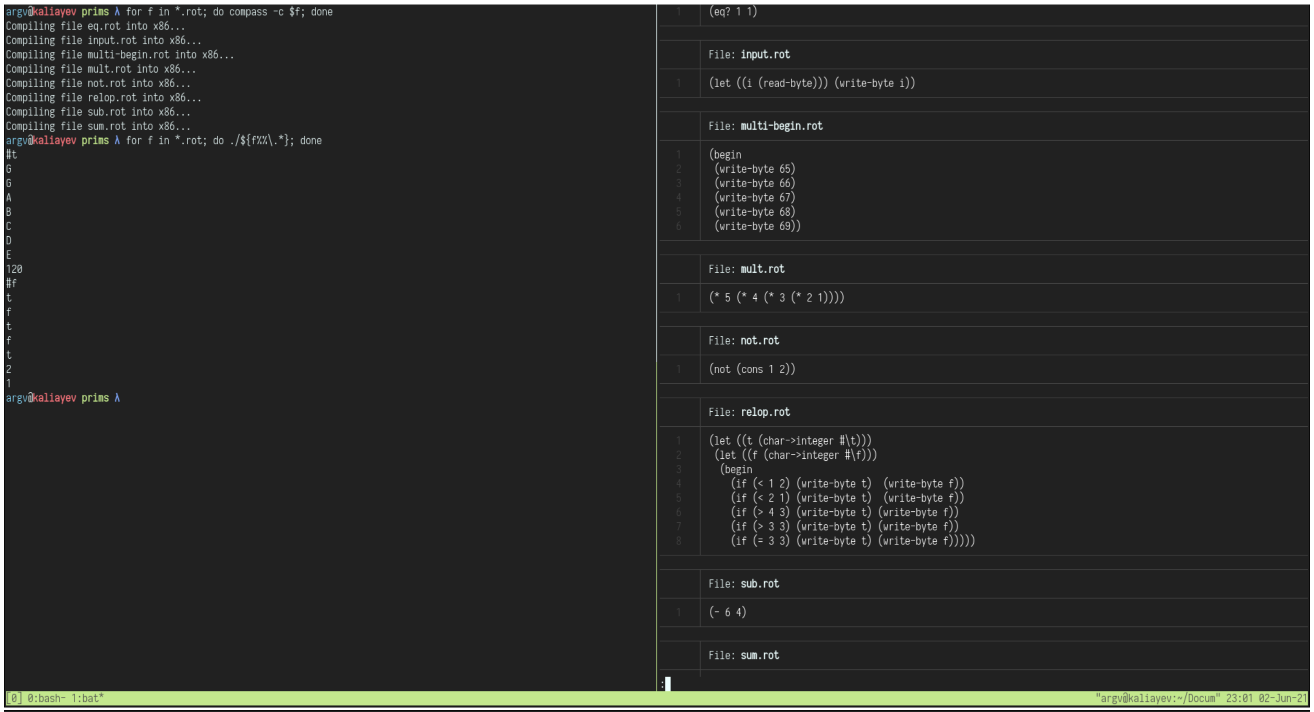
```
[0] 0:bash- 1:bash*                                   "argv@kaliayev:~/Docum" 22:51 02-Jun-21
```



```
argv@kaliayev predicates λ for f in *.rot; do compass -c $f; done
Compiling file boolp.rot into x86...
Compiling file charp.rot into x86...
Compiling file intp.rot into x86...
Compiling file stringp.rot into x86...
argv@kaliayev predicates λ for f in *.rot; do ./${f%%\.*}; done
#t
#t
"These are true:"
#t
#t
#t
"These are false:"
#f
#f
#f
#f
#t
argv@kaliayev predicates λ
```

```
argv@kaliayev predicates λ bat *.rot

File: boolp.rot

(boolean? #f)

File: charp.rot

(char? #\c)

File: intp.rot

(displayln "These are true:")
(displayln (integer? 1))
(displayln (integer? 2))
(displayln (integer? 100))

(displayln "These are false:")
(displayln (integer? #t))
(displayln (integer? #f))
(displayln (integer? #\c))
(displayln (integer? (cons 1 2)))

File: stringp.rot

(string? "hello world")

argv@kaliayev predicates λ
```

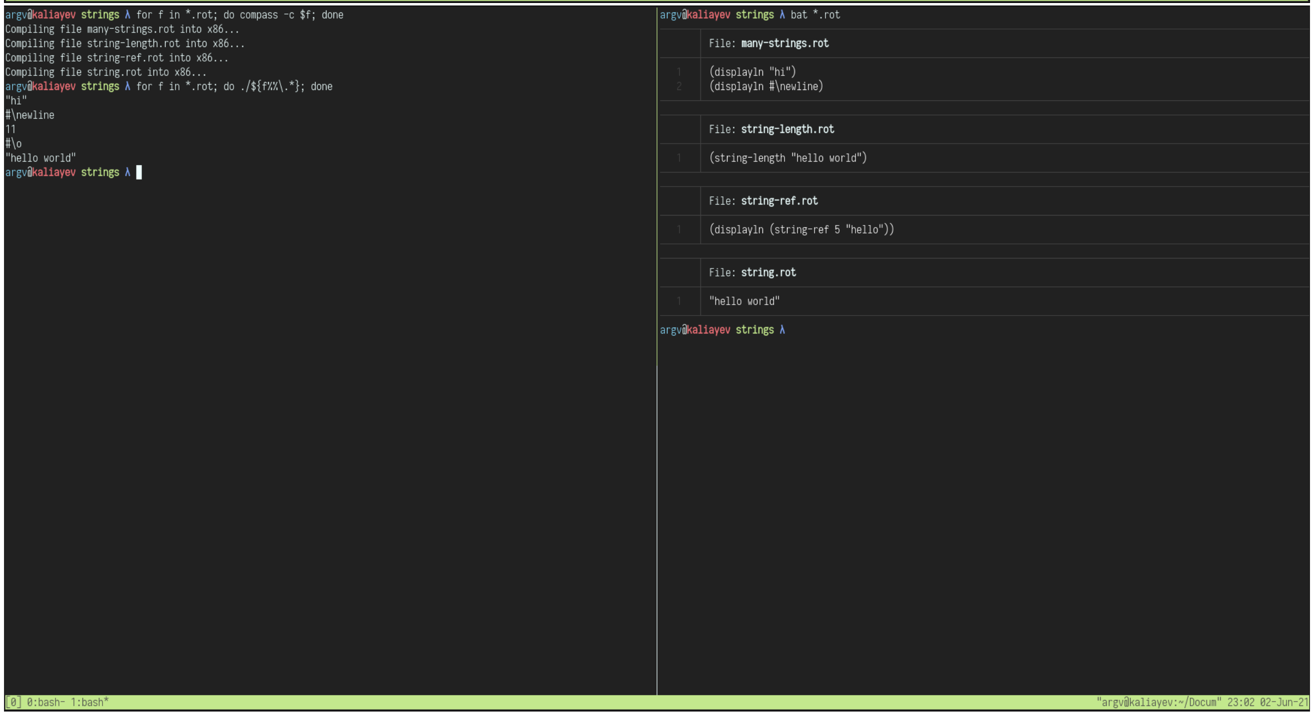
```
[0] 0:bash- 1:bash*                                   "argv@kaliayev:~/Docum" 22:53 02-Jun-21
```



```
argv@kaliayev prims λ for f in *.rot; do compass -c $f; done
Compiling file eq.rot into x86...
Compiling file input.rot into x86...
Compiling file multi-begin.rot into x86...
Compiling file mult.rot into x86...
Compiling file not.rot into x86...
Compiling file relop.rot into x86...
Compiling file sub.rot into x86...
Compiling file sum.rot into x86...
argv@kaliayev prims λ for f in *.rot; do ./${f%%\.*}; done
#t
G
G
A
B
C
D
E
120
#f
t
f
t
f
t
2
1
argv@kaliayev prims λ
```

```
(eq? 1 1)

File: input.rot

(let ((i (read-byte))) (write-byte i))

File: multi-begin.rot

(begin
 (write-byte 65)
 (write-byte 66)
 (write-byte 67)
 (write-byte 68)
 (write-byte 69))

File: mult.rot

(* 5 (* 4 (* 3 (* 2 1))))

File: not.rot

(not (cons 1 2))

File: relop.rot

(let ((t (char->integer #\t)))
 (let ((f (char->integer #\f)))
  (begin
    (if (< 1 2) (write-byte t)  (write-byte f))
    (if (< 2 1) (write-byte t)  (write-byte f))
    (if (> 4 3) (write-byte t) (write-byte f))
    (if (> 3 3) (write-byte t) (write-byte f))
    (if (= 3 3) (write-byte t) (write-byte f)))))

File: sub.rot

(- 6 4)

File: sum.rot

:
```

```
[0] 0:bash- 1:bat*                                    "argv@kaliayev:~/Docum" 23:01 02-Jun-21
```



```
argv@kaliayev strings λ for f in *.rot; do compass -c $f; done
Compiling file many-strings.rot into x86...
Compiling file string-length.rot into x86...
Compiling file string-ref.rot into x86...
Compiling file string.rot into x86...
argv@kaliayev strings λ for f in *.rot; do ./${f%%\.*}; done
"hi"
#\newline
11
#\o
"hello world"
argv@kaliayev strings λ
```

```
argv@kaliayev strings λ bat *.rot

File: many-strings.rot

(displayln "hi")
(displayln #\newline)

File: string-length.rot

(string-length "hello world")

File: string-ref.rot

(displayln (string-ref 5 "hello"))

File: string.rot

"hello world"

argv@kaliayev strings λ
```

```
[0] 0:bash- 1:bash*                                   "argv@kaliayev:~/Docum" 23:02 02-Jun-21
```

# Part II

# Reference Manual

# 5
# Quick Reference

## 5.1 Setup

Make sure to have gcc and nasm. Likewise, you should have Racket installed with megaparsack and a86:

```
raco pkg install 'https://github.com/cmsc430/www.git?path=langs#main'
raco pkg install megaparsack
```

Once you have met the required dependencies, clone the repository and run install.sh.

```
git clone https://github.com/argvniyx/lonnrot.git
cd lonnrot && ./install.sh
```

This will symlink compass, a CLI tool that allows to interpret (-i) or compile (-c) Lonnrot files. A Lonnrot file need not be identified with a file extension, but the common convention is to suffix your files with .rot.

## 5.2 Core Forms and Primitives

### void

0-arity primitive, simply produces the value of void. Exists as a result of effectful computation.

### read-byte

0-arity primitive. Reads one keystroke from the user. This uses C's getc function.

### peek-byte

Same as read-byte but ungets the char back onto stdin.

### +

`Integer x Integer -> Integer`
Takes two integers and produces their sum. Unlike other Schemes, this **is not n-ary**.

### -

`Integer x Integer -> Integer`
Takes two integers and produces their difference. Unlike other Schemes, this **is not n-ary**.

### add1

`Integer -> Integer`
Takes an integer $a$ and produces $a + 1$.

### sub1

`Integer -> Integer`
Takes an integer $a$ and produces $a - 1$.

### zero?

`Any -> Boolean`
Takes a value and returns `#t` if it is equal to zero, `#f` if not.

### null?

`Any -> Boolean`
Takes a value and checks if it is the empty list. If is, it returns `#t`, `#f` otherwise.

### integer?

`Any -> Boolean`
Takes a value and checks if it is an integer.

### boolean?

`Any -> Boolean`
Takes a value and checks if it is a boolean.

### char?

`Any -> Boolean`
Takes a value and checks if it is a character.

### eof-object?

`Any -> Boolean`
Checks if the given value is `eof`. There is no file reading in Lonnrot yet, but we can check this, for example, when a user types `C-c` at a prompt from `(read-byte)`.

### string?

`Any -> Boolean`
Takes a value and checks if it is a string.

### string-length

`String -> Integer`
Takes a string and returns its length.

### string-ref

`String x Integer -> Char`
Takes a string and an index and returns the character at the index, or an error if the index is out of bounds.

### eq?

`Any x Any -> Boolean`
Takes two values and checks if they are `eq?`, meaning, if they are exactly the same. This is trivial for immediates, but not so for pointers.

### =

`Integer x Integer -> Boolean`
Same as `eq?` but for integers.

### <

`Integer x Integer -> Boolean`
Returns `#t` if the first argument is strictly lower than the second.

### >

`Integer x Integer -> Boolean`
Returns `#t` if the first argument is strictly greater than the second.

### not

`Any -> Boolean`
If given a truthy value, this returns `#f`, `#t` otherwise.

### char->integer

`Char -> Integer`
Takes an character and returns its codepoint.

### integer->char

`Char -> Integer`
Takes an integer and returns the character it represents.

### write-byte

`Integer -> Void`
Displays a single byte to `stdout`

### displayln

`Any -> Void`
Displays any data given to it to `stdout`

### box

`Any -> Box`
Boxes any value (i.e. creates a reference for it in memory)

### unbox

`Box -> Any`
Returns a value boxed by a Box, or produces an exception if the value given is not a box.

### cons

`Any x Any -> Pair`
Produces a pair.

### list

`Any...-> Pair`
Produces a proper list. It is a shorthand for `(cons (cons (cons ...'())))`

## car

`Pair -> Any`
Returns the first element of a pair or produces an error if its not given a pair.

## cdr

`Pair -> Any`
Returns the second element of a pair (or rest, in the case of a list), or produces an error if not given a pair.

## if

`Any x Any a x Any b -> Either a or b`
Evaluates its first argument and, depending on whether it is true or not, produces value a (if true) or b (when false)

## cond

A sequence of clauses where the first clause with a truthful predicate returns the value of its expression.

```
(cond [(#f ''this is not returned'')]
      [(#t ''but this is'')]
     [else ''we don't even get here!''])
```

## let

Takes a binding, which is a list of a single pair, which binds the value to the symbol given and returns the evaluation of its body (the second argument)

```
  (let ((x 4)) (+ x 4)) -> 8
```

## letrec

Like `let`, but binds a `lambda` to a symbol. See `define`.

## define

Syntactic sugar for `letrec`

```
;; This...
(define (double x) (+ x x))

;; ...is the same as this:
(letrec ((f (lambda (x) (+ x x)))) ...)
```

### lambda

Produces a function pointer, which can then be applied to arguments.

## 5.3 Standard Library

### length

`Pair -> Integer` Takes a list and returns its size.

### append

`Pair x Pair -> Pair`
Takes two lists and returns the list that contains all of the elements of both lists in the order given.

### map

`Lambda x Pair -> Pair`
Returns the list that results of applying the first argument to each element of the list.

### filter

`Lambda x Pair -> Pair`
Returns the list of elements in the second argument that fulfill the predicate of the first argument.

### sort

`List Integer -> List Integer`
Takes a list of integers and sorts them in increasing order

### find

`List x Any -> Any or False`
Looks the second argument in the first. If it finds it, it returns the argument itself, `#f` otherwise.

### and

`Any x Any -> Boolean`
Takes two arguments and returns true if both are truthy (i.e. not false).

## or

```
Any x Any -> Boolean
```
Takes two arguments and returns true if either of them are truthy.

# Conclusion

Lisps are known to be high level, functional languages. They also had a reputation for inneficiency back in their heyday. But things have changed and lots of ground has been covered not only for Lisp but for functional languages in general.

To paraphrase Alan Perlis, lispers know the value of everything and the cost of nothing.

To say the least; even back when Peter Norvig first wrote *Artificial Intelligence: A Modern Approach*he benchmarked Lisp with favorable results

What did not change, however, was the awe-inspiring sense of magic that using Lisp produced. In that sense, learning how all of those nice, quote-unquote *pure* trickle down and become bits is a very enlightening experience. Likewise, the idea of incremental compiler construction is near and dear to me, even if I didn't do it justice. To me, the notion of *language* pervades our profession, and demystifying compilers is a huge leap in the discipline of making and understanding software and advancing our field.

No pun intended

# Appendix

## About the project

Lonnrot Scheme was developed as a final project for a compiler design and implementation course. In this section, we touch upon what we set out to do, how we went about doing it and a brief overview of the project's timeline.

### Purpose, Scope and Inspiration

The purpose of this project was to explore the design and implementation of a minimal language that had *Turing Completeness*. Naturally, however, this is a very broad description: in the case of a functional language, having `lambda` suffices to meet this criterion. In that sense, it is important to note that the project was meant to have other operations, which I outline in the Requirements 5.3. In this section, rather, I'd like to elucidate the scope of the project and talk about what made me try it out.

A euphemism for "justify my shortcomings"

The original idea was for Lonnrot to also include the core of miniKanren. The value proposition was to have miniKanren be compiled rather than interpreted, since it's something that has not been done before. However, life got in the way and I was only able to (marginally) implement the Scheme part of the whole affair. In any case, the inspiration for my workflow was given by Abdulaziz Ghuloum in *An Incremental Approach to Compiler Construction*. The main idea is to write multiple compilers, with each one covering a progressively larger language. This is opposed to the idea of writing one compiler, front to back, for the originally intended language.

Lindsey Kuper [Kup19] talks about this approach and Kent Dybvig's "cousin" approach of writing the compiler back-to-front, making a language progressively more high-level, for those interested.

This has, amongst other implications, the effect of guiding my efforts towards making stuff work for x86. Notably, there are no floats implemented in Lonnrot: since this was a semester long course and parsing, execution and other things were also a priority (as opposed to Dybvig's course), it followed that fiddling with bit representations of floats was a bit beyond the scope of the book at best, and an exercise in frustration at worst. Likewise, error reporting in Lonnrot is weaker compared to other (static) projects in the class, since it would have consumed too much time for marginal gains grade-wise. It is, of course, my wish to keep hacking away at it on the long term, hopefully incorporating things to make it more well-rounded and usable (or at least somewhat clearer for educational purposes; it'd be nice to contribute to the a86 repository).

## Requirements

IN A VERY BROAD SENSE, the project was required to achieve *Turing Completeness*. More specifically, being a functional language, Lonnrot Scheme was required to have at least:

- Binding forms, assignment: `let`, `letrec`
- Conditionals: `if`, `cond`
- Math expressions, logical and relational operators: `+`, `-`, `*`, `<`, `>`, `=`
- Modules, functions: `define`, `letrec`, `lambda`
- Structured element: `cons`, `box`, **strings**

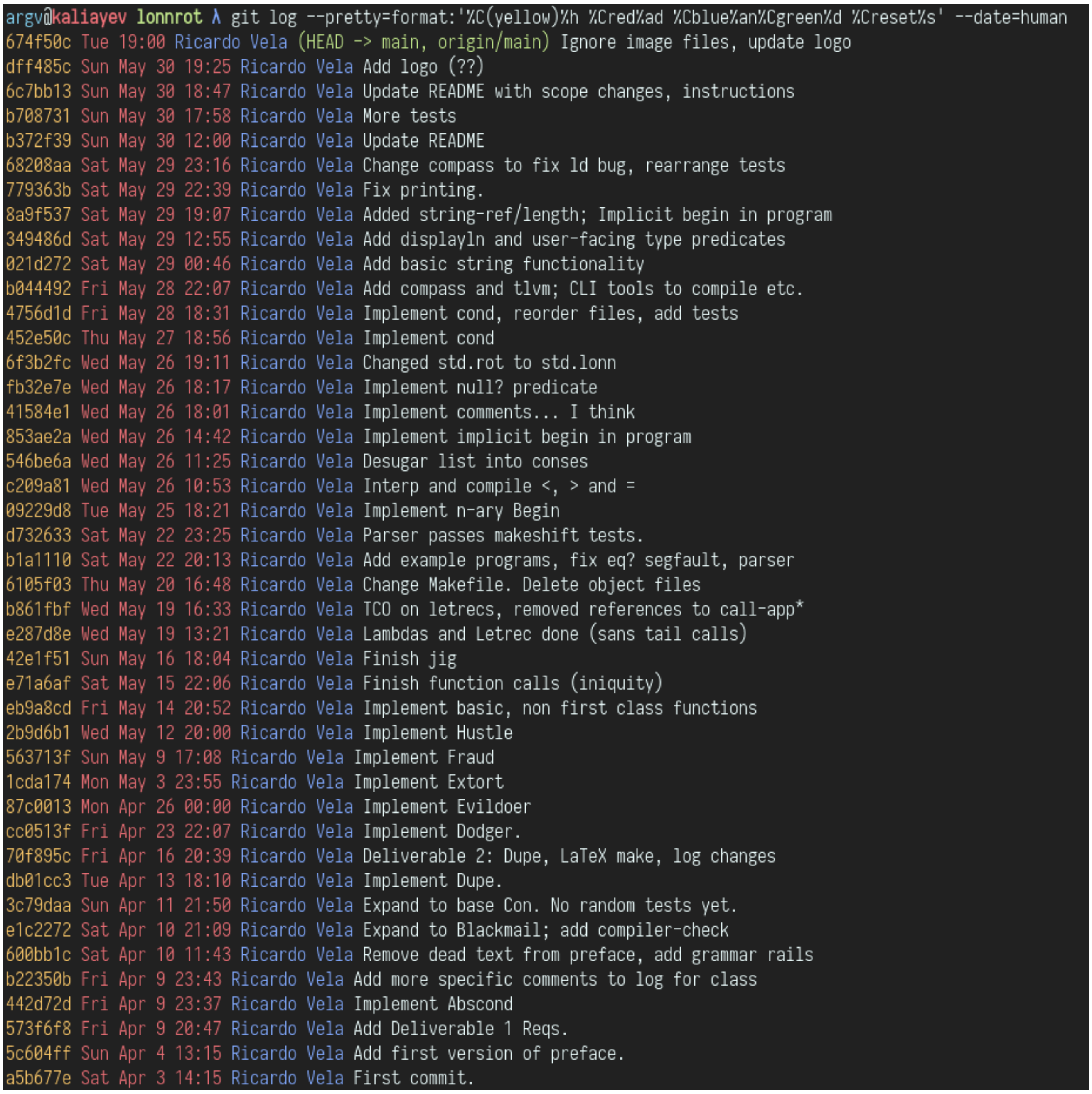
## Weekly Log

AS DESCRIBED IN THE EARLIER APPENDIX, the point of following Ghuloum's and Dybvig's ideas was to iteratively refine and grow the final compiler. As such, each entry of the log below can be understood as the process of, roughly speaking:

- Adding a node to the AST
- Adding a clause to `parse.rkt` and `mparse.rkt`
- Adding appropriate code to `interp.rkt` and `compile.rkt`
- Modifying the runtime if necessary

| Week | Progress |
|---|---|
| Week 1 (April 7th) | Implemented integers and primitives (`add1`, `sub1`, arithmetic `if`) |
| Week 2 (April 14th) | Implemented booleans, boolean `if` |
| Week 3 (April 21st) | Implemented chars |
| Week 4 (April 28th) | Implemented basic byte I/O, runtime exceptions |
| Week 5 (May 5th) | Implemented `let` and binary primitives such as `+` and `-` |
| Week 6 (May 12th) | Implemented inductive data (`box`, `cons`) and non first-class functions |
| Week 7 (May 19th) | Implemented TCO, `letrec` and `lambdas` |
| Week 8 (May 26th) | Implemented desugaring, added standard lib and amenities |
| Week 9 (June 2nd) | Finished documentation, added bells and whistles |

## Commit History



```
argv@kaliayev lonnrot λ git log --pretty=format:'%C(yellow)%h %Cred%ad %Cblue%an%Cgreen%d %Creset%s' --date=human
674f50c Tue 19:00 Ricardo Vela (HEAD -> main, origin/main) Ignore image files, update logo
dff485c Sun May 30 19:25 Ricardo Vela Add logo (??)
6c7bb13 Sun May 30 18:47 Ricardo Vela Update README with scope changes, instructions
b708731 Sun May 30 17:58 Ricardo Vela More tests
b372f39 Sun May 30 12:00 Ricardo Vela Update README
68208aa Sat May 29 23:16 Ricardo Vela Change compass to fix ld bug, rearrange tests
779363b Sat May 29 22:39 Ricardo Vela Fix printing.
8a9f537 Sat May 29 19:07 Ricardo Vela Added string-ref/length; Implicit begin in program
349486d Sat May 29 12:55 Ricardo Vela Add displayln and user-facing type predicates
021d272 Sat May 29 00:46 Ricardo Vela Add basic string functionality
b044492 Fri May 28 22:07 Ricardo Vela Add compass and tlvm; CLI tools to compile etc.
4756d1d Fri May 28 18:31 Ricardo Vela Implement cond, reorder files, add tests
452e50c Thu May 27 18:56 Ricardo Vela Implement cond
6f3b2fc Wed May 26 19:11 Ricardo Vela Changed std.rot to std.lonn
fb32e7e Wed May 26 18:17 Ricardo Vela Implement null? predicate
41584e1 Wed May 26 18:01 Ricardo Vela Implement comments... I think
853ae2a Wed May 26 14:42 Ricardo Vela Implement implicit begin in program
546be6a Wed May 26 11:25 Ricardo Vela Desugar list into conses
c209a81 Wed May 26 10:53 Ricardo Vela Interp and compile <, > and =
09229d8 Tue May 25 18:21 Ricardo Vela Implement n-ary Begin
d732633 Sat May 22 23:25 Ricardo Vela Parser passes makeshift tests.
b1a1110 Sat May 22 20:13 Ricardo Vela Add example programs, fix eq? segfault, parser
6105f03 Thu May 20 16:48 Ricardo Vela Change Makefile. Delete object files
b861fbf Wed May 19 16:33 Ricardo Vela TCO on letrecs, removed references to call-app*
e287d8e Wed May 19 13:21 Ricardo Vela Lambdas and Letrec done (sans tail calls)
42e1f51 Sun May 16 18:04 Ricardo Vela Finish jig
e71a6af Sat May 15 22:06 Ricardo Vela Finish function calls (iniquity)
eb9a8cd Fri May 14 20:52 Ricardo Vela Implement basic, non first class functions
2b9d6b1 Wed May 12 20:00 Ricardo Vela Implement Hustle
563713f Sun May 9 17:08 Ricardo Vela Implement Fraud
1cda174 Mon May 3 23:55 Ricardo Vela Implement Extort
87c0013 Mon Apr 26 00:00 Ricardo Vela Implement Evildoer
cc0513f Fri Apr 23 22:07 Ricardo Vela Implement Dodger.
70f895c Fri Apr 16 20:39 Ricardo Vela Deliverable 2: Dupe, LaTeX make, log changes
db01cc3 Tue Apr 13 18:10 Ricardo Vela Implement Dupe.
3c79daa Sun Apr 11 21:50 Ricardo Vela Expand to base Con. No random tests yet.
e1c2272 Sat Apr 10 21:09 Ricardo Vela Expand to Blackmail; add compiler-check
600bb1c Sat Apr 10 11:43 Ricardo Vela Remove dead text from preface, add grammar rails
b22350b Fri Apr 9 23:43 Ricardo Vela Add more specific comments to log for class
442d72d Fri Apr 9 23:37 Ricardo Vela Implement Abscond
573f6f8 Fri Apr 9 20:47 Ricardo Vela Add Deliverable 1 Reqs.
5c604ff Sun Apr 4 13:15 Ricardo Vela Add first version of preface.
a5b677e Sat Apr 3 14:15 Ricardo Vela First commit.
```

## Libraries and other Tooling

LONNROT SCHEME is written in Racket. However, there are other parts of the project that may warrant a more specific description of the tooling required in the project.

- **Parsing:** `megaparsack` was used for parsing. Notable dependencies include `data/monad` and `data/applicative`, which allow the generation of symbolic expressions upon the successful application of a parser combinator.
- **AST Transformations:** For facilities when dealing with representing x86 instructions with Racket structs, `a86` was used.
- **Runtime:** As stated in the installation instructions in the Reference II, the runtime was written in C, and we depend on `nasm` and `gcc` to compile and link the object files of both a Lonnrot program and the runtime.

  Lastly, Lonnrot Scheme was developed in Arch Linux.

I swear this is not a "btw i use arch" joke

# Bibliography

[AK91] Hassan Aït-Kaci. *Warren's abstract machine. A tutorial reconstruction*. Logic Programming. The MIT Press, 1991.

[Byr15] William E. Byrd. Differences between minikanren and prolog. `http://minikanren.org/minikanren-and-prolog.html`, 2015. Accessed April 3, 2021.

[Dyb09] R. Kent Dybvig. *The Scheme Programming Language*. The MIT Press, fourth edition edition, 2009.

[Fis20] Shriram Krishnamurthi; Benjamin S. Lerner; Joe Gibbs Politz; Kathi Fisler. Programming and programming languages. `https://papl.cs.brown.edu/2019/`, 2020. Accessed April 3, 2021.

[Hem18] Daniel P. Friedman; William E. Byrd; Oleg Kiselyov; Jason Hemann. *The Reasoned Schemer*. The MIT Press, 2nd edition, 2018.

[Kup19] Lindsey Kuper. My first fifteen compilers. `http://minikanren.org/minikanren-and-prolog.html`, 2019. Accessed April 3, 2021.

[MM12] Andreas Jaeger Michael Matz, Jan Hubicka. *System V Application Binary Interface: AMD64 Architecture Processor Supplement*. 0.99 edition, 2012.