Rhine

Final Report

COMS W4115 - Programming Languages and Translators

Stephen A Edwards

Gudbergur Erlendsson (ge2187) & Ramkumar Ramachandra (rr2893)

August 16, 2014

Contents

1 Introduction

Rhine is a Lisp style language inspired by Clojure. The syntax uses S-Expressions enclosed in parentheses. It has dynamic typing with first class functions and automatic type conversion and it's built on top of LLVM. Although Rhine was originally designed to interact with a text editor, time constraints did not permit the implementation of that feature. Nontheless the language supports a variety of features which will be explained in further detail in following chapters. The name Rhine is recursive acronym for "Rhine is not Emacs" - alluding to the origins of Rhine as a programming language to interact with a text editor.

2 Language Tutorial

This section is intended to teach the basics of Rhine with practical examples.

2.1 Program execution

Rhine programs are JIT compiled and can either be executed by feeding code straight to stdin of executable `rhine`, or interactive programming can be done using the included REPL called `rrhine` which also imports the Rhine standard library automatically.

2.2 REPL

A `REPL` is included called `rrhine`. It makes `rhine` interactive and executes then returns results for each entered S-Expression. Input is cumulative, so it's possible for example to define a function and then use it in following input. Multiline code can be pasted by entering "paste" and then ending the paste mode by entering Ctrl-C.

2.3 Types and variables

Rhine supports types such as integers, doubles, strings and lists. Because Rhine has dynamic typing, their types never need to be specified. Variables are defined with `let` and `def`. `def` should only be used in top level and variables are bound til end of file from where they are defined. `let` binds variables within the `let` S expressions and no more.

Listing 1: Types and variables

```
(println 5)
(println "this is a string")
(def c "This is a definition")
;; c is defined within let
(let [a "another string" b 5.0] (println a) (println b) (println c))
;; c is still defined, but a and b are not.
```

2.4 List and string operations

Rhine supports lists and types of values can be mixed, as well as operations on both lists and strings.

Listing 2: Lists and strings

```

;; Strings built ins include str-split and str-join
;; Prints "Hello world" after splitting and joining the string
(print (str-join (str-split "Hello world")))
```

```

;; Built in list operations include first, rest, cons and length
;; Prints 4
(print (length [1 2 3 4]))
;; Adds 0 to front of list
(cons 0 [1 2 3 4])
;; Gets first element and then [2 3]
(first [1 2 3])
(rest [1 2 3])
```

2.5 Control flow

Rhine has number of control flow structures

Listing 3: Control flow structures

```
;; Prints "Hi!"
(if (= 1 2) (print "Nonono") (print "Hi!"))
;; Prints nothing
(when (<= 3.0 2) (print "Hi"))
;; Prints 5 6 7 8 9
(dotimes [x 5] (print (+ x 5)))
```

2.6 Functions

Functions are defined with `defn` and they are first class citizens that can be passed around. Functions can be recursive.

Listing 4: Functions

```
;; functions are defined with defn
(defn addition [a b] (+ a b))
;; functions can be passed around
;; Prints 6
(defn plusone [a b] (a (+ b 1)))
(plusone print 5)
;; Recursion, prints 10th fibonacci number
(defn fib [n]
   (if (= n 0) 0
     (if (= n 1) 1
       (+ (fib (- n 1)) (fib (- n 2))))))
(println (fib 10))
```

2.7 Writing to stdout

Listing 5: Print

```
;; Use print or println to write to stdout
(print 5)
(println "This adds linebreak after printing")
```

3 Language Reference Manual

This manual intends to describe the Rhine language. It's syntax and semantics will be described in detail in following sections.

3.1 Lexical Conventions

A program is entirely contained within a single file and fed into the `rhine` executable or entered using the included REPL called `rrhine`. Rhine files have the extension ".rh". Rhine does not have a macro or pre-processor system like most Lisp syntax languages.

3.1.1 Tokens

The tokens we have are identifiers, operators, constants, and separators. White space (which consists of newlines, spaces and tabs) is ignored by the tokenizer.

3.1.2 Comments

The character sequence `;;` begins a comment, and it continues until the end of the line. There are no multi-line comments. Comments do not nest, and they do not occur within a string literals.

3.1.3 Identifiers

An identifier (or symbol) is a sequence of so called symbol characters and digits, but cannot begin with a digit. Symbol characters include all letters as well as ?, -, +, *, /, <, >, =, ., % and ˆ. Identifiers are case-sensitive.

3.1.4 Special identifiers

The following identifiers are special and may not be used to create functions or variables: `let`, `def`, `defn`, `dotimes`, `if`, `when`, `defn`.

3.1.5 Operators

The following are builtin operators and are further described in table under section 10: `+ − % ˆ / * > < = >= <= and or not`.

3.1.6 Constants

Integers, floats, string literals, `true`, `false`, `nil` are all constants in Rhine. `true`, `false`, `nil` cannot be used for function or variable names.

3.1.7 Integers

Integer constants are a sequence of digits that can optionally begin with - or + signs which signifies negative and positive integers.

3.1.8 Floats

A floating constant consists of an integer part, a decimal point, a fraction part, an e, and an optionally signed integer exponent. The integer and fraction parts both consist of a sequence of digits. Either the integer part or the fraction part (not both) may be missing; either the decimal point or the e and the exponent (not both) may be missing.

3.1.9 String literals

A string literal is a sequence of characters surrounded by double quotes. Double quotes within a string must be preceded by a backslash to be included in the string.

3.1.10 Others

`true`, `false` and `nil` are regular constants.

3.2 Syntactic structure

The syntax consists of parenthesis and square brackets. The square brackets enclose a “list” that is not to be evaluated, while the parenthesis encloses a S-expression to be used for computation.

An S-expression looks like:

```
(first-argument second-argument third-argument ...)
```

To illustrate a list, the following forms are equivalent:

```
'(first-argument second-argument third-argument ...)
[first-argument second-argument third-argument ...]
```

3.3 Program Structure

A program in Rhine consists of a list of S-expressions, and those that are defined in top level are executed right away. S-expressions when there are many are evaluated from left to right. `def` and `defn` are only to be used in top level. `let` can be used within function definitions and can be nested.

3.4 Type System

Rhine has 6 fundamental types (that are described further above): Integer, Float, Boolean, Symbol, nil, String, and List. The first five are fundamental data types, while the last one is a composite. There is no type checking built into the language.

3.4.1 List type

Lists are essentially S-expressions that are not evaluated. They are internally represented by arrays of pointers to `value_t` items, further described under implementation. They can contain elements of different data types, although care must be taken while mixing types.

3.4.2 Type checking

Functions have specific type signatures although they are not specified in the syntax: the arguments (or their conversions, which is described next) must conform to the types, or there’s a runtime error. Compile-time typechecking is much weaker, and can only catch certain types of errors.

3.4.3 Type conversions

Rhine does most common-sense implicit conversions. Integers get promoted to floats as necessitated by functions operating on them. Integers and floats are converted to Boolean as necessary. Nil is converted to the Boolean false. All types can be printed with print and println.

3.5 Expressions

Expressions are all S-expressions. A Rhine program consists of several S-expressions. Each S-expression can contain several other S-expressions, making a recursive structure. In this structure, the first element is a function, an operator, or a builtin, and the other elements are the arguments. S-expressions are evaluated from left to right when in a list.

3.5.1 Underlying representation

To illustrate how each S-expression is a cons cell:

```
(first-argument second-argument third-argument ...)
(cons first-argument '(second-argument third-argument ...))
(cons first-argument (cons second-argument '(third-argument ...))
```

3.5.2 Expression nesting

To illustrate S-expression nesting, and the first argument being treated as a function:

```
(first-argument0 second-argument0 ...
  (first-argument1 second-argument1 ...
    (...
    ...)))
```

3.6 Scope rules

All bindings in Rhine are lexically scoped. `def` and `defn` definitions extend to the end of the file from where they appear. `let` definitions extend to where the `let` parentheses ends.

3.7 Control Structures

Statement	Description
(if predicate sexpr1 sexpr2)	Evaluates predicate, if true sexpr1 is executed else sexpr2.
(when predicate sexpr)	Evaluates predicate and executes sexpr if predicate is true.
(dotimes [x n] sexpr)	Executes sexpr n-times with n bound to x.

3.8 Function definition and lexical bindings

Statement	Description
(def identifier sexpr)	Creates a variable with the name of the identifier and initializes to the sexp.
(defn identifier [arg*] sexpr*)	Defines a function with the given arguments, and binds it to an identifier.
(let [binding*] sexpr*)	Evaluates the expressions sexp with the bindings list bound in its lexical context.

3.9 Operators

All operators are left associative and since all operators must be enclosed within seperate S-expressions there are no precedence rules.

Statement	Description
'sexpr	Returns the sexpr unevaluated.
(+ sexpr ...)	Adds together all the arguments.
(- sexpr1 sexpr2)	Subtracts the second argument from the first argument.
(* sexpr ...)	Multiplies together all the arguments.
(/ sexpr1 sexpr2)	Divides the first argument with the second argument.
(% sexpr1 sexpr2)	Returns the reminder of (/ sexpr1 sexpr2).
(ˆ sexpr1 sexpr2)	Returns result of sexpr1 in the power of sexpr2.
(> sexpr1 sexpr2)	Evaluates to the boolean sexpr1 > sexpr2.
(< sexpr1 sexpr2)	Evaluates to the boolean sexpr1 < sexpr2.
(= sexpr1 sexpr2)	Tests equality between sexpr1 and sexpr2.
(>= sexpr1 sexpr2)	Evaluates to the boolean sexpr1 >= sexpr2.
(<= sexpr1 sexpr2)	Evaluates to the boolean sexpr1 <= sexpr2.
(and sexpr1 sexpr2)	Evaluates to the logical and between sexpr1 and sexpr2.
(or sexpr1 sexpr2)	Evaluates to the logical or between sexpr1 and sexpr2.
(not sexpr)	Evaluates to the logical not of sexpr.

3.10 List functions

Statement	Description
(first list)	Returns the first element of the list.
(rest list)	Returns everything but the first element of the list.
(cons item list)	Returns the concatenation of [item] and list.
(length list)	Returns the length of the list.

3.11 String functions

Statement	Description
(str-split string)	Splits a string into a list of its characters.
(str-join list)	Takes a list of strings and returns them joined together in one string.
(str-length string)	Takes a string and returns length.

3.12 I/O functions

Statement	Description
(print sexpr)	Evaluates the sexpr and prints to stdout.
(println sexpr)	Evaluates the sexpr, prints to stdout and appends a newline character.

4 Project Processes

4.1 Planning, specifications and development

The project was big and the summer course was short so we met as much as we could, which was usually about 6-10 hours 4-5 times a week. During our meetings we discussed who was working on what and upcoming milestones. We then spent the time developing the language features using Git to pass commits between us, and created unit tests along the way. Our first meeting we decided on doing a Lisp style language on top of LLVM, and when we created the LRM we hashed out most of the specifications for the language.

We worked jointly on scanning and parsing as well as some other elements within the project. Ramkumar was responsible for most LLVM code generation such as arrays and operations on them, control flow structures and function/variable definition sequences while Gudbergur was responsible for the arithmetic and control sequence operators as well as automatic type conversion, the standard library, tests and the test runner and the REPL.

4.2 Development Environment

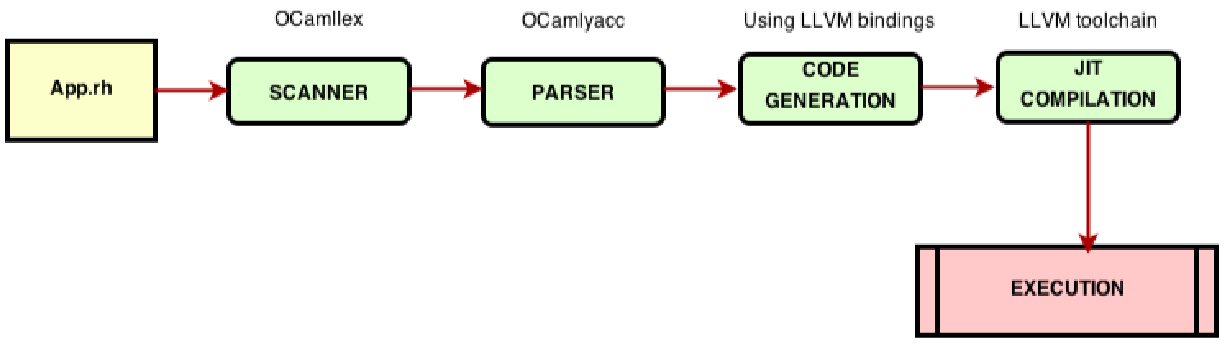
We both used Mac OS X for development, and OCaml, OCamllex, and OCamlyacc, LLVM bindings, LLVM MC Jit compiler and for the compiler itself. We used Git hosted on Github for version control. Test runner and the REPL were written using Python. We used make for compiling.

4.3 Timeline

- July 13th - Decided on Lisp style language with Clojure-like features
- July 14th - Proposal
- July 23rd - LRM due date
- July 30th - Dynamic typing and tests working
- August 10th - Compiler is working and most features except strings are working
- August 14th - All tests pass and almost all features from the original LRM have been implemented
- August 15th - Final report due

5 Implementation and architectural design

5.1 Pipeline



5.1.1 Compilation

A Rhine program can be executed either with `rhine` binary, which executes code directly or the `rrhine`, the included REPL which makes rhine interactive and automatically imports the standard library. The code is run through an Ocamllex program for tokenization, which is then run through Ocamlyacc for parsing. At this point an AST tree has been generated and the code generation then takes over which either generates native LLVM functions or calls to C functions that are defined in bindings.c and called directly from LLVM. LLVM's new JIT compiler, MC JIT then takes over, compiles the bindings and the generated LLVM code, and executes it.

5.2 Implementation details

5.2.1 value_t

`value_t` is the struct responsible for all values. It is not a tagged union, but a tagged struct that has the following fields: type_tag (i32 type), integer (i64 type), bool (i1 type), string (pointer), array (pointer), array/string length (i64 type), double (double type), function (pointer). All values defined within Rhine are saved in one of these fields and type_tag (and sometimes array/string length) is set appropriately. We defined functions that handle the boxing and unboxing of these values into/from the `value_t` struct.

5.2.2 Type conversion

Integers are automatically upgraded to floats when arithmetic operations get at least one double as a parameter. The type conversions are done in the C bindings.

5.2.3 LLVM code generation

For every top level S-expressions we define an anonymous function in LLVM IR that is called immediately.

LLVM codegen statements all have side-effects. In most places, order in which variables are codegen'ed are unimportant, and lets can be moved around; but in conditionals, statements must be codegen'ed exactly in order: this means let statements can't be permuted, leading to ugly imperative-style OCaml code.
Âă

Since LLVM IR is strongly typed, it is possible to inspect the types of llvalues from OCaml at generation-time. However, there is no way for the codegen-stage to inspect the values of variables while the program is running. This has the unfortunate consequence that a loop hinged upon an llvalue must be implemented in LLVM IR; hence, for functions that require iterating on variable-length arrays, we end up writing tedious LLVM IR generation code instead of an equivalent OCaml code.

Debugging errors from LLVM bindings is hard. Using ocamldebug does not work, since the callstack leading up to an LLVM call in C++ is not owned by OCaml. The alternative is to use lldb, but matching up the C++ call with a line in the OCaml code is non-trivial.

5.2.4 Example generated code

Running this code in rhine:

```
(println (rest [1 2 3]))
```

Generates this LLVM IR code:

```
define i64 @0() {
entry:
  %ar = alloca [3 x %value_t*]
  %value = call i8* @malloc(i64 ptrtoint (%value_t* getelementptr (%value_t* null, i32 1) t
  %malloc_value = bitcast i8* %value to %value_t*
  %boxptr = getelementptr inbounds %value_t* %malloc_value, i32 0, i32 0
  %boxptr1 = getelementptr inbounds %value_t* %malloc_value, i32 0, i32 1
  store i32 1, i32* %boxptr
  store i64 1, i64* %boxptr1
  %value2 = call i8* @malloc(i64 ptrtoint (%value_t* getelementptr (%value_t* null, i32 1)
  %malloc_value3 = bitcast i8* %value2 to %value_t*
  %boxptr4 = getelementptr inbounds %value_t* %malloc_value3, i32 0, i32 0
  %boxptr5 = getelementptr inbounds %value_t* %malloc_value3, i32 0, i32 1
  store i32 1, i32* %boxptr4
  store i64 2, i64* %boxptr5
  %value6 = call i8* @malloc(i64 ptrtoint (%value_t* getelementptr (%value_t* null, i32 1)
  %malloc_value7 = bitcast i8* %value6 to %value_t*
  %boxptr8 = getelementptr inbounds %value_t* %malloc_value7, i32 0, i32 0
  %boxptr9 = getelementptr inbounds %value_t* %malloc_value7, i32 0, i32 1
  store i32 1, i32* %boxptr8
  store i64 3, i64* %boxptr9
  %arptr = getelementptr inbounds [3 x %value_t*]* %ar, i32 0, i32 0
  store %value_t* %malloc_value, %value_t** %arptr
  %arptr10 = getelementptr inbounds [3 x %value_t*]* %ar, i32 0, i32 1
  store %value_t* %malloc_value3, %value_t** %arptr10
  %arptr11 = getelementptr inbounds [3 x %value_t*]* %ar, i32 0, i32 2
  store %value_t* %malloc_value7, %value_t** %arptr11
  %value12 = call i8* @malloc(i64 ptrtoint (%value_t* getelementptr (%value_t* null, i32 1)
  %malloc_value13 = bitcast i8* %value12 to %value_t*
  %lenptr = getelementptr inbounds %value_t* %malloc_value13, i32 0, i32 5
  store i64 3, i64* %lenptr
  %boxptr14 = getelementptr inbounds %value_t* %malloc_value13, i32 0, i32 0
  %boxptr15 = getelementptr inbounds %value_t* %malloc_value13, i32 0, i32 4
  store i32 4, i32* %boxptr14
  store %value_t** %arptr, %value_t*** %boxptr15
  %load = load i64* %lenptr
  %rest = getelementptr inbounds [3 x %value_t*]* %ar, i32 0, i64 1
  %restsub = add i64 %load, -1
  %value18 = call i8* @malloc(i64 ptrtoint (%value_t* getelementptr (%value_t* null, i32 1)
  %malloc_value19 = bitcast i8* %value18 to %value_t*
  %boxptr20 = getelementptr inbounds %value_t* %malloc_value19, i32 0, i32 0
  %lenptr21 = getelementptr inbounds %value_t* %malloc_value19, i32 0, i32 5
  %boxptr22 = getelementptr inbounds %value_t* %malloc_value19, i32 0, i32 4
  store i32 4, i32* %boxptr20
  store i64 %restsub, i64* %lenptr21
  store %value_t** %rest, %value_t*** %boxptr22
  %call = call %value_t* @println(%value_t* %malloc_value19)
  %boxptr23 = getelementptr inbounds %value_t* %call, i32 0, i32 1
  %load24 = load i64* %boxptr23
  ret i64 %load24
}
```

6 Testing

6.1 Test runner

The testrunner was written in Python. It accepts and looks in the tests/ directory for "Rhine Test files" that are files that contain both a Rhine program, and expected output from that program, seperated by three hyphens or "—". We created individual tests for every feature of the language.

6.2 Tests

Listing 6: Our tests

```
;; Types
(println 5)
(println 5999999)
(println 2.7)
(println 2e-5)
(println .2)
(println 2.)
(println "hello")
(println nil)
(println true)
(println false)
(println [])
(println [1 2 3])
(println ["a" "b" "c"])

;; Arithmetic operators
(println (and true false))
(println (and true true))
(println (or true false))
(println (or false false))
(println (not false))
(println (+ 2 (- 1 5)))
(println (+ 2 5 10 60))
(println (* 2 5 10 60))
(println (+ (+ 2 (- 1 5)) 10))
(println (- (+ 2 (- 1 5)) 10))
(println (* 2 (/ 1 5)))
(println (* 2 4))
(println (* 6 5))
(println (* .2 (/ 2.0 5)))
(println (+ .2 (- 2.0 5)))
(println (+ 1e2 (- 2.0 5)))
(println (% 5 4))
(println (% 23 4))
(println (% 5 4))
(println (% 5 4))
(println (^ 2 4))

;; Conditional operators
(println (= 4 4))
(println (= 1 1))
(println (= 0 0))
```

```
(println (= 2 4))
(println (= [] []))
(println (= [1] [1]))
(println (< 2 4))
(println (> 1 6))
(println (> 3 4))
(println (<= 3 4))
(println (<= 4 5))
(println (<= 5.0 4.0))
(println (<= 4 4))
(println (>= 5 4))
(println (>= 5.0 4.0))
(println (>= 4 4))
(println (>= 1 3))
(println (< 1.0 4))
(println (< 1 4))
(println (> 1.0 4.0))
(println (> (* 2 2.0) (* 5 5)))
(println (< (* 2 4.0) (* 9 5)))
(println (= true true))
(println (= false true))
(println (= false false))
(println (= true false))



;; Lists
(println (cons 1 [2 3 4]))
(println (cons 1 [2.5 3.0 4]))
(println (cons "str" ["s" 3.0 "x"]))
(if (= [] []) (print "OK!") (print "NO"))
(if (= [1] [1]) (print "OK!") (print "NO"))
(print (nth 1 [1.0 "string" [1 2 3] 4 5 6]))
(print (nth 2 [1.0 "string" [1 2 3] 4 5 6]))
(print (nth 3 [1.0 "string" [1 2 3] 4 5 6]))
(print (nth 4 [1.0 "string" [1 2 3] 4 5 6]))
(print (length [ 1 2 3 4]))
(println [1 2 3])
(println (first [66 2 4 5 6]))
(println (first ["asd" "ad" "ll"]))
(println (first [2.0 4.0]))
(println (rest [2.0 4.0]))
(println (rest ["asd" "ad" "ll"]))
(println (rest [1 3 5 6]))


;; Print
(print "hello")
(print 83)
(print 2.0)
(println 3.0)
(println 2)
(println [1 2 3])
```

```
(print [1 2.0])
(println true)


;; Strings
(println "this is a string")
(println "this
is
a
string")
(println "this is a \"string\" with a \"")
(print "the end")
(println (str-join ["f" "o" "o"]))
(str-length "foom")
(print (str-join (str-split "hello")))
(println (str-split "foo"))


;; Def, Defn, Let and Dotimes
(def test 5.0)
(print test)
(defn test1 [b] b)
(defn test2 [a b] (+ a b))
(defn test3 [a b c] (if true (* a b c) []))
(print (test1 55))
(print (test2 55 66))
(print (test3 55 66 22))
(dotimes [x 10] (print "hello world!"))
(dotimes [x 4] (print (* x (+ x 5))))
(defn foo [f] (println 5))
(defn bar [f] (f 5))
(foo println)
(bar println)
(let [a 1 b 2 c 3] (println (+ a b c)))
(let [a "string" b 2.30 c 3.0] (println (+ b c)))
(let [a 1 b a] (println b))
(let [a 1] (let [b a c b] (println c)))
(println (let [b 1 c 2] (+ b 2) (+ c 2)))


;; Conditionals
(println (= false true))
(println (= true true))
(println (if (= false true) 88 99))
(println (if (= 0 1) 88 99))
(println (if (= 1 1) 88 99))
(println (if (= true true) 88 99))
(println (if false "hello" "hmm"))
(if (not false) (println "hey!") (println "no"))
(when (= false true) (println 43))
(when (= true true) (println 42))
```

7 Lessons Learned

7.1 Gudbergur

I learned a lot this summer, both in class and doing the project. Implementing the machinery needed for modern programming languages to work is a daunting task and although Rhine only scratched the surface of it, it was an eye opening experience. From scanning, parsing, code generation to test suites, LLVM intricacies etc there was a lot to do and learn. OCaml, the LLVM IR as the Lisp style were all pretty new to me in the beginning so it was hard at first to get into the mindset of working with all these different technologies.

OCaml in particular was very nice to work with. Algebraic data types with pattern matching made building AST using the lexer and parser incredibly smooth. Although we couldn't explore all the options, by using the LLVM infrastructure a lot of features can come cheaply by generating LLVM IR, such as garbage collection and different targest like Javascript using Emscripten even though the JIT compiler does not seem to get a lot of attention. Having a regression test suite was a great boon for the development progress and I recommend future teams to incorporate them early.

7.2 Ramkumar

At the time of the proposal, we hadn't fully realized the repurcussions of a fully dynamically-typed language. We initially thought we could get away with some sort of type inference, but the real challenge of the project was supporting arrays containing multiple types: it's impossible to associate a type to such an array, and hence reason about it.

We started out with zero experience in LLVM, and built up a high degree of expertise by the end of the term. At first, the task of using LLVM OCaml bindings seemed unsurmountable due to absolute lack of documentation or support. Over the weeks, we learnt mostly via type signatures; a typesystem is a powerful tool in reasoning about programs.

We didn't realize the problem with functions taking multiple value_t* arguments and returning a value_t* until we needed to implement first class functions (using function pointers).

Implementing complex functions as builtins (by directly generating IR) is perhaps more efficient than implementing them in Rhine, but the gap is very small due to optimization passes applied on the Rhine-generated IR. The marginal benefit outweighs the cost of correctly weaving complex IR; perhaps we should have chosen simpler string primitives than str-split and str-join.

if and dotimes were incredibly hard to get right. This is because we had to generate basic blocks and build a phi node to unify them correctly. We chose to do only one loop, but perhaps the language needs more looping constructs; recursion doesn't always cut it because of the inficieny of creating a stack frame for each iteration.Âă

8 Standard library

Listing 7: stdlib.rh

```
;; Absolute value of number
(defn abs [n] (if (< n 0) (* n -1) n))

;; Concats two strings
(defn concat [a b]
    (if (not (= [] a))
      (cons (first a) (concat (rest a) b))
      b))

;; Factorial
(defn factorial [n]
    (if (< n 2)
       n
       (* n (factorial (- n 1)))))

;; Decrement number by 1
(defn dec [n] (- n 1))

;; Increment number by 1
(defn inc [n] (+ n 1))

;; Return first n items of coll
(defn take [n coll]
    (if
       (and (> n 0) (not (= [] coll)))
       (cons (first coll) (take (- n 1) (rest coll)))
       []))

;; Drop the first n items of coll
(defn drop [n coll]
   (if (and (> n 0) (not (= [] coll)))
       (drop (dec n) (rest coll))
       coll))

;; Return the nth element of coll
(defn nth [n coll] (first (drop (dec n) coll)))


;; Returns the result of applying f to each element of coll
(defn map
  [f coll]
  (if (not (= [] coll))
    (cons (f (first coll))
          (map f (rest coll)))
    []))
```

9 Appendix

9.1 Makefile

```

OBJS = parser.cmo lexer.cmo pretty.cmo codegen.cmo toplevel.cmo main.cmo bindings.o
ocamlc = ocamlc -g -w @5@8@10@11@12@14@23@24@26@29@40

rhine : $(OBJS)
	ocamlfind $(ocamlc) -package llvm -package llvm.executionengine -package llvm.analysis -package
lexer.ml : lexer.mll
	ocamllex lexer.mll
parser.ml parser.mli : parser.mly
	ocamlyacc parser.mly
codegen.cmo: codegen.ml
	ocamlfind $(ocamlc) -c -package llvm -linkpkg $<
toplevel.cmo: toplevel.ml
	ocamlfind $(ocamlc) -c -package llvm -linkpkg $<
%.cmo : %.ml
	$(ocamlc) -c $<
%.cmi : %.mli
	$(ocamlc) -c $<
%.o   : %.c
	cc -c -o $@ $<
clean :
	rm -f rhine parser.ml parser.mli lexer.ml *.o *.cmo *.cmi *.cmx

codegen.cmo : ast.cmi
codegen.cmx : ast.cmi
lexer.cmo : parser.cmi
lexer.cmx : parser.cmx
main.cmo : toplevel.cmo pretty.cmo parser.cmi lexer.cmo ast.cmi
main.cmx : toplevel.cmx pretty.cmx parser.cmx lexer.cmx ast.cmi
parser.cmo : ast.cmi parser.cmi
parser.cmx : ast.cmi parser.cmi
pretty.cmo : ast.cmi
pretty.cmx : ast.cmi
toplevel.cmo : codegen.cmo ast.cmi
toplevel.cmx : codegen.cmx ast.cmx
ast.cmi :
parser.cmi : ast.cmi
```

9.2 ast.mli

```
type atom =
    Symbol of string
  | Int of int
  | Bool of bool
  | Double of float
  | String of string
  | Nil
```

```
type sexpr =
    Atom of atom
  | DottedPair of sexpr * sexpr
  | Vector of sexpr array

type prog =
    Prog of sexpr list

type proto =
    Prototype of string * string array

type func =
    Function of proto * sexpr
```

9.3 lexer.mll

```
{
  open Parser
  exception SyntaxError of string
}

let digit = ['0'-'9']
let characters = ['a'-'z' 'A'-'Z']
let symbol_chars = ['a'-'z' 'A'-'Z' '?' '-' '+' '*' '/' '<' '>' '=' '.' '%' '^']
let e = ['E''e']['-''+']?['0'-'9']+

rule token = parse
 | [' ' '\t' '\r' '\n'] { token lexbuf }
 | ";;" { comment lexbuf }
 | '(' { LPAREN }
 | ')' { RPAREN }
 | '[' { LSQBR }
 | ']' { RSQBR }
 | "nil" { NIL }
 | "true" { TRUE }
 | "false" { FALSE }
 | (['0'-'9']*)'.'['0'-'9']+e? as s { DOUBLE(float_of_string s) }
 | (['0'-'9']+)'.'['0'-'9']*e? as s { DOUBLE(float_of_string s) }
 | (['0'-'9']+)e as s { DOUBLE(float_of_string s) }
 | ['-''+']?digit+ as s { INTEGER(int_of_string s) }
 | (symbol_chars (digit|symbol_chars)*) as s { SYMBOL(s) }
 | '"' { let b = Buffer.create 1024 in read_string b lexbuf }
 | eof { EOF }

and comment = parse
 '\n' { token lexbuf }
 | _ { comment lexbuf }

and read_string b = parse
 "\\\"" { Buffer.add_string b "\""; read_string b lexbuf }
 | '"' { STRING(Buffer.contents b) }
 | [^'"'] { Buffer.add_string b (Lexing.lexeme lexbuf); read_string b lexbuf }
```

9.4 parser.mly

```
%{ open Ast %}

%token LPAREN RPAREN LSQBR RSQBR NIL TRUE FALSE EOF
%token <int> INTEGER
%token <float> DOUBLE
%token <string> SYMBOL
%token <string> STRING

%start prog
%type <Ast.prog> prog
%type <Ast.atom> atom
%type <Ast.sexpr> sexpr

%%

prog:
    sexprs EOF { Prog($1) }

atom:
   NIL { Nil }
 | TRUE { Bool(true) }
 | FALSE { Bool(false) }
 | INTEGER { Int($1) }
 | DOUBLE { Double($1) }
 | STRING { String($1) }
 | SYMBOL { Symbol($1) }

sexpr:
   atom { Atom($1) }
 | LPAREN sexprs RPAREN {
       let rec buildDP = function
            [] -> Atom(Nil)
          | h::t -> DottedPair(h, buildDP t)
       in buildDP($2)
    }

qsexpr:
    LSQBR sexprs RSQBR { Vector(Array.of_list $2) }
  | LSQBR RSQBR { Vector([||]) }

sexprs:
   sexpr sexprs { $1::$2 }
 | qsexpr sexprs { $1::$2 }
 | sexpr { [$1] }
 | qsexpr { [$1] }
```

9.5 main.ml

```
open Ast
type action = Pprint | Normal
```

```

let main () =
  let action =
    if Array.length Sys.argv > 1 then
      try
        List.assoc Sys.argv.(1) [ ("-p", Pprint) ]
      with Not_found -> Normal
    else
      Normal
  in
  let lexbuf = Lexing.from_channel stdin in
  let prog = Parser.prog Lexer.token lexbuf in
  match action with
    Normal -> (match prog with
                 Prog(ss) -> Toplevel.main_loop ss)
  | Pprint -> Pretty.pprint prog
;;

main ()
```

9.6 toplevel.ml

```
open Llvm
open Llvm_executionengine
open Llvm_target
open Llvm_scalar_opts
open Codegen

exception Error of string

let _ = initialize_native_target ()
let the_execution_engine = ExecutionEngine.create_jit the_module 1
let the_fpm = PassManager.create_function the_module

let emit_anonymous_f s =
  codegen_func(Ast.Function(Ast.Prototype("", [||]), s))

let extract_strings args = Array.map (fun i ->
                                       (match i with
                                          Ast.Atom(Ast.Symbol(s)) -> s
                                        | _ -> raise (Error "Bad argument")))
                                     args

let parse_defn_form sexpr = match sexpr with
    Ast.DottedPair(Ast.Atom(Ast.Symbol(sym)),
                   Ast.DottedPair(Ast.Vector(v), body)) ->
     (sym, extract_strings v, body)
  | _ -> raise (Error "Unparseable defn form")

let parse_def_form sexpr = match sexpr with
    Ast.DottedPair(Ast.Atom(Ast.Symbol(sym)),
                   Ast.DottedPair(expr, Ast.Atom(Ast.Nil))) -> (sym, expr)
  | _ -> raise (Error "Unparseable def form")

let sexpr_matcher sexpr =
  let value_t = match type_by_name the_module "value_t" with
      Some t -> t
    | None -> raise (Error "Could not look up value_t") in
  match sexpr with
    Ast.DottedPair(Ast.Atom(Ast.Symbol("defn")), s2) ->
    let (sym, args, body) = parse_defn_form s2 in
    codegen_func(Ast.Function(Ast.Prototype(sym, args), body))
  | Ast.DottedPair(Ast.Atom(Ast.Symbol("def")), s2) ->
      (* Emit initializer function *)
      let the_function = codegen_proto (Ast.Prototype("", [||])) in
      let bb = append_block context "entry" the_function in
      position_at_end bb builder;
      let (sym, expr) = parse_def_form s2 in
      let llexpr = codegen_sexpr expr in
      let llexpr = build_load llexpr "llexpr" builder in
      let global = define_global sym (const_null value_t) the_module in
      ignore (build_store llexpr global builder);
      ignore (build_ret (const_int i64_type 0) builder);
      the_function
  | _ -> emit_anonymous_f sexpr

let print_and_jit se =
  let f = sexpr_matcher se in

  (* Validate the generated code, checking for consistency. *)
  (* Llvm_analysis.assert_valid_function f;*)

  (* Optimize the function. *)
  ignore (PassManager.run_function f the_fpm);

  dump_value f;

  if Array.length (params f) == 0 then (
    let result = ExecutionEngine.run_function f [||] the_execution_engine in
    print_string "Evaluated to ";
    print_int (GenericValue.as_int result);
    print_newline ()
  )

let main_loop ss =
  (* Do simple "peephole" optimizations and bit-twiddling optzn. *)

  add_instruction_combination the_fpm;

  (* reassociate expressions. *)
  add_reassociation the_fpm;

  (* Eliminate Common SubExpressions. *)
  add_gvn the_fpm;
  (* Simplify the control flow graph (deleting unreachable blocks, etc). *)
  add_cfg_simplification the_fpm;

  ignore (PassManager.initialize the_fpm);

  (* Declare global variables/ types *)
  let llvalue_t = named_struct_type context "value_t" in
  let value_t_elts = [| i32_type;                 (* value type of struct, integer: 1, bool: 2, stri
                        i64_type;                 (* integer *)
                        i1_type;                  (* bool *)
                        (pointer_type i8_type);   (* string *)
                        (pointer_type (pointer_type llvalue_t));  (* array *)
                        i64_type; (* array length *)
                        double_type; (* double *)
                        pointer_type (function_type
                                        (pointer_type llvalue_t)
                                        [| (pointer_type llvalue_t) |]);
                     |] in
  struct_set_body llvalue_t value_t_elts false;

  (* Declare external functions *)
  let ft = function_type (pointer_type i8_type) [| i64_type |] in
  ignore (declare_function "malloc" ft the_module);
  let ft = function_type i64_type [| pointer_type i8_type |] in
  ignore (declare_function "strlen" ft the_module);
  let ft = function_type i32_type
                        [| pointer_type i8_type; pointer_type i8_type;
                           i64_type; i32_type; i1_type |] in
  ignore (declare_function "llvm.memcpy.p0i8.p0i8.i64" ft the_module);
  ignore (codegen_proto (Ast.Prototype("println", Array.make 1 "v")));
  ignore (codegen_proto (Ast.Prototype("print", Array.make 1 "v")));
  ignore (codegen_proto (Ast.Prototype("cadd", Array.make 2 "v")));
  ignore (codegen_proto (Ast.Prototype("cdiv", Array.make 2 "v")));
  ignore (codegen_proto (Ast.Prototype("csub", Array.make 2 "v")));
  ignore (codegen_proto (Ast.Prototype("cmul", Array.make 2 "v")));
  ignore (codegen_proto (Ast.Prototype("cmod", Array.make 2 "v")));
  ignore (codegen_proto (Ast.Prototype("cexponent", Array.make 2 "v")));
  ignore (codegen_proto (Ast.Prototype("clt", Array.make 2 "v")));
  ignore (codegen_proto (Ast.Prototype("cgt", Array.make 2 "v")));
  ignore (codegen_proto (Ast.Prototype("clte", Array.make 2 "v")));
  ignore (codegen_proto (Ast.Prototype("cgte", Array.make 2 "v")));
  ignore (codegen_proto (Ast.Prototype("cequ", Array.make 2 "v")));
  ignore (codegen_proto (Ast.Prototype("cand", Array.make 2 "v")));
  ignore (codegen_proto (Ast.Prototype("cor", Array.make 2 "v")));
  ignore (codegen_proto (Ast.Prototype("cnot", Array.make 1 "v")));
  ignore (codegen_proto (Ast.Prototype("cstrjoin", Array.make 1 "v")));

  List.iter (fun se -> print_and_jit se) ss
```

9.7 pretty.ml

```
open Ast
```

```

let print_bool = function true -> print_string "true"
                        | false -> print_string "false"

(* pretty print atoms *)
let ppatom p = match p with
    Symbol(s) -> print_string ("sym:" ^ s)
  | String(s) -> print_string ("str:" ^ s)
  | Int(i) -> print_string "int:"; print_int i
  | Bool(i) -> print_string "bool:"; print_bool i
  | Double(d) -> print_string "dbl:"; print_float d
  | Nil -> print_string "nil"

(* pretty print S-expressions *)
let rec ppsexpr p islist = match p with
    Atom(a) -> ppatom a
  | DottedPair(se1,se2) -> if islist then () else
                             print_string "( ";
                            (match se1 with
                               Atom(a) -> ppatom a
                              |_ -> ppsexpr se1 false);
                            print_char ' ';
                            (match se2 with
                               Atom(Nil) -> print_char ')'
                              |Atom(_ as a) -> print_string ". ";
                                               ppatom a;
                                               print_string " )"
                              |_ -> ppsexpr se2 true)
  | Vector(qs) -> print_string "[ ";
                  Array.iter (fun i -> ppsexpr i false;
                                      print_char ' ') qs;
                  print_char ']'
(* pretty print the program *)
let pprint p = match p with
    Prog(ss) -> List.iter (fun i -> ppsexpr i false;
                                    print_newline ();
                                    print_newline ()) ss
```

9.8 codegen.ml

```
open Llvm

module StringSet = Set.Make(String)

exception Error of string

let context = global_context ()
let the_module = create_module context "Rhine JIT"
let builder = builder context
let named_values:(string, llvalue) Hashtbl.t = Hashtbl.create 10
let i8_type = i8_type context
let i32_type = i32_type context
let i64_type = i64_type context
let i1_type = i1_type context
let double_type = double_type context
let void_type = void_type context

let int_of_bool = function true -> 1 | false -> 0

let (--) i j =
    let rec aux n acc =
      if n < i then acc else aux (n-1) (n :: acc)
    in aux j []

let arith_ops = List.fold_left (fun s k -> StringSet.add k s)
                               StringSet.empty
                               [ "+"; "-"; "*"; "/"; "/."; "<"; ">"; "%"; "^";
                                 "<="; ">="; "="; "and"; "or"; "not" ]

let array_ops = List.fold_left (fun s k -> StringSet.add k s)
                               StringSet.empty
                               [ "first"; "rest"; "cons"; "length" ]

let string_ops = List.fold_left (fun s k -> StringSet.add k s)
                                StringSet.empty
                                [ "str-split"; "str-join"; "str-length" ]

let cf_ops = List.fold_left (fun s k -> StringSet.add k s)
                            StringSet.empty
                            [ "if"; "when"; "dotimes" ]

let binding_ops = List.fold_left (fun s k -> StringSet.add k s)
                                 StringSet.empty
                                 [ "let"; "def" ]


let create_entry_block_alloca the_function var_name =
  let value_t = match type_by_name the_module "value_t" with
      Some t -> t
    | None -> raise (Error "Could not look up value_t")
  in
  let builder = builder_at context (instr_begin (entry_block the_function)) in
  build_alloca value_t var_name builder

let build_malloc llsize llt id builder =
  let callee = match lookup_function "malloc" the_module with
      Some callee -> callee
    | None -> raise (Error "Unknown function referenced") in
  let raw_ptr = build_call callee [| llsize |] id builder in
  build_bitcast raw_ptr (pointer_type llt) "malloc_value" builder

let build_strlen llv =
  let callee = match lookup_function "strlen" the_module with
      Some callee -> callee
    | None -> raise (Error "Unknown function referenced") in
  build_call callee [| llv |] "strlen" builder
```

```
let build_memcpy src dst llsize =
  let callee = match lookup_function "llvm.memcpy.p0i8.p0i8.i64" the_module with
      Some callee -> callee
    | None -> raise (Error "Unknown function referenced") in
  build_call callee [| dst; src; llsize;
                       const_int i32_type 0;
                       const_int i1_type 0 |] "memcpy" builder


let idx n = [| const_int i32_type 0; const_int i32_type n |]

let undef_vec len =
  let undef_list = List.map (fun i -> undef i64_type) (0--(len - 1)) in
  const_vector (Array.of_list undef_list)

let box_llar llval lllen =
  let value_t = match type_by_name the_module "value_t" with
      Some t -> t
    | None -> raise (Error "Could not look up value_t")
  in
  let value_ptr = build_malloc (size_of value_t) value_t "value" builder in
  let type_dst = build_in_bounds_gep value_ptr (idx 0) "boxptr" builder in
  let lenptr = build_in_bounds_gep value_ptr (idx 5) "lenptr" builder in
  let dst = build_in_bounds_gep value_ptr (idx 4) "boxptr" builder in
  let lltype_tag = const_int i32_type 4 in
  ignore (build_store lltype_tag type_dst builder);
  ignore (build_store lllen lenptr builder);
  ignore (build_store llval dst builder);
  value_ptr

let box_value llval =
  let value_t = match type_by_name the_module "value_t" with
      Some t -> t
    | None -> raise (Error "Could not look up value_t")
  in
  let rharray_type size = array_type (pointer_type value_t) size in
  let value_ptr = build_malloc (size_of value_t) value_t "value" builder in
  let match_pointer ty = match ty with
    | ty when ty = pointer_type (function_type (pointer_type value_t)
                                               [| (pointer_type value_t) |]) ->
      (7, llval)
    | _ ->
      match element_type ty with
        ty when ty = i8_type ->
        let len = build_strlen llval in
        let lenptr = build_in_bounds_gep value_ptr (idx 5) "lenptr" builder in
        ignore (build_store len lenptr builder);
        (3, llval)
      | ty when ty = rharray_type (array_length ty) ->
         let len = const_int i64_type (array_length ty) in
         let lenptr = build_in_bounds_gep value_ptr (idx 5) "lenptr" builder in
         ignore (build_store len lenptr builder);
```

```
          (4, build_in_bounds_gep llval (idx 0) "llval" builder)
       | ty -> raise (Error "Don't know how to box type") in
  let match_composite ty = match classify_type ty with
      TypeKind.Pointer -> match_pointer ty
    | _ -> raise (Error "Don't know how to box type") in
  let (type_tag, llval) = match type_of llval with
      ty when ty = i64_type ->
       (1, llval)
    | ty when ty = i1_type ->
       (2, llval)
    | ty when ty = double_type ->
       (6, llval)
    | ty -> match_composite ty
  in
  let type_dst = build_in_bounds_gep value_ptr (idx 0) "boxptr" builder in
  let dst = build_in_bounds_gep value_ptr (idx type_tag) "boxptr" builder in
  let lltype_tag = const_int i32_type type_tag in
  ignore (build_store lltype_tag type_dst builder);
  ignore (build_store llval dst builder);
  value_ptr

let get_type llval =
  let type_num = build_in_bounds_gep llval (idx 0) "boxptr" builder in
  build_load type_num "load" builder

let unbox_dbl llval =
  let dst = build_in_bounds_gep llval (idx 6) "boxptr" builder in
  build_load dst "load" builder

let unbox_int llval =
  let dst = build_in_bounds_gep llval (idx 1) "boxptr" builder in
  build_load dst "load" builder

let unbox_bool llval =
  let dst = build_in_bounds_gep llval (idx 2) "boxptr" builder in
  build_load dst "load" builder

let unbox_function llval =
  let dst = build_in_bounds_gep llval (idx 7) "boxptr" builder in
  build_load dst "load" builder

let unbox_str llval =
  let dst = build_in_bounds_gep llval (idx 3) "boxptr" builder in
  build_load dst "load" builder

let unbox_length llval =
  let dst = build_in_bounds_gep llval (idx 5) "boxptr" builder in
  build_load dst "load" builder

let unbox_ar llval =
  let value_t = match type_by_name the_module "value_t" with
      Some t -> t
    | None -> raise (Error "Could not look up value_t")
  in
  let ptr = build_in_bounds_gep llval (idx 4) "boxptr" builder in
  let el = build_load ptr "el" builder in
  let rharray_type size = pointer_type (array_type
                                          (pointer_type value_t) size) in
  let vec n = build_bitcast el (rharray_type n) "arptr" builder in
  let arload n = build_load (vec n) "load" builder in
  arload 10

let codegen_atom atom =
  let value_t = match type_by_name the_module "value_t" with
      Some t -> t
    | None -> raise (Error "Could not look up value_t")
  in
  let unboxed_value = match atom with
      Ast.Int n -> const_int i64_type n
    | Ast.Bool n -> const_int i1_type (int_of_bool n)
    | Ast.Double n -> const_float double_type n
    | Ast.Nil -> const_null (pointer_type value_t)
    | Ast.String s -> build_global_stringptr s "string" builder
    | Ast.Symbol n -> match lookup_global n the_module with
                        Some v -> v
                      | None ->
                         match lookup_function n the_module with
                           Some f -> box_value f
                         | None ->
                            try Hashtbl.find named_values n with
                              Not_found -> raise (Error "Symbol not bound")
  in match atom with
       Ast.Symbol n -> unboxed_value
     | Ast.Nil -> unboxed_value
     | _ -> box_value unboxed_value

let rec extract_args s = match s with
    Ast.DottedPair(s1, s2) ->
    begin match (s1, s2) with
            (Ast.Atom m, Ast.DottedPair(_, _)) ->
            (codegen_atom m)::(extract_args s2)
          | (Ast.Atom m, Ast.Atom(Ast.Nil)) -> [codegen_atom m]
          | (Ast.DottedPair(_, _), Ast.DottedPair(_, _)) ->
             (codegen_sexpr s1)::(extract_args s2)
          | (Ast.DottedPair(_, _), Ast.Atom(Ast.Nil)) -> [codegen_sexpr s1]
          | (Ast.Vector(qs), Ast.DottedPair(_, _)) ->
             (codegen_array qs)::(extract_args s2)
          | (Ast.Vector(qs), Ast.Atom(Ast.Nil)) -> [codegen_array qs]
          | _ -> raise (Error "Malformed sexp")
    end
  | _ -> raise (Error "Expected sexp")

and codegen_one_arg s = match s with
    Ast.DottedPair(s1, s2) ->
    begin match (s1, s2) with
            (Ast.Atom m, Ast.DottedPair(_, _)) ->
            codegen_atom m
          | (Ast.Atom m, Ast.Atom(Ast.Nil)) -> codegen_atom m
          | (Ast.DottedPair(_, _), Ast.DottedPair(_, _)) ->
             codegen_sexpr s1
          | (Ast.DottedPair(_, _), Ast.Atom(Ast.Nil)) -> codegen_sexpr s1
          | (Ast.Vector(qs), Ast.DottedPair(_, _)) ->
             codegen_array qs
          | (Ast.Vector(qs), Ast.Atom(Ast.Nil)) -> codegen_array qs
          | _ -> raise (Error "Malformed sexp")
    end
  | _ -> raise (Error "Expected sexp")

and codegen_arith_op op args =
    let hd = List.hd args in
    let tl = List.tl args in
    match op with
      "not" -> codegen_call_op "cnot" [hd]
    | _ ->
        if tl == [] then hd else
          let snd = List.nth args 1 in
          match op with
            "+" -> codegen_call_op "cadd" [hd;(codegen_arith_op op tl)]
          | "-" -> codegen_call_op "csub" [hd;snd]
          | "/" -> codegen_call_op "cdiv" [hd;snd]
          | "*" -> codegen_call_op "cmul" [hd;(codegen_arith_op op tl)]
          | "%" -> codegen_call_op "cmod" [hd;snd]
          | "^" -> codegen_call_op "cexponent" [hd;snd]
          | "<" -> codegen_call_op "clt" [hd;snd]
          | ">" -> codegen_call_op "cgt" [hd;snd]
          | "<=" -> codegen_call_op "clte" [hd;snd]
          | ">=" -> codegen_call_op "cgte" [hd;snd]
          | "=" -> codegen_call_op "cequ" [hd;snd]
          | "and" -> codegen_call_op "cand" [hd;snd]
          | "or" -> codegen_call_op "cor" [hd;snd]
          | _ -> raise (Error "Unknown arithmetic operator")

and codegen_array_op op args =
  let value_t = match type_by_name the_module "value_t" with
      Some t -> t
    | None -> raise (Error "Could not look up value_t") in
  let arg = List.hd args in
  match op with
    "first" ->
    let ar = unbox_ar arg in
    build_extractvalue ar 0 "extract" builder
  | "rest" ->
     let ptr = build_in_bounds_gep arg (idx 4) "boxptr" builder in
     let lenptr = build_in_bounds_gep arg (idx 5) "boxptr" builder in
     let len = build_load lenptr "load" builder in
     let el = build_load ptr "el" builder in
     let newptr = build_in_bounds_gep el [| const_int i64_type 1 |]
                                     "rest" builder in
     let newlen = build_sub len (const_int i64_type 1) "restsub" builder in
```

```
      box_llar newptr newlen
  | "length" ->
      box_value (unbox_length arg)
  | "cons" ->
      let tail = List.nth args 1 in
      let lenptr = build_in_bounds_gep tail (idx 5) "boxptr" builder in
      let len_32 = build_load lenptr "lenptr" builder in
      let len = build_zext len_32 i64_type "len_64" builder in
      let sizeof = size_of value_t in
      let size = build_mul len sizeof "size" builder in
      let newlen = build_add len (const_int i64_type 1) "conslen" builder in
      let newsize = build_mul newlen sizeof "newsize" builder in
      let ptr = build_malloc newsize (pointer_type value_t) "malloc" builder in
      let ptrhead = build_in_bounds_gep ptr [| const_int i32_type 0 |]
                                    "ptrhead" builder in
      let ptrrest = build_in_bounds_gep ptr [| const_int i32_type 1 |]
                                    "ptrrest" builder in
      let tailptr = build_in_bounds_gep tail (idx 4) "ptrhead" builder in
      let tailel = build_load tailptr "tailptr" builder in
      let rawsrc = build_bitcast tailel (pointer_type i8_type)
                              "rawsrc" builder in
      let rawdst = build_bitcast ptrrest (pointer_type i8_type)
                              "rawdst" builder in
      ignore (build_store arg ptrhead builder);
      ignore (build_memcpy rawsrc rawdst size);
      box_llar ptr newlen
  | _ -> raise (Error "Unknown array operator")

and codegen_string_op op s2 =
  let value_t = match type_by_name the_module "value_t" with
      Some t -> t
    | None -> raise (Error "Could not look up value_t") in
  let rharel_type = pointer_type value_t in
  let rhstring_type size = array_type i8_type size in
  let nullterm = const_int i8_type 0 in
  match op with
    "str-join" ->
      let arg = List.hd s2 in
        codegen_call_op "cstrjoin" [arg]
    | "str-split" ->
      let arg = List.hd s2 in
      let str = unbox_str arg in
      let len = unbox_length arg in
      let size = build_mul (size_of rharel_type)
                         len "size" builder in
      let newar = build_malloc size rharel_type "newar" builder in

      let var_name = "i" in
      let loop_lim = box_value len in
      let start_val = codegen_sexpr (Ast.Atom(Ast.Int(0))) in
      let start_bb = insertion_block builder in
      let the_function = block_parent start_bb in
      let loop_bb = append_block context "loop" the_function in
      ignore (build_br loop_bb builder);
      position_at_end loop_bb builder;
      let variable = build_phi [(start_val, start_bb)] var_name builder in
      let old_val =
        try Some (Hashtbl.find named_values var_name) with Not_found -> None
      in
      Hashtbl.add named_values var_name variable;
      (* start body *)
      let loopidx = unbox_int variable in
      let ptr = build_in_bounds_gep str [| loopidx |]
                                  "extract" builder in
      let el = build_load ptr "extractload" builder in
      let strseg = build_alloca (rhstring_type 2) "strseg" builder in
      let strseg0 = build_in_bounds_gep strseg (idx 0) "strseg0" builder in
      let strseg1 = build_in_bounds_gep strseg (idx 1) "strseg1" builder in
      ignore (build_store el strseg0 builder);
      ignore (build_store nullterm strseg1 builder);
      let newptr = build_in_bounds_gep newar [| loopidx |] "arptr" builder in
      ignore (build_store (box_value strseg0) newptr builder);
      (* end body *)
      let next_var = build_add (unbox_int variable)
                              (const_int i64_type 1) "nextvar" builder in
      let next_var = box_value next_var in
      let end_cond = build_icmp Icmp.Slt (unbox_int next_var)
                               (unbox_int loop_lim) "end_cond" builder in
      let loop_end_bb = insertion_block builder in
      let after_bb = append_block context "after_loop" the_function in
      ignore (build_cond_br end_cond loop_bb after_bb builder);
      position_at_end after_bb builder;
      add_incoming (next_var, loop_end_bb) variable;
      begin match old_val with
            Some old_val -> Hashtbl.add named_values var_name old_val
          | None -> ()
      end;
      box_llar newar len
  | "str-length" ->
     box_value (unbox_length (List.hd s2))
  | _ -> raise (Error "Unknown string operator")

and codegen_cf_op op s2 =
  let value_t = match type_by_name the_module "value_t" with
      Some t -> t
    | None -> raise (Error "Could not look up value_t") in
  match op with
    "if" ->
    let truese = match s2 with
        Ast.DottedPair(_, next) -> next
      | _ -> raise (Error "Malformed if expression") in
    let falsese = match truese with
        Ast.DottedPair(_, next) -> next
      | _ -> raise (Error "Malformed if expression") in
    let cond_val = unbox_bool (codegen_one_arg s2) in
    let start_bb = insertion_block builder in
    let the_function = block_parent start_bb in
    let truebb = append_block context "then" the_function in
    position_at_end truebb builder;
    let true_val = codegen_one_arg truese in
    let new_truebb = insertion_block builder in
    let falsebb = append_block context "else" the_function in
    position_at_end falsebb builder;
    let false_val = codegen_one_arg falsese in
    let new_falsebb = insertion_block builder in
    let mergebb = append_block context "ifcont" the_function in
    position_at_end mergebb builder;
    let incoming = [(true_val, new_truebb); (false_val, new_falsebb)] in
    let phi = build_phi incoming "iftmp" builder in
    position_at_end start_bb builder;
    ignore (build_cond_br cond_val truebb falsebb builder);
    position_at_end new_truebb builder; ignore (build_br mergebb builder);
    position_at_end new_falsebb builder; ignore (build_br mergebb builder);
    position_at_end mergebb builder;
    phi
  | "when" ->
     let truese = match s2 with
         Ast.DottedPair(_, next) -> next
       | _ -> raise (Error "Malformed when expression") in
     let cond_val = unbox_bool (codegen_one_arg s2) in
     let start_bb = insertion_block builder in
     let the_function = block_parent start_bb in
     let truebb = append_block context "then" the_function in
     position_at_end truebb builder;
     let true_val = codegen_one_arg truese in
     let new_truebb = insertion_block builder in
     let falsebb = append_block context "else" the_function in
     position_at_end falsebb builder;
     let false_val = const_null (pointer_type value_t) in
     let new_falsebb = insertion_block builder in
     let mergebb = append_block context "ifcont" the_function in
     position_at_end mergebb builder;
     let incoming = [(true_val, new_truebb); (false_val, new_falsebb)] in
     let phi = build_phi incoming "iftmp" builder in
     position_at_end start_bb builder;
     ignore (build_cond_br cond_val truebb falsebb builder);
     position_at_end new_truebb builder; ignore (build_br mergebb builder);
     position_at_end new_falsebb builder; ignore (build_br mergebb builder);
     position_at_end mergebb builder;
     phi
  | "dotimes" ->
     let qs, body = match s2 with
         Ast.DottedPair(Ast.Vector(qs), body) -> qs, body
       | _ -> raise (Error "Malformed dotimes expression") in
     let var_name = match qs.(0) with
         Ast.Atom(Ast.Symbol(s)) -> s
       | _ -> raise (Error "Expected symbol in dotimes") in
     let loop_lim = codegen_sexpr qs.(1) in
     let start_val = codegen_sexpr (Ast.Atom(Ast.Int(0))) in
```

```
      let start_bb = insertion_block builder in
      let the_function = block_parent start_bb in
      let loop_bb = append_block context "loop" the_function in
      ignore (build_br loop_bb builder);
      position_at_end loop_bb builder;
      let variable = build_phi [(start_val, start_bb)] var_name builder in
      let old_val =
        try Some (Hashtbl.find named_values var_name) with Not_found -> None
      in
      Hashtbl.add named_values var_name variable;
      ignore (codegen_sexpr body);
      let next_var = build_add (unbox_int variable)
                               (const_int i64_type 1) "nextvar" builder in
      let next_var = box_value next_var in
      let end_cond = build_icmp Icmp.Slt (unbox_int next_var)
                                (unbox_int loop_lim) "end_cond" builder in
      let loop_end_bb = insertion_block builder in
      let after_bb = append_block context "after_loop" the_function in
      ignore (build_cond_br end_cond loop_bb after_bb builder);
      position_at_end after_bb builder;
      add_incoming (next_var, loop_end_bb) variable;
      begin match old_val with
              Some old_val -> Hashtbl.add named_values var_name old_val
            | None -> ()
      end;
      box_value (const_int i64_type 0)
  | _ -> raise (Error "Unknown control flow operation")

and codegen_call_op f args =
  let callee = match lookup_function f the_module with
      Some callee -> callee
    | None ->
       let v = try Hashtbl.find named_values f with
                 Not_found -> raise (Error ("Unknown function: " ^ f)) in
       unbox_function v
  in
  let args = Array.of_list args in
  build_call callee args "call" builder;

and codegen_binding_op f s2 =
  let old_bindings = ref [] in
  match f with
    "let" ->
    let bindlist, body = match s2 with
        Ast.DottedPair(Ast.Vector(qs), next) -> qs, next
      | _ -> raise (Error "Malformed let") in
    let len = Array.length bindlist in
    if len mod 2 != 0 then
      raise (Error "Malformed binding form in let");
    let bind n a =
      let s = match n with
          Ast.Atom(Ast.Symbol(s)) -> s
        | _ -> raise (Error "Malformed binding form in let") in
      let llaptr = codegen_sexpr a in
      let lla = build_load llaptr "load" builder in
      let the_function = block_parent (insertion_block builder) in
      let alloca = create_entry_block_alloca the_function s in
      ignore (build_store lla alloca builder);
      begin try let old_value = Hashtbl.find named_values s in
                old_bindings := (s, old_value) :: !old_bindings;
            with Not_found -> ()
      end;
      Hashtbl.add named_values s alloca in
    Array.iteri (fun i m ->
                 if (i mod 2 == 0) then
                   bind m (bindlist.(i+1))) bindlist;
    let llbody = codegen_sexpr body in
    List.iter (fun (s, old_value) ->
                Hashtbl.add named_values s old_value
              ) !old_bindings;
    llbody
    | _ -> raise (Error "Unknown binding operator")

and match_action s s2 =
  if StringSet.mem s binding_ops then
    codegen_binding_op s s2
  else if StringSet.mem s cf_ops then
    codegen_cf_op s s2
  else
    let args = extract_args s2 in
    if StringSet.mem s arith_ops then
      codegen_arith_op s args
    else if StringSet.mem s array_ops then
      codegen_array_op s args
    else if StringSet.mem s string_ops then
      codegen_string_op s args
    else
      codegen_call_op s args

and codegen_sexpr s = match s with
    Ast.Atom n -> codegen_atom n
  | Ast.DottedPair(Ast.Atom n, Ast.Atom Ast.Nil) -> codegen_atom n
  | Ast.DottedPair(s1, s2) ->
     begin match s1, s2 with
             (Ast.Atom(Ast.Symbol s), _) -> (* single sexpr *)
             match_action s s2
           | (Ast.DottedPair(Ast.Atom(Ast.Symbol ss1), ss2), _) ->
              begin match s2 with
                    Ast.DottedPair(_, _) ->
                    let _ = match_action ss1 ss2 in
                    codegen_sexpr s2
                    | Ast.Atom(Ast.Nil) -> match_action ss1 ss2
                    | _ -> raise (Error "Sexpr parse error");
              end
           | _ -> raise (Error "Expected function call");
     end
  | Ast.Vector(qs) -> codegen_array qs

and codegen_array qs =
  let value_t = match type_by_name the_module "value_t" with
      Some t -> t
    | None -> raise (Error "Could not look up value_t") in
  let len = Array.length qs in
  let rharray_type size = array_type (pointer_type value_t) size in
  let new_array = build_alloca (rharray_type len) "ar" builder in
  let ptr n = build_in_bounds_gep new_array (idx n) "arptr" builder in
  let llqs = Array.map codegen_sexpr qs in
  Array.iteri (fun i m -> ignore (build_store m (ptr i) builder)) llqs;
  box_value new_array

let codegen_proto p =
  let value_t = match type_by_name the_module "value_t" with
      Some t -> t
    | None -> raise (Error "Could not look up value_t")
  in
  match p with
    Ast.Prototype (name, args) ->
    let args_len = Array.length args in
    let argt = Array.make args_len (pointer_type value_t)  in
    let ft = if args_len == 0 then
                function_type i64_type argt
              else
                function_type (pointer_type value_t) argt in
    let f =
      match lookup_function name the_module with
        None -> declare_function name ft the_module

      (* If 'f' conflicted, there was already something named 'name'. If it
       * has a body, don't allow redefinition or reextern. *)
      | Some f ->
          (* If 'f' already has a body, reject this. *)
          if block_begin f <> At_end f then
            raise (Error "redefinition of function");

          (* If 'f' took a different number of arguments, reject. *)
          if element_type (type_of f) <> ft then
            raise (Error "redefinition of function with different # args");
          f
    in

    (* Set names for all arguments. *)
    Array.iteri (fun i a ->
                  let n = args.(i) in
                  set_value_name n a;
                  Hashtbl.add named_values n a;
                ) (params f);
    f

let codegen_func = function
```

```
  | Ast.Function (proto, body) ->
      Hashtbl.clear named_values;
      let the_function = codegen_proto proto in
      (* Create a new basic block to start insertion into. *)
      let bb = append_block context "entry" the_function in
      position_at_end bb builder;

      try
        let ret_val =
          if Array.length (params the_function) == 0 then
            unbox_int (codegen_sexpr body)
          else
            codegen_sexpr body in

        (* Finish off the function. *)
        let _ = build_ret ret_val builder in
        the_function
      with e ->
        delete_function the_function;
        raise e
```

9.9 REPL: rrhine

```
#!/usr/bin/env python

from os import walk
from subprocess import Popen, PIPE
import sys

class colors:
    HEADER = '\033[95m'
    OK = '\033[92m'
    WARNING = '\033[93m'
    FAIL = '\033[91m'
    END = '\033[0m'

def run():
    stdlib = open("stdlib.rh").read()
    pasting = False
    i = ""
    last_llvm = ""
    last_io = ""
    current_input = ""
    total_input = ""
    total_output = " "
    total_failed_input = ""
    while i not in ["quit", "exit", "q"]:
        if pasting:
            sys.stdout.write(colors.HEADER)
        else:
            sys.stdout.write(colors.END)
        try:
```

```python
            i = raw_input('> ')
            i = i.strip()
            if pasting:
                ci += i + "\n"
                continue
            else:
                ci = i
        except KeyboardInterrupt:
            if pasting:
                pasting = False
                sys.stdout.write(colors.HEADER)
                print "\nPASTED:"
                print ci
            else:
                continue
        except EOFError:
            sys.stdout.write(colors.WARNING)
            print "\n\nThanks for using Rhine! :)"
            exit(1)

        if ci == "":
            continue
        elif ci in ["paste", "p"]:
            sys.stdout.write(colors.HEADER)
            pasting = True
            ci = ""
            print "\n\nNow in paste mode..."
            continue
        elif ci in ["llvm", "ll"]:
            print last_llvm
            continue
        elif ci in ["i", "input"]:
            print total_input
            continue
        elif ci in ["o", "output"]:
            print total_output
            continue
        elif ci in ["last"]:
            print last_io
            continue

        p = Popen(["./rhine", "-"], stdin=PIPE, stdout=PIPE, stderr=PIPE)
        output = ""
        stdout, stderr = p.communicate(stdlib+"\n"+total_input+"\n"+ci)

        if stderr.find("error") >= 0:
            sys.stdout.write(colors.FAIL)
            print stderr.strip()
            continue
        else:
            total_input += "\n"+ci
            for line in stdout.split("\n"):
                if line.startswith("Evaluated to ") or line.strip() == "":
                    continue
                output += line.strip()+"\n"


            from_idx = len(total_output)-1
            if from_idx < 0:
                from_idx = 0
            print output[:-1][from_idx:]
            total_output = output

            last_io = ci+"\n\n"+output[:-1][len(total_output):]
            last_llvm = stderr


if __name__ == "__main__":
    run()
```

9.10 bindings.c

```

#include <stdio.h>
#include <stdlib.h>
#include <stdbool.h>
#include <math.h>
#include <string.h>

struct value_t {
     int type_tag;
     long int_val;
     bool bool_val;
     char *string_val;
     struct value_t **array_val;
     long array_len;
     double dbl_val;
};

void print_atom(struct value_t *v) {
     int i;
     if (!v) {
          printf("(nil)");
          return;
     }
     switch(v->type_tag) {
     case 1:
          printf("(int) %ld", v->int_val);
          break;
     case 2:
          printf("(bool) %s", !v->bool_val ? "false" : "true");
          break;
     case 3:
          printf("(string len %ld) %s", v->array_len, v->string_val);
          break;
```

```
    case 4:
        printf("(array len %ld) [", v->array_len);
        for (i = 0; i < v->array_len; i++) {
            struct value_t *el = (v->array_val)[i];
            print_atom(el);
            printf(";");
        }
        printf("]");
        break;
    case 6:
        printf("(double) %f", v->dbl_val);
        break;
    default:
        printf("Don't know how to print type %d", v->type_tag);
    }
}

extern struct value_t *print(struct value_t *v) {
    struct value_t *ret;
    print_atom(v);
    ret = malloc(sizeof(struct value_t));
    ret->int_val = 0;
    ret->type_tag = 1;
    return ret;
}

extern struct value_t *println(struct value_t *v) {
    struct value_t *ret = print(v);
    printf("\n");
    return ret;
}

double get_value(struct value_t *v, int *ret_type) {
    if (v->type_tag == 6) {
        *ret_type = 6;
        return v->dbl_val;
    } else {
        return (double) v->int_val;
    }
}

struct value_t *save_value(double val, int ret_type) {
    struct value_t *ret;
    ret = malloc(sizeof(struct value_t));

    if (ret_type == 6) {
        ret->dbl_val = val;
        ret->type_tag = 6;
    } else if (ret_type == 2) {
        ret->bool_val = (bool) val;
        ret->type_tag = 2;
    } else {
        ret->int_val = (int) val;
        ret->type_tag = 1;
    }
    return ret;
}

extern struct value_t *cadd(struct value_t *v, struct value_t *v2) {
    int ret_type = 0;
    double val = get_value(v, &ret_type) + get_value(v2, &ret_type);
    struct value_t *ret = save_value(val, ret_type);
    return ret;
}

extern struct value_t *cdiv(struct value_t *v, struct value_t *v2) {
    int ret_type = 6; // force float
    double val = get_value(v, &ret_type) / get_value(v2, &ret_type);
    struct value_t *ret = save_value(val, ret_type);
    return ret;
}

extern struct value_t *csub(struct value_t *v, struct value_t *v2) {
    int ret_type = 0;
    double val = get_value(v, &ret_type) - get_value(v2, &ret_type);
    struct value_t *ret = save_value(val, ret_type);
    return ret;
}

extern struct value_t *cmul(struct value_t *v, struct value_t *v2) {
    int ret_type = 0;
    double val = get_value(v, &ret_type) * get_value(v2, &ret_type);
    struct value_t *ret = save_value(val, ret_type);
    return ret;
}

extern struct value_t *clt(struct value_t *v, struct value_t *v2) {
    int ret_type = 0;
    struct value_t *ret;
    if (get_value(v, &ret_type) < get_value(v2, &ret_type)) {
        ret = save_value(1.0, 2);
    } else {
        ret = save_value(0.0, 2);
    }
    return ret;
}

extern struct value_t *cgt(struct value_t *v, struct value_t *v2) {
    int ret_type = 0;
    struct value_t *ret;
    if (get_value(v, &ret_type) > get_value(v2, &ret_type)) {
        ret = save_value(1.0, 2);
    } else {
        ret = save_value(0.0, 2);
    }
    return ret;
```

```c
}

extern struct value_t *clte(struct value_t *v, struct value_t *v2) {
     int ret_type = 0;
     struct value_t *ret;
     if (get_value(v, &ret_type) <= get_value(v2, &ret_type)) {
          ret = save_value(1.0, 2);
     } else {
          ret = save_value(0.0, 2);
     }
     return ret;
}

extern struct value_t *cgte(struct value_t *v, struct value_t *v2) {
     int ret_type = 0;
     struct value_t *ret;
     if (get_value(v, &ret_type) >= get_value(v2, &ret_type)) {
          ret = save_value(1.0, 2);
     } else {
          ret = save_value(0.0, 2);
     }
     return ret;
}

extern struct value_t *cequ(struct value_t *v, struct value_t *v2) {
     int ret_type = 0;
     struct value_t *ret;
     // only makes sense for integers and bools currently
     switch(v->type_tag) {
     case 1:
          if (v->int_val == v2->int_val) {
               ret = save_value(1.0, 2);
          } else {
               ret = save_value(0.0, 2);
          }
          break;
     case 2:
          if (!v->bool_val == !v2->bool_val) {
               ret = save_value(1.0, 2);
          } else {
               ret = save_value(0.0, 2);
          }
          break;
    case 4:
        if (v->array_len == v2->array_len) {
            ret = save_value(1.0, 2);
        } else {
            ret = save_value(0.0, 2);
        }
        break;
    case 3:
        if (v->array_len == v2->array_len && memcmp(v->array_val, v2->array_val, sizeof(char)*v->arra
            ret = save_value(1.0, 2);
        } else {
            ret = save_value(0.0, 2);
        }
        break;
    }
     return ret;
}

extern struct value_t *cand(struct value_t *v, struct value_t *v2) {
     int ret_type = 0;
     struct value_t *ret;
     if (!(!v->bool_val) && !(!v2->bool_val)) {
          ret = save_value(1.0, 2);
     } else {
          ret = save_value(0.0, 2);
     }
     return ret;
}

extern struct value_t *cor(struct value_t *v, struct value_t *v2) {
     int ret_type = 0;
     struct value_t *ret;
     if (!(!v->bool_val) || !(!v2->bool_val)) {
          ret = save_value(1.0, 2);
     } else {
          ret = save_value(0.0, 2);
     }
     return ret;

}

extern struct value_t *cnot(struct value_t *v) {
     int ret_type = 0;
     struct value_t *ret;
     if (!(!v->bool_val)) {
          ret = save_value(0.0, 2);
     } else {
          ret = save_value(1.0, 2);
     }
     return ret;
}

extern struct value_t *cmod(struct value_t *v, struct value_t *v2) {
     int ret_type = 1;
     int val = v->int_val % v2->int_val;
     struct value_t *ret = save_value(val, ret_type);
     return ret;

}

extern struct value_t *cexponent(struct value_t *v, struct value_t *v2) {
     int ret_type = 0;
     double val = pow(get_value(v, &ret_type),get_value(v2, &ret_type));
     struct value_t *ret = save_value(val, ret_type);
     return ret;
}


extern struct value_t *cstrjoin(struct value_t *v) {
    struct value_t *ret;
    int i = 0;
    ret = malloc(sizeof(struct value_t));
    ret->string_val = malloc(sizeof(char) * (v->array_len+1));
    ret->type_tag = 3;
    ret->array_len = v->array_len;
    while (i < v->array_len) {
        ret->string_val[i] = *((v->array_val)[i]->string_val);
        i++;
    }
    *(ret->string_val+i) = '\0';


    return ret;
}
```

9.11 run_tests.py

```
from os import walk, unlink
from subprocess import Popen, PIPE

class colors:
    HEADER = '\033[95m'
    OK = '\033[92m'
    WARNING = '\033[93m'
    FAIL = '\033[91m'
    END = '\033[0m'

def get_test_files():
    filelist = []
    for (dirpath, dirnames, filenames) in walk("tests/"):
        filelist.extend(filenames)
    return filelist

def run():
    stdlib = open("stdlib.rh").read()
    tests = get_test_files()
    no_tests = len(tests)
    no_successes = 0

    print colors.HEADER
    print "--------------------------"
    print "RUNNING TESTS..."
    print "--------------------------"
    print colors.END
    for filename in tests:
        if filename.startswith("output.") or not filename.endswith(".rht"):
            no_tests -= 1
            continue
        contents = open("tests/"+filename).read().split("---")
        test_input = contents[0].strip()
        expected_output = "---".join(contents[1:]).strip().lower()

        p = Popen(["./rhine", "-"], stdin=PIPE, stdout=PIPE, stderr=PIPE)
        stdout, stderr = p.communicate(stdlib+"\n"+test_input)
        stdout = stdout.lower()
        stderr = stderr.lower()

        if stderr.find(expected_output) >= 0 or stdout.find(expected_output) >= 0:
            no_successes += 1
            try:
                unlink('tests/output.'+filename)
            except OSError:
                pass
            print "%s %s: SUCCESS%s" % (colors.OK, filename, colors.END)
        else:
            f = open('tests/output.'+filename,'w')
            f.write(stdout+stderr)
            f.close()
            print "%s %s: FAILED (output written to tests/)" % (colors.FAIL, filename)

    if no_successes == no_tests:
        print colors.OK
    else:
        print colors.FAIL


    print "--------------------------"
    print "TOTAL TESTS: %i" % no_tests
    print "SUCCESSES: %i" % no_successes
    print "FAILS: %i" % (no_tests - no_successes)
    print "--------------------------"
    print colors.END



if __name__ == "__main__":
    run()
```