IB Extended Essay

Computer Science HL:

An Implementation of Scheme Interpreter (R^5RS*)

Zhang Xinyuan

2013

*Revised^5 Report on the Algorithmic Language Scheme

Abstract

Abstract

This essay is about the research and study of scheme programming language interpreter implementing.

Functional programming paradigm is one of the most mathematically beautiful things I met in computer science. Scheme is one of the most pure programming languages. I like this two things always. Thus I tried to learn more about functional programming and implemented a scheme interpreter by myself.

In this essay, I will bring up what I have done and what I have learnt from this project. Practically I tried two times on implementing this scheme interpreter. I'll separate a chapter to discuss how the first trials failed and the improvement brought by the current one.

This interpreter is named *revo*. The word '*revo*' means 'dream' in Esperanto. Originally I was planned to implement an operating system names *fantazio*, which means 'fantasy', with *revo* as it's primitive interpreter. But as far *fantazio* project has not been started.

The codes of the project is open-sourced under MIT license. You may get them from https://github.com/shouya/revo, Firstly of all, you could see the ascii-cast of a demonstration of the interpreter I recorded months ago to preview what you can do in this interpreter.[1]

[1] This demonstration is just for the REPL, the main interpreter is to execute source file

Contents

1 Specification

1.1 Functional Programming Paradigm

In functional programming, a function is a first-class member. A function treated as a regular data type, no more special than an integer or a string. It is both data and code, as it's executable.

To program in functions gives me a feeling like to translate complex programming procedures into mathematical patterns. The steps and manner to abstract a procedure into a function was marvelous. To keep a function pure, out of interference by external environment, to get rid of states and mutable variables, programmer should focus on the function's application. The things you need to consider is the what the function is and how it is defined, just in mathematical language, in contrast to imperative programming paradigm, in which you need to think of translating an obvious declaration into intelligible statements.

Due to its simplicity and flexibility,

1.2 Lisp Specification

Lisp is an old programming language which a bunch of revolutionary features. Such as garbage collection, dynamic typing,

1.3 Scheme Specification

2 From Source Code to Syntax Tree

The process of forming a abstract syntax tree from source code is usually done by two part of works. First the source code will be scanned and split into small tokens with singular meaning. This part of job is done by an advanced string parser, which is what we called lexical scanner. After proceeded by a lexical scanner, or lexer for simple, the code will be in a number of linear tokens. Then we form these tokens into an abstract syntax tree. By following the syntax of a programming language, a source code could have its meaning. For example, consider the following operation code: `1+1`. The lexical scanner will turn it into token list: an integer `1`, a symbol `+`, and another integer `1`. Then the syntax parser will recognize the this code into an binary plus operation, it takes two integer `1` as its operands.

2.1 Lexical Scanner

The lexical scanner is written completely by myself. Firstly LISP has very few types of token that is quite simple so it might be not necessary to use a lexical scanner generator like lex[2]. I wish I can hold more controls on the scanner since I will implement a REPL[3] and it requires to communicate with the lexical scanner.

My lexical scanner is implemented as a class. An instantiated scanner can scan source code from standard input, string, file, or any IO stream.

For most basic usage, this scanner can be regarded as an enumerable object, so you can just specify the input and then fetch an array of tokens directly from it. (usage example)

The principle of this lexical scanner is simple. It load the source code string into memory, and set the position pointer to the head of the string. Then it traverse the lexical rule list until it find one that matches the string from current position. Afterward it do action specified by that rule, and move the position pointer to skip the matched text. The scanner loops then, until the it meets EOF(end-of-file).

A technical problems need to be noticed is the *state*. A scanner could behave differently when it meets the same text under different state. This method is necessary to cope with complex syntax patterns. For example, the text `if` means differently in a direct context, string context, or a comment context. What the context usually deliminators do is transferring state. (code example: string context, comment context)

[2] lex: https://en.wikipedia.org/wiki/Lex_(software), or its successor lex. The ruby port of lex rex.
[3] Read-evaluate-print loop, an interactive interface for code executing

Each token the scanner parsed produce a pair in format of $[type, value]$, and for the last token, EOF, it returns $[false, false]$. This agreement is specified by racc for the syntax parser to recognize what a token is.

2.2 Syntax Parser

The syntax parser is to form an abstract syntax tree from a linear token composition produced by a lexical scanner. The most popular syntax parser is YACC[4], or it's GNU successor, bison. The ruby port of YACC is called racc. This is what I adopt as the parser generator for *revo*.

racc has a DSL(domain specific language) for the syntax rule file. Similar to YACC, a syntax rule can be defined in regular BNF(Backus–Naur Form). With such a powerful parser generator, all I need to do is specify the various reduction rules for LISP. LISP has very simple syntax, so the syntax specification is short as well. Mainly it consists just a group of rules for LISP atoms, which includes basic data types and identifiers, and another for the list syntax. An atom or a list is reduced into a LISP expression.

Everything will be finally reduced to a LISP expression. Strictly, we don't consider a bunch of LISP expressions, or no expression, as a single expression to be returned by the syntax parser. In a regular scheme source file, it is normal to have more than two expressions. The way to solve this problem is the `begin` operator. The `begin` operator in Scheme takes any number of expressions as argument, execute them and return the value of the last one. That part of rules are illustrated below(16source position):

```
        main: multi_expr        { val[0] }
            | expr              { val[0] }
            | /* empty */       { NULL }

  multi_expr: multi_expr_x      { Cons[:begin, val[0]] }

multi_expr_x: expr expr         { Cons[val[0], Cons[val[1], NULL]] }
            | expr multi_expr_x { Cons[val[0], val[1]] }
```

So far the code is formed well into a syntax tree. What I do in this section is like to build a body for the plain text source. The next step, the runtime, is a kind of magical — to make it come alive.

[4]YACC: Yet Another Compiler Compiler

3 Runtime

The usage of a runtime is to create a space for the code to execute. Basically a runtime provides a scope stack for the local variables to store, and trace the call stack for the function invoking.

3.1 Scope Model

Scopes in *revo* is only for storing local symbols, which is generally known as variables. Technically, these symbols are stored in a hash table, and is referable with their names.

3.1.1 Free Variables

Usually, a scope has a parent, this is what we called 'scope nesting'. When the program requested a variable in a scope, it will firstly search its own symbol table to see if the variable is defined locally. If that variable is not found, it will ask its parent for it. It does this recursively until the variable is found, or it will raise a '*name-not-found*' exception and interrupt the execution.

A creation of a scope is done by function closure evaluation. A function closure is usually known as a lambda(λ) in functional programming. Concisely a closure could be regarded as a composition of a function, which is a bunch of codes with optional parameters to accept input and returns a value, and a scope pointer. A lambda will reserve the pointer to the scope where it is created, and when it evaluates, a new scope will be created and its parent will be set as the one it reserves.

So consider this code snippet:

```
(define my-lambda       ;; define the lambda with the name 'my-lambda'
  (let ((x 1))          ;; start a local scope and set x to 1
    (lambda () (x))))   ;; create a lambda that return the value of x
(define x 2)            ;; define a global variable x, set it to 2

(display (my-lambda))   ;; it shows 1 instead of 2
```

Because `my-lambda` is created under the scope where `x` is set to `1`, it will regard this as the parent to search

for variables, no matter what scope it is evaluated at.

The situation I mentioned above is the way how *revo* deal with free variables. A free variable, as its name, it not bounded on any specified value. The concept might be a little complicated to understand, but I'll explain with a simple example:

```
(lambda (a b) (+ a b c))
```

In this example, it is a anonymous lambda with two parameters `a` and `b`. When we invoke this lambda, we have to specify two arguments for it, let's say, `1` and `2`. In this case, `1` will be bounded on `a`, and `2` will be bounded on `b`. For the name `c`, the variable it represent is not designated when the lambda is invoked, so this is what we called 'free variable'.

3.1.2 Bounded Variable

I used a quick way to cope with bounded variables for lambdas. In the new scope created by a called lambda, I will put a group of variables with the name of its parameters and the value of the arguments passed into the symbol table of the scope. So referring its arguments in the body is right like referring local variables. Actually there's no concept of local scope in Scheme. Usually, the operator `let` to start a local scope is also done by lambda. In the standard template syntax, `let` is defined like:

```
(define-syntax let
  (syntax-rules ()
    ((let ((name val) ...)
       body ...)
     ((lambda (name ...) body ...)
      val ...))))
```

So as shown, `(let ((a 41)) (+ a 1))` is equivalent to `((lambda (a) (+ a 1)) 41)`, which is not different from calling a temporary lambda.

3.2 Call Stack

Call stack is an important component of a runtime for most programming language interpreters/compilers. Technically, in a call stack we record function calls and when a function returns, it will pop the top, in which stores an address of where the function is called. The instruction pointer(suppose there is one) will jump to that address to continue the execution of the program.

In *revo*, I didn't record a call stack specifically. When a closure is invoked, it will directly store the bounded variables with their arguments passed to the closure in a new scope, then it just simply evaluate the body. In a imperative programming language, a call will usually start a completely new scope to store variables. However, in *revo*, there is actually no pure function without a binding[5]. For a closure, a new scope for it is stacked on the top of the scope where it is created, another reason that call stack is not necessarily needed is the functional programming specification. Typically, Scheme doesn't have any interrupt allowed(ignore `call/cc`), that means, there is no `break`, `goto`, or even `return` statements. The main purpose to keep a call stack is for `return` statements to jump among the function stacks.

3.3 Macro: Template Expansion

The template system is one of my favorite features in Scheme programming language. A template in Scheme is usually known as a *macro*. Macro gives lisp family ultra-magical power to form the code. Generally, a macro rule holds a pattern of code and a transformer. To process a macro, a macro matcher will firstly scan the code and match it with a macro rule using its specified pattern, then it will apply the transformation.

Unlike a function, a macro take *codes* rather than *values* as its parameters. The codes passed to it will not be evaluated before the macro expansion. Therefore it has meta-power to control the code.

Macros syntax in Scheme is like:

```
(define-syntax my-if (else)
  (syntax-rules
    ((my-if cond body ...)
     (if cond (begin body ...) nil))
    ((my-if cond if-part ... else else-part ...)
```

[5] Frankly, there is. You can define this kind of functions in ruby, as primitive functions in *revo*.

```
(if cond (begin if-part ...) (begin else-part ...)))))
```

So when I call it with `(my-if 1 'a else 'b)`, it will be transformed to `(if 1 (begin 'a) (begin 'b))`. This process will be done before the final code evaluation. My implementation is slightly different. Since *revo* is an interpreter than a compiler, the compile-time and the run-time is combined together. I deal with macros as long as I deal with other callable objects in the run-time.

The matcher and the transformer[6] are separate parts. When a macro is found called somewhere, the macro processor will invoke the bunch of matchers for that macro; then it will return the one that matches the pattern. The matcher scan pattern and relate different parts of code with the names specified in the macro definition. This relation table will be passed to the corresponding transformer, and the transformer restore the snippet of code with the transformation rules defined according to the table. Finally the snippet generate will replace the original macro calling, and will be evaluated as a regular code.

3.3.1 Unhygienic Macro

In a programming language that has unhygienic macro support, it is possible to poison the external binding from macro expansion. Unhygienic macro is a kind of black magic that you can use its power to construct things beyond, but if it is mishandled, its will cause unexpected results.

Macros in Common Lisp is unhygienic. R^{5}RS specified that macros in Scheme should be hygienic. In order to give programmer more freedom, I added an option in *revo*'s implementation to support unhygienic macros.

Unhygienic macro is relative easy to implement. All need to do is just substitute the names with it's

3.3.2 Hygienic Macro

http://lifegoo.pluskid.org/?p=418 In opposite to unhygienic macros, hygienic macro has no side effect

In specification document, it states:

> If a macro transformer inserts a binding for an identifier (variable or keyword), the identifier will in effect be renamed throughout its scope to avoid conflicts with other identifiers.
>
> –R^{5}RS

[6] In *revo* it is called as a template.

That means a

4 Bibliography

References

[1] Leslie Lamport, *LaTeX: A Document Preparation System*. Addison Wesley, Massachusetts, 2nd Edition, 1994.

[2] Creative Common, *Hygienic Macro*. https://en.wikipedia.org/wiki/Hygienic_macro.