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 word `if` means differently in a direct context, string context, or a comment context. What the context usually deliminators do is transfer 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 for the syntax parser to recognize what a token is, and specified by racc.

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 Bibliography

References

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