Scheme Interpreter(PPCA 2013 Summer Project)

赵卓越

2013.8

Contents

1 Introduction **2**

2 Build **2**

3 Run **3**

4 Implementation features **3**

4.1 Syntax 3

4.2 Data Types 6

4.3 Standard procedures 7

4.4 Other implementation features 13

5 Acknowledgement **13**

1 Introduction

Scheme Interpreter is an implementation of a subset of Revised[5] Report on the Algorithmic Language Scheme (R5RS). It supports most syntax, data types and procedures defined by R5RS. The project is written in C++. Due to the use of multiple C++11 features, a compiler that supports C++11 is needed to build the project. GNU GMP, GNU MPFR, GNU MPC are introduced to implement the number types. These libraries are distributed under the Gnu Lesser General Public License.

2 Build

There is a Code::Blocks project file provided. It was created by Code::Blocks IDE 12.11. The project was successfully complied by GNU GCC 4.7.3 on Linux Mint 15 Olivia 32-bit. It was also successfully compiled by MinGW GCC 4.7.2 on Windows 7 x86, which need a patch to enable std::to_string and std::stoi. Refer to http://tehsausage.com/mingw-to-string.

The debug mode compile and linking options are:

g++ -Wall -fexceptions -std=c++11 -g -l:libmpc.a -l:libmpfr.a -l:libgmpxx.a -l:libgmp.a

The release mode compile and linking options are:

g++ -Wall -fexceptions -O2 -std=c++11 -DNDEBUG -l:libmpc.a -l:libmpfr.a -l:libgmpxx.a -l:libgmp.a

If you want additional dubug infomation, define marco

_SCMINTERP_DEBUG_PRINT_

It will turn on file log which records each evaluation in run.log, and print the type of return value of each evaluation. However, when running complicated user-defined procedures, definging the macro may led to a huge performance loss and rapid consumption of disk space because of the recording of evalution. The scheme interpreter project used 3 thrid-party library: GNU GMP, GNU MPFR, GNU MPC. These libraries needs to be installed on your system. In the cbp file, the linking options are all static:

-l:libmpc.a -l:libmpfr.a -l:libgmpxx.a -l:libgmp.a

For more details on these libraries, refer to their websites:

http://gmplib.org

http://www.mpfr.org

http://www.multiprecision.org

source files are:

./builtin/*

./interp/*

./object/*

./parser/*

./prompt/*

config.cpp

config.h

3 Run

schemeInterpreter [OPTIONS]...

Valid options:

-h : Show the help and exit.

-l <filename> : Load the specified file before interaction begins. Multiple files can be loaded by providing multiple -l parameters. Files would be loaded in the sequence they're provided. By default, no file will be loaded.

-f <precision> : Set the floating point precision. Input that is less than 64 will be ignored. The default precision is 128. Only the last valid input will take effect.

-cl <length> : Set the maximum length of inexact number literal that would be stored as integer or rational. The default length is 10. Set length to 0 to forbid such conversion.

-mp <length> : Set the maximum length of prefix zero (which does not include the zero before the decimal dot) that a real number will be displayed. If a real number cannot be displayed in fixed point form without exceeding the limit, it will be displayed in scientific form. The default length is 10. Set length to 0 to disable prefix zero.

-np : Disable protection of builtin symbols.

If there is no options provided, the program will use the default configuration and directly enter interaction mode. Otherwise, the options are processed sequentially and enter the interaction mode using the provided configuration.

When the interpreter enters interaction mode, the user can type any scheme expressions. Errors of parsing is reported as early as possible, but input following the error will still be parsed. After a successful parsing, the parsing result will be evaluated, any evaluation error are reported as early as possible. After a successful evaluation, the return value is printed in the form of external representation. Each output begins with " ==> " and ends with a newline.

Note: -f -cl -mp -np options could have significant impact on the behaviour of the interpreter.

4 Implementation features

Most syntax, data types and procedures defined by R5RS are implemented, except hygienic macro, continuation, do syntax, vector, port and procedures related to them. Supported and unsupported features, minor behaviour difference from R5RS, unspecified behaviour are given in this section.

If a supported feature is not described, it is thought to conform to R5RS, while unspecified behaviour and violation to R5RS are give under the features. For details on the standard of the feature, refer to R5RS.

4.1 Syntax

- R5RS Section 4.1.1 Varibale references:
  - <variable> syntax

- R5RS Section 4.1.2 Literal expressions:
  - (quote <datum>) syntax
  - '<datum> syntax
  - <constant> syntax
- R5RS Section 4.1.3 Procedure calls
  - (<operator> <oprand$_1$> ...) syntax

    R5RS claims that the evaluation of operator and operands are in an unspecified consistent order. In this implementation, the evaluation order is always from left to right.
- R5RS Section 4.1.4 Procedures
  - (lambda <formals> <body>) syntax

    Note: In this implementation, lambda do not perform syntax check on <body>. Syntax check of <body> only happens during a procedure call.
- R5RS Section 4.1.5 Conditionals
  - (if <test> <consequent> <alternate>) syntax
  - (if <test> <consequent>) syntax

    R5RS claims that if <test> yields a false value and no <alternate> is specified, then the result of the expression is unspecified. In this implementation, the return value is an object of void type.
- R5RS Section 4.1.6 Assignments
  - (set! <variable> <expression>) syntax

    R5RS claims that the result of the set! expression is unspecified. In this implementation, the return value is an object of void type.

    There is no claim that whether variables of builtin procedures could be assigned. In this implementation, such assignment is disallowed by default, or it can be enabled by -np option.
- R5RS Section 4.2.1 Conditionals
  - (cond <clause$_1$> ...) library syntax

    R5RS requires cond to have at least one clause. In this implementation, no clause cond is allowed and it returns an object of void type. If no clause's test evaluate to true value, cond returns an object of void type.
  - (case <key> <clause$_1$> ...) library syntax

    R5RS requires case to have at least one clause. In this implementation, no clause cond is allowed and it returns an object of void type. If no clause's test evaluate to true value, cond returns an object of void type.

Note: case always uses the builtin eqv?, assignment to eqv? never affects the behaviour of case.

- (and <test$_1$> ...) library syntax
- (or <test$_1$> ...) library syntax

- R5RS Section 4.2.2 Binding Constructs
  - (let <bindings> <body>) library syntax

    R5RS claims that the evaluation of <init>s is in unspecified order. In this implementation, the order is always from left to right.
  - (let* <bindings> <body>) library syntax
  - (letrec <bindings> <body>) library syntax

    R5RS claims that the evaluation of <init>s is in unspecified order. In this implementation, the order is always from left to right.

    R5RS requires evaluating referencing any <variable> in <bindings> before all <init> have been evaluated to be an error. In this implementation, it the result of reference is evaluated to an object of void, which in common cases could probably lead to an error of type mismatch.
- R5RS Section 4.2.3 Sequencing
  - (begin <expression$_1$> ...) library syntax

    R5RS requires begin followed by at least one expression. In this implementation, no expression following begin is allowed, which will be evaluated to an object of void type.
- R5RS Section 4.2.4 Iteration
  - (do ((<variable$_1$> <init$_1$> <step$_1$>) ...)
    (<test> <expression> ...) <command> ...) library syntax

    Unsupported.
  - (let <variable> <bindings> <body>) library syntax

    R5RS claims that the evaluation of <init>s is in unspecified order. In this implementation, the order is always from left to right.
- R5RS Section 4.2.5 Delayed evaluation
  - (delay <expression> ...) library syntax

    R5RS requires delay followed by only one expression. In this implementation, multiple expressions are allowed. They will be evaluated from left to right when forced and result is the value of the last expression.
- R5RS Section 4.2.6 Quasiquotation

  - (quasiquote <qq template>) syntax
  - ‘<qq template> syntax

- R5RS Section 4.3 Macros

  Unsupported. Related syntax is not supported either, including let-syntax, letrec-syntax, syntax-rules, define-syntax.

- R5RS Section 5.2 Definitions

  - (define <variable> <expression>) syntax
  - (define (<variable> <formals>) <body>) syntax
  - (define (<variable> . <formal>) <body>) syntax

    R5RS requires definitions to appear at the top level or at the beginning of a <body>. In this implementation, the restriction is removed and definitions can appear at any place that requires an expression. R5RS did not specify the return value, which is an object of void in this implementation.

4.2 Data Types

Supported data types:

- Numbers
  - Complex
  - Real
  - Rational
  - Integer
- Booleans
- Pairs and lists
- Symbols
- Characters
- Strings
- Promises
- Procedures
- Void (To represent unspecified values)

Data types with limited support:

- Environment (Currently cannot be acquired by users)

Unsupported data types:

- Vectors
- Continuations
- Ports

4.3 Standard procedures

- R5RS Section 6.1 Equivalence predicates
  - (eqv? obj_1 obj_2) procedure
  - (eq? obj_1 obj_2) procedure

    In this implementation, eq? functions the same as eqv?.
  - (equal? obj_1 obj_2) library procedure
- R5RS Section 6.2.5 Numerical operations
  - (number? obj) procedure
  - (complex? obj) procedure
  - (real? obj) procedure
  - (interger? obj) procedure
  - (exact? obj) procedure
  - (inexact? obj) procedure
  - (= z_1 z_2 z_3 ...) procedure
  - (< x_1 x_2 x_3 ...) procedure
  - (> x_1 x_2 x_3 ...) procedure
  - (<= x_1 x_2 x_3 ...) procedure
  - (>= x_1 x_2 x_3 ...) procedure
  - (zero? z) library procedure
  - (positive? x) library procedure
  - (negative? x) library procedure
  - (odd? n) library procedure
  - (even? n) library procedure
  - (max x_1 x_2 ...) library procedure
  - (min x_1 x_2 ...) library procedure

– (+ z_1 ...) procedure

– (* z_1 ...) procedure

– (- z_1 z_2) procedure

– (- z_1) procedure

– (- z_1 z_2 ...) optional procedure

– (- z_1 z_2) procedure

– (- z_1) procedure

– (- z_1 z_2 ...) optional procedure

– (abs x) library procedure

– (quotient n_1 n_2) procedure

– (remainder n_1 n_2) procedure

– (modulo n_1 n_2) procedure

– (gcd n_1 ...) library procedure

– (lcm n_1 ...) library procedure

– (numerator q) procedure

– (denominator q) procedure

– (floor x) procedure

– (ceiling x) procedure

– (truncate x) procedure

– (round x) procedure

– (exp z) procedure

– (log z) procedure

– (sin z) procedure

– (cos z) procedure

– (tan z) procedure

– (asin z) procedure

– (acos z) procedure

– (atan z) procedure

– (atan y x) procedure

– (sqrt z) procedure

– (expt z_1 z_2) procedure

– (make-rectangular x_1 x_2) procedure

  - (make-polar x_3 x_4) procedure
  - (real-part z) procedure
  - (imag-part z) procedure
  - (magnitude z) procedure
  - (angle z) procedure
  - (exact->inexact z) procedure
  - (inexact->exact z) procedure

  Unsupported features:

  - (rationalize x y) (Unsupported) library procedure

- R5RS Section 6.2.6 Numerical input and output
  - (number->string z) procedure
  - (number->string z *radix*) procedure
  - (string->number *string*) procedure
  - (string->number *string* *radix*) procedure
- R5RS Section 6.3.1 Booleans
  - (not *obj*) library procedure
  - (boolean? *obj*) library procedure
- R5RS Section 6.3.2 Pairs and lists
  - (pair? *obj*) procedure
  - (cons obj_1 obj_2) procedure
  - (car *pair*) procedure
  - (cdr *pair*) procedure
  - (set-car! *pair*) procedure

    R5RS claims that return value is unspecified. In this implementation, an object of void type is returned.
  - (set-cdr! *pair*) procedure

    R5RS claims that return value is unspecified. In this implementation, an object of void type is returned.
  - (caar *pair*) library procedure
  - (cadr *pair*) library procedure

    ⋮

- (cdddar *pair*) library procedure
- (cddddr *pair*) library procedure
- (null? *obj*) library procedure
- (list? *obj*) library procedure
- (list *obj* ...) library procedure
- (length *list*) library procedure
- (append *list* ...) library procedure
- (reverse *list*) library procedure
- (list-tail *list k*) library procedure
- (list-ref *list k*) library procedure

  Index k in list-tail and list-ref is assumed to fit in an int (usually at most $2^{32} - 1$).
- (memq *obj list*) library procedure
- (memv *obj list*) library procedure
- (member *obj list*) library procedure
- (assq *obj alist*) library procedure
- (assv *obj alist*) library procedure
- (assoc *obj alist*) library procedure

- R5RS Section 6.3.3 Symbols
  - (symbol? *obj*) procedure
  - (symbol->string *symbol*) procedure
  - (string->symbol *string*) procedure
- R5RS Section 6.3.4 Characters
  - (char? *obj*) procedure
  - (char=? $char_1$ $char_2$) procedure
  - (char<? $char_1$ $char_2$) procedure
  - (char>? $char_1$ $char_2$) procedure
  - (char<=? $char_1$ $char_2$) procedure
  - (char>=? $char_1$ $char_2$) procedure
  - (char-ci=? $char_1$ $char_2$) library procedure
  - (char-ci<? $char_1$ $char_2$) library procedure
  - (char-ci>? $char_1$ $char_2$) library procedure

- (char-ci<=? $char_1$ $char_2$) library procedure
- (char-ci>=? $char_1$ $char_2$) library procedure
- (char-alphabetic? *char*) library procedure
- (char-numeric? *char*) library procedure
- (char-whitespace? *char*) library procedure

  white-space is extended to space( ), horizontal tab(\t), line feed(\n), vertical tabl(\v), form feed(\f), carriage return(\r)
- (char-upper-case? *letter*) library procedure
- (char-lower-case? *letter*) library procedure
- (char->integer *char*) procedure
- (integer->char n) procedure

  Currently this implementation only supports ASCII characters.
- (char-upcase *char*) library procedure
- (char-downcase *char*) library procedure

- R5RS Section 6.3.5 Strings
  - (string? *obj*) procedure
  - (make-string k) procedure
  - (make-string k *char*) procedure

    R5RS claims that if the second argument *char* is not given, the contents of the string are unspecified. In this implementation, the string is filled with null character(\0).
  - (string *char* ...) library procedure
  - (string-length *string*) procedure
  - (string-ref *string* k) procedure
  - (string-set! *string* k *char*) procedure

    The index k in string-ref and string-set! is assumed to fit in an int (usually at most $2^{31} - 1$).

    R5RS claims that string-set! returns an unspecified value. In this implementation, it returns an object of void type.
  - (string=? $string_1$ $string_2$) library procedure
  - (string-ci=? $string_1$ $string_2$) library procedure
  - (string<? $string_1$ $string_2$) library procedure
  - (string>? $string_1$ $string_2$) library procedure
  - (string<=? $string_1$ $string_2$) library procedure
  - (string>=? $string_1$ $string_2$) library procedure

- (string-ci<? $string_1$ $string_2$) library procedure
- (string-ci>? $string_1$ $string_2$) library procedure
- (string-ci<=? $string_1$ $string_2$) library procedure
- (string-ci>=? $string_1$ $string_2$) library procedure
- (substring *string start end*) library procedure

  The index in substring is assumed to fit in an int (usually at most $2^{31} - 1$).
- (string-append *string* ...) library procedure
- (string->list *string*) library procedure
- (list->string *list*) library procedure
- (string-copy *string*) library procedure
- (string-fill! *string char*) library procedure

  R5RS claims that string-fill! returns an unspecified value. In this implementation, it returns an object of void type.

- R5RS Section 6.3.6 Vectors

  Unsupported.

- R5RS Section 6.4 Control features
  - (procedure? *obj*) procedure
  - (apply *proc* arg_1 ... *args*) procedure
  - (map *proc* $list_1$ $list_2$ ...) library procedure

    R5RS claims that the dynamic order in which *proc* is applied to the elements of the lists is unspecified. In this implementation, the order is always from left to right.
  - (for-each *proc* $list_1$ $list_2$ ...) library procedure

    R5RS claims that the return value of for-each is unspecified. In this implementation, for-each returns an object of void type.
  - (force *promise*) library procedure

  Unsupported procedures:
  - (call-with-current-continuation *proc*) (unsupported) procedure
  - (values *obj* ...) (unsupported) procedure
  - (call-with-values *producer consumer*) (unsupported) procedure
  - (dynamic-wind $before$ *thunk* $after$) (unsupported) procedure

- R5RS Section 6.5 Eval

  Unsupporeted.

- R5RS Section 6.6.1 Ports

  Unsupported.

- R5RS Section 6.6.2 Input

  Unsupported.

- R5RS Section 6.6.3 Output

  - (display *obj*) library procedure
  - (newline) library procedure

  Unsuppoted features:

  - (write *obj*) (unsupported) library procedure
  - (write *obj port*) (unsupported) library procedure
  - (display *obj port*) (unsupported) library procedure
  - (newline *port*) (unsupported) library procedure
  - (write-char *char*) (unsupported) library procedure
  - (write-char *char port*) (unsupported) library procedure

- R5RS Section 6.6.4 System interface

  - (load $filename$) optional procedure

  Unsupported features:

  - (transcript-on $filename$) (unsupported) optional procedure
  - (transcript-off) (unsupported) optional procedure

4.4 Other implementation features

- Tail recursion.

5 Acknowledgement

I want to thank Hao Chen, Tongliang Liao and Maofan Ying, who has been working on the project, for the enlightening discussion. I also want to thank our TA Jingcheng Liu for the detailed references on the PPCA website and for answering our questions with great patience.

References

[1] R. Kelsey, W. Clinger, J. Rees (eds.), Revised5 Report on the Algorithmic Language Scheme, Higher-Order and Symbolic Computation, Vol. 11, No. 1, August, 1998.

[2] Peter Norvig. (How to Write a (Lisp) Interpreter (in Python)). From http://norvig.com/lispy.html.

[3] Peter Norvig. (An ((Even Better) Lisp) Interpreter (in Python)). From http://norvig.com/lispy2.html.

[4] Torbj¨orn Granlund and the GMP development team. (2013). The GNU Multiple Precision Arithmetic Library. From http://http://gmplib.org/.

[5] The MPFR team. (2013). The Multiple Precision Floating-Point Reliable Library. From http://www.mpfr.org/.

[6] Andreas Enge, Philippe Th´eveny, Paul Zimmermann. (2012). The GNU Multiple Precision Complex Library. From http://www.multiprecision.org/.