Specification of Scheme

Liu Guangyuan (Gary), An Jingyi

April 2021

Scheme is a statically scoped and properly tail-recursive dialect of the Lisp family of programming languages. Our implementation of Scheme is largely based on the *Revised*^{5} *Report on the Algorithmic Language Scheme* (R5RS), with certain omissions and a different macro system.

1 Syntax

The syntax of Scheme is defined using extended Backus–Naur form.

All spaces in the grammar are for legibility. Case is insignificant. <*empty*> stands for the empty string.

The following extensions to BNF are used to make the description more concise: <*thing*>* means zero or more occurrences one <*thing*>; and<*thing*>$^+$ means at least one <*thing*>.

A Scheme program is a *program*, as defined below:

```
<program>  ::=  <expression>+
<expression>  ::=  <atom>
                 | <list>
                 | <improper list>
                 | <quotation shorthands>
<atom>  ::=  number
           | string
           | #t | #f
           | identifier
<list>  ::=  (expression*)
<improper>  ::=  (expression+ .  expression)
<quotation shorthands>  | 'expression
                        | `expression
                        | ,expression
                        | ,@expression
```

Additionally, anything following a `;` and before the line ending will be treated as comments and ignored.

Notice that if we do not consider <*quotation shorthands*>[1], an expression can either be an atom, or zero or more expressions surrounded in parentheses, possible with a dot before the last expression. These expressions are commonly known as symbolic expressions, or S-expressions.

The above definition is simple, but it only focuses on the structure and does not capture enough information about the special syntactic constructs in the Scheme language. We provide an alternative and more rigorous definition that captures almost all the syntactic information as follows.

[1]We'll see later that they are in fact equivalent to some list representation

```
          <program>  ::=  <expression>+
       <expression>  ::=  <literal>
                       |  <variable>
                       |  <procedure call>
                       |  <macro use>
                       |  <lambda expression>
                       |  <conditional>
                       |  <definition>
                       |  <assignment>
                       |  <syntax definition>
                       |  <derived expression>
          <literal>  ::=  <self-evaluating> | <quotation> | <quasiquotation>
<self-evaluating>  ::=  <boolean> | <number> | <string>
          <boolean>  ::=  #t | #f
         <variable>  ::=  <identifier>
   <procedure call>  ::=  (<operator> <operand>*)
         <operator>  ::=  <expression>
          <operand>  ::=  <expression>
        <macro use>  ::=  (<keyword> <datum>*)
          <keyword>  ::=  <identifier>
<lambda expression>  ::=  (lambda <formals> <body>)
          <formals>  ::=  (<variable>*)
                       |  <variable>
                       |  (<variable>+ . <variable>)
             <body>  ::=  <expression>*
      <conditional>  ::=  (if <test> <consequent> <alternate>)
             <test>  ::=  <expression>
       <consequent>  ::=  <expression>
        <alternate>  ::=  <expression> | <empty>
```

```
<definition>  ::=  (define <variable> <expression>
                |  (define (<variable> <def formals>) <body>)
<def formals>  ::=  <variable>*
                |  <variable>* . <variable>
<assignment>  ::=  (set!  <variable> <expression>)
<syntax definition>  ::=  (defmacro <keyword> (<formals>) <body>)
<derived expression>  ::=  (cond <cond clause>+)
                |  (cond <cond clause>* (else <expression>*))
                |  (and <test>*)
                |  (or <test>*)
                |  (let (<binding spec>*) <body>)
                |  (let* (<binding spec>*) <body>)
                |  (lectrec (<binding spec>*) <body>)
                |  (begin <expression>*)
<cond clause>  ::=  ((<test>) <expression>*)
                |  ((<test>) => <expression>)
<binding spec>  ::=  (<variable> <expression>)
```

```
<quotation>          ::=  '<datum> | (quote <datum>)
<datum>              ::=  <simple datum>
                        | <compound datum>
<simple datum>       ::=  <boolean>
                        | <number>
                        | <string>
                        | <identifier>
<compound datum>     ::=  <list>
                        | <improper list>
<list>               ::=  (<datum>*)
<improper list>      ::=  (<datum>+ .  <datum>)
<quasiquotation>     ::=  `<qq template> | (quasiquote <qq template>)
<qq template>        ::=  <simple datum>
                        | <list qq template>
                        | <unquotation>
<list qq template>   ::=  (<qq template or splice>*)
                        | (<qq template or splice>+ .  <qq template>)
                        | '<qq template>
                        | <quasiquotation>
<qq template or splice> ::=  <qq template>
                        | ,@<qq template>
                        | (unquote-splicing <qq template>)
<unquotation>        ::=  ,<qq template>
                        | (unquote <qq template>)
```

2 Evaluation rules

Self-evaluating values

Numbers, booleans, and strings evaluate to themselves.

Identifiers

The definition for identifiers is as follows:

```
<identifier>::= <initial> <subsequent>*
              | <peculiar identifiers>
```

An <*initial*> is either an English letter (case-insensitive), or one of !, $, %, &, *, /, :, <, =, >, ?, ^, _ and @. An <*peculiar identifier*> is one of +, −, or ...

Identifiers have two use cases in Scheme:

- An identifier may be used as variable or as a syntactic keyword.
- When an identifier appears as literal or within a literal, it is used to denote a symbol.

Variables

An identifier may name a type of syntax, or it may name a location where a value can be stored. An identifier that names a type of syntax is called a `syntactic keyword` to and is said to be bound to that syntax.
An identifier that names a location is called a `variable` and is said to be bound that location.

An expression consisting of a variable is a variable reference. The value of a variable references is the value stored in the location to which the variable is bound.

We can use *define* to introduce a new location, bind an identifier to that location, and store a value in that location. We can also use *set!* to mutate

the value stored in the location to which the variable is bound. It is an error to reference an unbound variable. The following are some examples about variables:

```
x                                   ===> error - undefined variable x

(define x 10)
x                                   ===>10

(set! x 30)
x                                   ===> 30
```

Pairs

A pair is a primitive data structure in Scheme. It is a record structure with two fields called the `car` and `cdr` (for historical reasons). Pairs are created by the procedure cons.The `car` and `cdr` fields are accessed by the procedures *car* and `cdr`. The `car` and `cdr` fields are assigned by the procedures `set-car!` and `set-cdr!`.

Here are some examples about the pairs:

```
(define my-pair (cons 1 2))
 my-pair                            ===> (1 . 2)

(car my-pair)                       ===> 1

(cdr my-pair)                       ===> 2


(set-car! my-pair 10)
(set-cdr! my-pair 20)
 my-pair                            ===> (10 . 20)
```

Lists

A list is defined as either the empty list, or a pair whose tail is a list.

Here are some examples about the lists:

```
(define my-list (list 1 2 3))
 my-list                            ===> (1 2 3)
```

```
(cadr my-list)                      ===> 2

(caddr my-list)                     ===> 3


(map (lambda (x) (+ x 1)) my-list) ===> (2 3 4)
```

A pair that is not a list is an improper list.

```
(define my-improper-list (cons 1 (cons 2 3)))
my-improper-list                    ===> (1 2 . 3)

(cadr my-improper-list)             ===> 2

(cddr my-improper-list)                    ===> 3
```

Procedures

A procedure is like a function in other languages like C and Python.

A lambda expression evaluates to a procedure. Recall that Scheme has lexical scoping. The environment in effect when the `lambda` expression was evaluated is remembered as part of the procedure. When the procedure is later called with some actual argument, the environment in which the lambda expression was evaluated will be extended by binding the variables in the formal argument list to the fresh locations, the corresponding actual argument will be stored in these locations, and the expression in the body of the lambda expression will be evaluated sequentially in the extended environment.

Scheme provides a number of built-in procedures that the user can call without defining them. See later sections for a full list of such procedures. Procedures that are defined using `lambda` expressions are called compound procedures.

Ex1
Here are some examples of built-in procedures:

```
(+ 10 20)                   ===> 30
(- 20 5)                    ===> 15
(* 2 3)                     ===> 6
(/ 10 4)                    ===> 2.5
(+ 1 2 3 4 5)               ===> 15
```

```
(- 100 10 10)              ===> 80
```

Ex2
This example demonstrates how to define and call a compound procedure.

```
(define fn (lambda (x) (+ x 10)))
(fn 5)                                    ===> 15

; alternatively, apply the procedure without giving it a name
((lambda (x) (+ x 10)) 5)                 ===> 15
```

There is also a shorthand syntax to define procedures using `define`.

Ex3
This example represents shorthand procedure definition syntax.

```
(define (fn x)
    (+ x 10))
```

The procedures defined above take in a fixed number of arguments equal to the number of parameters. Calling the procedures with the wrong number of arguments results in an error.

Ex4

```
(fn 10 10)      ===> error - fn expected exactly 1 argument,
                     but encountered 2
```

We can also define procedures that take in variable number of arguments. Such procedures can specify a number of compulsory arguments. Any additional arguments supplied will be lumped into a list and assigned to the name of the last parameter.

Ex5
This example demonstrates how to define and call a procedure with var-args.

```
(define (fn x y . rest-args)
    (display x)
    (display y)
    (display rest-args))
(fn 1 2 3 4)                    ===> displays 1 2 (3 4)
```

Besides applying the procedure directly, we can use the `apply` built-in procedure, which is useful when the arguments are in a list and need to be 'spread'.

Ex6

```
(apply + (list (1 2 3)))                ===> 6
```

Conditionals

Conditional enable conditional evaluation. As a recap, the syntax for conditionals is as follows:

```
(if <test> <consequent> <alternate>)
```

or

```
(if <test> <consequent>)
```

A conditional expression is evaluated as follows: first, `<test>` is evaluated. If it yields a true value, then `<consequent>` is evaluated and its value is returned. Otherwise `<alternate>` is evaluated and its value is returned.

A value is false if it is `#f`. Otherwise, the value is true.

If `<test>` evaluates to false and `<consequent>` is missing, then the return value of the conditional expression is unspecified.

Quotation

The syntax for quotation is:

```
(quote <datum>)
```

which can be abbreviated as:

```
'<datum>
```

The two notation are equivalent is all aspects.

Quotation is used to introduce literals in Scheme.

Quotation is evaluated as follows:

- Quotation of a self-evaluating value returns the value itself.

- Quotation of an identifier returns a symbol.
- Quotation of a pair returns a pair whole `car` and `cdr` are the result of quoting the `car` and `cdr` of the original pair.

The followings gives some examples:

Ex1

```
(quote 1)                                ===> 1
(quote "string")                         ===> "string"
(quote symbol)                           ===> symbol
(quote (+ 1 2))                          ===> (+ 1 2)
(quote (this is a list))                 ===> (this is a list)
```

The result of a quotation has no semantic difference from an equivalent value produced through other means. Therefore, we can manipulate quoted values exactly like all other values.

Ex2

```
(define my-list
    (quote (1 2 (list 3 4))))
my-list                                  ===> (1 2 (list 3 4))
(caddr my-list)                          ===> (list 3 4)
```

Quasiquotation

Syntax:

```
(quasiquote <qq template>)
```

which can be abbreviated as:

```
`<qq template>
```

The two notation are equivalent is all aspects.

Backquote or quasiquote expressions are useful for constructing a list when most but not all of the desired structure is known in advance. If no commas appear within the `<qq template>`, the result of evaluating `` `<qq template> `` is equivalent to the result of evaluating `'<qq template>`. If a comma appears within the <qq template>, however, the expression following the comma is

evaluated (unquoted) and its result is inserted into the list instead of the comma and the expression. If a comma appears followed immediately by an at-sign (@), then the following expression must evaluate to a list; the opening and closing parentheses of the list are then stripped away and the elements of the list are inserted in place of the comma at-sign expression sequence. A comma at-sign should only appear within a list `<qq template>`.

The following are some examples:
Ex1

```
(quasiquote (this is a list))          ===> (this is a list)
```

Ex2

```
(quasiquote (1 2 (unquote (list 3 4))))     ===> (1 2 (3 4))
```

Ex3

```
(quasiquote (1 2 (unquote-splicing (list 3 4))))
                     ===> (1 2 3 4)
```

`,<expression>` is identical to `(unquote <expression>)`, and `,@<expression>` is identical to `(unquote-splicing <expression>)` in all aspects.

Quasiquote forms may be nested. Substitutions are made only for unquoted components appearing at the same nesting level as the outermost backquote. The nesting level increases by one inside each successive quasiquotation, and decreases by one inside each unquotation.

Ex4

```
`(a `(b ,(+ 1 2) ,(foo ,(+ 1 3) d) e) f)
                                ===> (a `(b ,(+ 1 2) ,(foo 4 d) e) f)
```

Macros

Scheme has a special property: homoiconicity. As we have seen, Scheme syntax consists of S-expressions, and S-expressions also happen to be the a native data structure supported by Scheme. More formally, the syntactic domain of scheme is a subset of its semantic domain. As a result, we can manipulate syntax as data in Scheme. For example, the programmer can define new expressions types by converting new syntax into syntax that Scheme natively understands. This is done using a mechanism called Macros.

A macro definition binds an identifier to a macro body, which transforms input syntaxes into output syntaxes.

For example, we can define a new syntax called `swap!` that swaps the value of two variable. Note that this cannot be done with a procedure.

```
(defmacro swap! (lhs rhs)
    `(let ((temp ,lhs))
        (set! ,lhs ,rhs)
        (set! ,rhs temp)))

(define x 1)
(define y 2)
(swap! x y)

x    ===> 2
y    ===> 1
```

Here, the macro we defined takes in two arguments `lhs` and `rhs`. When we use this macro, these parameters will be bound to the syntax that we supply at the use site. In this case, `lhs` is bound to the symbol `x` and `rhs` is bound to the symbol `y`.

The macro body can manipulate the values bound to these names. In this case, we use quasiquotation to produce a list that contains the values bound to `lhs` and `rhs`. We say that this macro has expanded into this list.

We can use the `macroexpand` built-in procedure to view what a macro use expands into.

```
(macroexpand '(swap! x y))
                    ===> (let ((temp x)) (set! x y) (set! y temp))
```

Note that this list is valid Scheme syntax - it introduces a temporary variable and uses it to swap the values between the variables whose names are stored in `lhs` and `rhs`.

The final step in a macro use is that the expanded value will be evaluated. In this case, the expanded `let` form it does exactly what we want - to swap the value of two variables.

As a result, we have introduced a new syntax `swap!` that swaps the value of two variales. We achieve this by defining a macro which breaks down our new syntax into primitive syntax that Scheme already understands. This allows us to extend the language by introducing new syntax that suits our needs.

As a more complex example, let's look at how the `let` special form is defined using a macro is our standard library. The definition is as follows:

```
(defmacro let (bindings . bodies)
  `((lambda (,@(map car bindings)) ,@bodies) ,@(map cadr bindings)))

(macroexpand '(let ((x 1) (y 2)) (+ x y)))
                          ===> ((lambda (x y) (+ x y)) 1 2)
```

The macro takes in a compulsory parameter `bindings`, and a variable number of expressions as `bodies`. Within the macro body, it decomposes the `bindings` and `bodies` into an immediately applied lambda expression, whose parameters are the names in `bindings` (represented as the first element of each sub-list in `bindings`). We supply the values in `bindings` (represented as the second element of each sub-list in `bindings`) to this procedure. The `bodies` constitute the procedure's body, which will be evaluated in an environment where all bindings are in effect.

Note that we used `map` to manipulate `bindings`. In fact, we have the full power of Scheme within the macro body. Since Scheme understands its full syntax, the macro body can do any arbitrary computation on the input syntax. This would not have been possible if not for homoiconicity.

In some other macro systems, (e.g., C's macro system), syntax is represented as plain strings. In such cases, the macro body can only do limited operations like concatenation and template substitution on the syntax (unless we encode a parser into the macro expansion system).

Macro hygiene

Macros can introduce unwanted name captures. Recall the example about `swap!`. What if one of the variables that we want to swap has the name `temp`?

```
(macroexpand '(swap! x temp))
               ===> (let ((temp x)) (set! x temp) (set! temp temp))
```

This does not look right. In fact, it will not swap the two variables. When we evaluate the `let` in the expanded syntax, we introduce a new environment where `temp` is bound to the value of `x`. When we try to `set!` the value of `temp`, it sets the value in this inner environment, instead of the environment where the macro is used!

This macro fails because the expanded syntax introduced a name conflict between a variable defined by the expanded syntax and user code. Such

macros are called unhygienic macros.

As a workaround, there exists a built-in procedure `gensym` which returns a symbol that is guaranteed to not to be used anywhere else and will not be used anywhere else. With `gensym`, we can give the temporary variable a unique name so that it will never conflict with names in user code.

Let's redefine `swap!` using `gensym`:

```
(defmacro swap! (lhs rhs)
  (let ((temp (gensym)))
    `(let ((,temp ,lhs))
       (set! ,lhs ,rhs)
       (set! ,rhs ,temp))))


(define x 1)
(define temp 2)

(macroexpand '(swap! x temp))
            ===> (let (($:0 x)) (set! x temp) (set! temp $:0))

(swap! x temp)
x    ===> 2
temp ===> 1
```

With `gensym`, we can write hygienic macros under an unhygienic macro system by introducing unique names.

Derived expression types

Derived expression types are expressions that can be defined using primitive expressions and macros. As a result, the Scheme evaluator does not need to understand these expressions natively. They are defined using macros as part of our standard library. The definitions can be found at `https://gist.github.com/gary-lgy/51e03d4d387ed2eec322326769bfd153`.

begin

The following is the syntax of `begin`:

```
(begin <expression>+)
```

The <expression>s are evaluated sequentially from left to right, and the value of the last <expression> is returned.

cond

The following is the syntax of cond:

```
(cond <clause> <clause> ...)
```

Each <clause> should be form (<test> <expression> ...). And <test> is any expression. When a <test> evaluates to a true value, then the remaining <expression>s in its <clause> are evaluated in order, and the results of the last <expression> in the clause is(are) returned as the result(s) of the entire cond expression. If the selected <clause> contains only the <test> and no <expression>s, then the value of the <test> is returned as the result. If all <test>s evaluate to false values, and there is no else clause, then the result of the conditional expression is unspecified; if there is an else clause, then its <expression>s are evaluated, and the value(s) of the last one is(are) returned.

The following gives some examples about cond:

```
(cond ((> 3 2) 'greater)
      ((< 3 2) 'less))                              ===>  greater

(cond ((> 3 3) 'greater)
      ((< 3 3) 'less)
      (else 'equal))                                ===>  equal
```

and

The following is the syntax of and:

```
(and <test> ...)
```

The <test> expressions are evaluated from left to right, and the value of the first expression that evaluates to a false value is returned. Any remaining expressions are not evaluated. If all the expressions evaluate to true values, the value of the last expression is returned. If there are no expressions then #t is returned.

or

The following is the syntax of `or`:

```
(or <test> ...)
```

The `<test>` expressions are evaluated from left to right, and the value of the first expression that evaluates to a true value is returned. Any remaining expressions are not evaluated. If all expressions evaluate to false values, the value of the last expression is returned. If there are no expressions then #f is returned.

Binding constructs

The three bindings constructs *let*, *let**, and *letrec* give Scheme a block procedure. The syntax of three constructs is identical, but they differ in the regions they establish for their variable bindings.

- `(let <bindings> <body>)`
- `(let* <bindings> <body>)`
- `(letrec <bindings> <body>)`

  where `<body>` should be a sequence of one or more expressions.

For `let` expression, the initial values are computed before any of the variables become bound. In a `let*`expression, the bindings and evaluations are performed sequentially; while in a `letrec` expression, all the bindings are in effect while their initial values are being computed, thus allowing mutually recursive definitions. To be specific, we give an example about `letrec` here:

```
(letrec ((even?
          (lambda (n)
            (if (zero? n)
                #t
                (odd? (- n 1)))))
         (odd?
          (lambda (n)
```

```
          (if (zero? n)
              #f
              (even? (- n 1))))))
(even? 88))                                  ===>  #t
```

3 Tail call optimization

Scheme is "properly tail recursive", meaning that procedure calls from certain contexts do not consume extra resources and can an unbounded number of tail calls is be supported.

Tail calls are procedure calls that occur in tail contexts. Tail context is defined inductively as follows:

- The last expression within the body of a lambda expression, shown as `<tail expression>` below, occurs in a tail context.

  ```
  (lambda <formals>
          <expression>* <tail expression>)}
  ```

- If one of the following expressions is in a tail context, then the subexpressions shown as `<tail expression>` are in a tail context. Here we replace some occurrences of `<expression>` with `<tail expression>`. Only those rules that contain tail contexts are shown here.

  ```
  (if <expression> <tail expression> <tail expression>)
  (if <expression> <tail expression>)

  (cond <cond clause>+)
  (cond <cond clause>* (else <tail sequence>))

  (and <expression>* <tail expression>)
  (or <expression>* <tail expression>)

  (let (<binding spec>*) <tail body>)
  (let* (<binding spec>*) <tail body>)
  (letrec (<binding spec>*) <tail body>)

  (begin <tail sequence>)
  ```

where

```
<cond clause>  ::=  (<test> <tail sequence>)
  <tail body>  ::=  <tail sequence>
<tail sequence>::=  <expression>* <tail expression>
```

- If a `cond` expression is in a tail context, and has a clause of the form `(<expression1> => <expression2>)` then the (implied) call to the procedure that results from the evaluation of `<expression2>` is in a tail context. `<expression2>` itself is not in a tail context.

4 Deviations from R5RS

- The full numeric tower is not supported. All numbers behave similarly to numbers in JavaScript.
- Strings are immutable.
- No vector or character data types.
- No hygienic macro system. Only an unhygienic macro system is provided.
- Some standard library procedures are simplified.

5 Denotational Semantics of Scheme

In this section, we follow the notation used in CS4215 lectures, unless otherwise stated.

We define the syntactic domain of Scheme is follows.

Domain name	Definition	Explanation
Syn	**Num** + **Bool** + **Str** + **Symbol**+ **Pair** + **EmptyList**	syntax

We define the semantic domains of Scheme as follows.

Domain name	Definition	Explanation
EV	$\mathbf{Num} + \mathbf{Bool} + \mathbf{Str} + \mathbf{Symbol} +$ $\mathbf{Pair} + \mathbf{EmptyList} + \mathbf{Proc} + \mathbf{Macro}$	expressible values
SV	$\mathbf{Num} + \mathbf{Bool} + \mathbf{Str} + \mathbf{Symbol} +$ $\mathbf{Pair} + \mathbf{EmptyList} + \mathbf{Proc} + \mathbf{Macro}$	storable values
DV	$\mathbf{Loc}$	denotable values
Proc	$\mathbf{DV} * \cdots * \mathbf{DV} * \mathbf{Store(EV, Store)}$	procedure values
Macro	$\mathbf{Syn} * \cdots * \mathbf{Syn} * \mathbf{Store(Syn, Store)}$	macro values
Store	$\mathbf{Loc} \rightsquigarrow \mathbf{SV}$	stores
Env	$\mathbf{Id} \rightsquigarrow \mathbf{DV}$	environments

We present an outline of the denotational semantics of Scheme.

nil denotes unspecified return value.

Variable definition

$$\frac{\sum' [l_1 \leftarrow v_1] \, | \Delta[x \leftarrow l_1] \Vdash E_2 \rightarrow \left(v, \sum''\right)}{\sum | \Delta \Vdash (define \quad x \quad E_1) \quad E_2 \rightarrow \left(v, \sum''\right)}$$

if $\sum | \Delta \Vdash E_1 \rightarrowtail (v_1, \sum')$,
and l_1 is a new location.

Variable access

$$\frac{}{\sum | \Delta \Vdash x \rightarrowtail (\sum(\Delta(x)), \sum)}$$

Variable mutation

$$\frac{\sum | \Delta \Vdash E \rightarrowtail (v, \sum')}{\sum | \Delta \Vdash (set! \quad x \quad E) \rightarrowtail (nil, \sum'[\Delta(x) \leftarrow v])}$$

Lambda expression

$$\frac{}{\sum | \Delta \Vdash (lambda \quad (x) \quad E) \rightarrowtail (f, \sum)}$$

where $f(l, \sum') = (v', \sum'')$,
where $\sum' | \Delta[x \leftarrow l] \Vdash E \rightarrowtail (v', \sum'')$

Procedure application

$$\frac{\sum | \Delta \Vdash E_1 \rightarrowtail (f, \sum') \qquad \sum' | \Delta \Vdash E_2 \rightarrowtail (v_2, \sum'')}{\sum | \Delta \Vdash (E1 \quad E2) \rightarrowtail f(l, \sum''[l \leftarrow v_2])}$$

where l is a new location in $\sum''$
and $f \in$ `Proc`

Macro definition

$$\frac{\sum[l_1 \leftarrow f] | \Delta[m \leftarrow l_1]| \Vdash E_2 \rightarrowtail (v, \sum')}{\sum | \Delta \Vdash (defmacro \quad m \quad (x) \quad E) \quad E_2 \rightarrowtail \left(v, \sum'\right)}$$

where $f(l, \sum') = (v', \sum'')$,
where $\sum' | \Delta[x \leftarrow l] \Vdash E \rightarrowtail (v', \sum'')$
and l_1 is a new location.

Macro use

$$\frac{\sum | \Delta \Vdash E_1 \rightarrowtail (f, \sum') \quad \sum' | \Delta \Vdash f(l, \sum'[l \leftarrow E_2]) \rightarrowtail (S, \sum'') \quad \sum'' | \Delta \Vdash S \rightarrowtail (v, \sum''')}{\sum | \Delta \Vdash (E1 \quad E2) \rightarrowtail (v, \sum''')}$$

where l is a new location in $\sum'$
and $f \in$ `Macro`

The definitions for the other constructs are straight-forward.

6 Built-in Procedures

To be specific, we summarize all built-in procedures here.
These procedures are accessing the basic calculating function:

- `(+ z ...)`

- `(- z ...)`
- `(* z ...)`
- `(/ z ...)`

These procedures implement number-theoretic (integer) division. n2 should be non-zero. All three procedures return integers. If n1/n2 is an integer:

- `(quotient` n_1 n_2`)` `===>` n_1/n_2
- `(remainder` n_1 n_2`)` `===>` 0
- `(modulo` n_1 n_2`)` `===>` 0

If n1/n2 is not an integer:

- `(quotient` n_1 n_2`)` `===>` n_q
- `(remainder` n_1 n_2`)` `===>` n_r
- `(modulo` n_1 n_2`)` `===>` n_m

where n_q is n_1/n_2 rounded towards zero, $0 < |n_r| < |n_2|$, $0 < |n_m| < |n_2|$, n_r and n_m differ from n_1 by a multiple of n_2, n_r has the same sign as n_1, and n_m has the same sign as n_2.

- `(=` z_1 z_2`)`
- `(<` x_1 x_2`)`
- `(>` x_1 x_2`)`
- `(<=` x_1 x_2`)`
- `(>=` x_1 x_2`)`

These procedures return #t if their arguments are (respectively): equal, monotonically increasing, monotonically decreasing, monotonically nondecreasing, or monotonically nonincreasing.

- `(cons` obj_1 obj_2`)`

Returns a newly allocated pair whose car is obj1 and whose cdr is obj2.

- (car *pair*)

Returns the contents of the car field of pair.

- (cdr *pair*)

Returns the contents of the cdr field of pair.

- (set-car! *pair obj*)

Stores obj in the car field of pair. The value returned by set-car! is unspecified.

- (set-cdr! *pair obj*)

Stores obj in the cdr field of pair. The value returned by set-cdr! is unspecified.

- (list *obj* ...)

Returns a newly allocated list of its arguments.

- (number? *obj*)

Returns #t if obj is a number, otherwise returns #f.

- (boolean? *obj*)

Returns #t if obj is a boolean, otherwise returns #f.

- (string? *obj*)

Returns #t if obj is a string, otherwise returns #f.

- (symbol? *obj*)

Returns #t if obj is a symbol, otherwise returns #f.

- (procedure? *obj*)

Returns #t if obj is a procedure, otherwise returns #f.

- (pair? *obj*)

Returns #t if obj is a pair, otherwise returns #f.

- (null? *obj*)

Returns #t if obj is the empty list, otherwise returns #f.

- (eq? obj_1 obj_2)
- (eqv? obj_1 obj_2)
- (equal? obj_1 obj_2)

The three equivalence predicates above are the computational analogue of a mathematical equivalence relation. The eqv? procedure defines a useful equivalence relation on objects. Briefly, it returns t if obj1 and obj2 should normally be regarded as the same object. eq?'s behavior on numbers and characters is implementation-dependent, but it will always return either true or false, and will return true only when eqv? would also return true. Equal? recursively compares the contents of pairs, and strings, applying eqv? on other objects such as numbers and symbols. A rule of thumb is that objects are generally equal? if they print the same.

- (apply *proc* arg_1 ...arg_s)

Calls proc with the elements of the list (append (list arg1 ...) args) as the actual arguments.Proc must be a procedure and args must be a list.

- (error *obj* ...)

Aborts further evaluation and print the supplied objects.

- (display *obj*)

Displays the object.

7 Special Syntax Definitions

- `(begin <expression> <expression> ...)`: Evaluates the expression(s) from left to right, and the values of the last expression is(are) returned.
- `(cond <clause> <clause> ...)`: Each <clause> should be form (<test> <expression> ...). And <test> is any expression. When a <test> evaluates to a true value, then the remaining <expression>s in its <clause> are evaluated in order, and the results of the last <expression> in the clause is(are) returned as the result(s) of the entire `cond` expression. If the selected `<clause>` contains only the <test> and no `<expression>`s, then the value of the `<test>` is returned as the result. If all `<test>`s evaluate to false values, and there is no else clause, then the result of the conditional expression is unspecified; if there is an else clause, then its `<expression>`s are evaluated, and the value(s) of the last one is(are) returned.
- `(and <test> ...)`:The `<test>` expressions are evaluated from left to right, and the value of the first expression that evaluates to a false value is returned. Any remaining expressions are not evaluated. If all the expressions evaluate to true values, the value of the last expression is returned. If there are no expressions then #t is returned.
- `(or <test> ...)`:The `<test>` expressions are evaluated from left to right, and the value of the first expression that evaluates to a true value is returned. Any remaining expressions are not evaluated. If all expressions evaluate to false values, the value of the last expression is returned. If there are no expressions then #f is returned.

8 Libraries

The following libraries are always available in this language.

The definitions can be found at
`https://gist.github.com/gary-lgy/519fd61f3df7d89bf4e84cbc8486c5f6`.

Pair Accessors

The following pair accessing functions are supported and there procedures are compositions of car and cdr:

- (caar *pair*)
- (cadr *pair*)
- (cadr *pair*)
- (cddr *pair*)
- (caaar *pair*)
- (caadr *pair*)
- (cadar *pair*)
- (caddr *pair*)
- (cdaar *pair*)
- (cdadr *pair*)
- (cddar *pair*)
- (cdddr *pair*)
- (caaaar *pair*)
- (caadar *pair*)
- (caaddr *pair*)
- (caaddr *pair*)
- (cadddr *pair*)
- (cdaaar *pair*)
- (cdaddr *pair*)
- (cddaar *pair*)
- (cdddadr *pair*)
- (cddddar *pair*)
- (cddddr *pair*)

Logical Operator

- (not *obj*): Returns #t if obj is false, and returns #f otherwise.

Numerical Operator

- `(zero? z)`: Returns #t if z is 0, and returns #f otherwise.

List Helpers

- `(list? obj)`: Returns #t if obj is a list, otherwise returns #f.
- `(length list)`: Returns length of list.
- `(append list1 list2)`: Returns a list consisting of the elements of the first list followed by the elements of the other lists.
- `(reverse list)`: Returns a newly allocated list consisting of the elements of list in reverse order.
- `(map proc list)`: Applies *proc* element-wise to the elements of the list and returns a list of the returns, in order.
- `(for-each proc list)`: Calls `proc` on the elements of the `list` in order from the firsr element(s) to the last, and the value returned is unspecified.