A brief description of Slip

Theo D'Hondt
Thursday 2 January 2014

Rationale:

Slip is a variant of the programming language Scheme. It is intended to return to the original spirit of Scheme: to provide a simple, powerful "all things reasonable are possible" kind of language targeted at teaching and fundamental research. In this, Slip does not follow the more recent evolution to transform Scheme into a language for engineering and not so much for designing.

Slip is an anagram for Lisp - the second language to carry that name; the first Slip was an (obsolete) extension of Fortran dating back to the 60's. Several acronym-like definitions could be conceived for Slip; we prefer to think of the simplicity and slenderness associated with some definitions of the noun slip.

Slip is closer to R5RS than to any of the other releases of Scheme. In fact, Slip is intended to be SICP compliant, and endeavours to encapsulate the spirit of Abelson and Sussman's textbook. Some simple statistics provide support for Slip's design. Scanning the source code published on the SICP website, we find that:

- the let* form appears exactly twice - in exercise 4.7;
- the letrec form appears exactly twice - in exercise 4.20/21;
- the pattern "named let" does not occur at all;
- the define form is used significantly more frequently than the let form - excluding chapter 1 (where the ratio is 129 to 7), the ratio varies between 3 and 7 to 1;
- most define forms are used on a global level - however, depending on the chapter, up to 1 out of 3 define forms are used locally;
- the do special form is not used; neither is the case form;
- it would really take a lot more space to report on of how little (or not at all) standard library functions are used;
- streams are extensively used, but are absent from any of the Scheme standards.

There definitely is an issue with the define form in Scheme; this has been pointed out in ample detail elsewhere. However, if SICP code can be used as a guideline, the define is the variable binding of choice, if only for its convenience. The second-rate significance that Scheme reserves for the define form does not do it justice. Furthermore the recent introduction of a letrec* form (proposed for R7RS) reveals the shortcomings of mapping local define forms to block structure. Hence the decision in Slip to return define to first-class status – similar to the role of the define form in the meta-circular scheme interpreter of chapter 4 of SICP.

A result of this decision is that Slip disallows forward references to as yet undefined variables – except for variables at the global read-eval-print level[1]. It also implies that certain clauses in special forms are evaluated in an extended nested scope. For instance: variables introduced inside a clause of an if form will not be accessible outside the clause. Also, a define local to a function will not we allowed to shadow a parameter of that function; this is not the case for Scheme. Finally, variables will be immediately available within their scope as bound to their value

A more subtle difference between Slip and Scheme relates to the order of operand evaluation during function application. Starting with the initial design of Scheme by Steele and Sussman in 1978, this order has been defined to be arbitrary. Reasons given for this design can hardly be considered to be completely sound since they are generally concerned with implementation issues and not with language consistency. Slip follows Lisp and evaluates operands strictly from left to right.

Slip essential syntax:

Below is the essential syntax for Slip. It generally coincides with Slip's essential syntax:

⟨expression⟩ → ⟨begin⟩ | ⟨define⟩ | ⟨assignment⟩ | ⟨if⟩ |⟨application⟩ |
⟨lambda⟩ | ⟨quote⟩ | ⟨variable⟩ | ⟨literal⟩

⟨begin⟩ → (begin ⟨expression⟩+)

⟨define⟩ → (define ⟨variable⟩ ⟨expression⟩) |
(define ⟨pattern⟩ ⟨expression⟩+)

⟨assignment⟩ → (set! ⟨variable⟩ ⟨expression⟩)

⟨if⟩ → (if ⟨expression⟩ ⟨expression⟩ ⟨expression⟩) |
(if ⟨expression⟩ ⟨expression⟩)

⟨application⟩ → (⟨expression⟩+)

⟨lambda⟩ → (lambda () ⟨expression⟩+) |
(lambda ⟨variable⟩ ⟨expression⟩+) |
(lambda (⟨pattern⟩) ⟨expression⟩+)

⟨quote⟩ → '⟨expression⟩ | (quote ⟨expression⟩)

⟨variable⟩ → [symbol]

⟨pattern⟩ → (⟨variable⟩+) | (⟨variable⟩+ . ⟨variable⟩)

⟨literal⟩ → [number] | [character] | [string] | #t | #f | ()

[1] a compromise to conform to SICP usage

The derived forms for and, delay, cond, let and or follow R5RS' syntax. Quasiquoting is close to R5RS, with the difference that only lists and not vectors can be quasiquoted. For cons-stream, please consult chapter 3 of SICP.

Slip keywords:

Below are Slip's keywords[1]:

```
and                  lambda
begin                let⁴
cons-stream²         or
cond                 quasiquote
define³              quote
delay                set!
else                 unquote
if                   unquote-splicing
```

The following keywords from R5RS are (intentionally) not available in Slip:

```
=>                   let*
case                 letrec
define-syntax        letrec-syntax
do                   syntax-rules
let-syntax
```

Slip domains:

Slip adopts all domains from R5RS, as illustrated by Slip's type predicates:

```
boolean?             port?
char?                procedure?
integer?             real?
null?                string?
pair?                symbol?
vector?
```

Note however that Slip does not implement R5RS' numerical tower: only integers and reals are supported.

---

[1] these are true keywords - they may not be reused for identifiers, not even in a nested scope
[2] consult chapter 3 of SICP
[3] with first-class semantics
[4] Slip's **`let`** is identical to R5RS' **`let*`**

Slip primitives:

Below are listed all primitive functions that Slip reuses from R5RS:

```
-
*
/
+
<
<=
=
>
>=
abs
acos
append[1]
apply[2]
asin
assoc
assq
atan[3]
boolean?
caaaar
caaadr
caaar
caadar
caaddr
caadr
caar
cadaar
cadadr
cadar
caddar
cadddr
caddr
cadr
call-with-current-continuation
car
cdaaar
cdaadr
cdaar
cdadar
cdaddr
cdadr
cdar
cddaar
cddadr
cddar
cdddar
cddddr
cdddr
cddr
cdr
char->integer
char?
char<?
char<=?
char=?
char>?
char>=?
close-input-port
close-output-port
cons
cos
display
eof-object?
eq?
equal?
eval[4]
even?
exp
expt
for-each
force
input-port?
integer?
length
list
list->vector
load
log
make-string
make-vector
map
max
member
memq
min
```

[1] exactly two list arguments
[2] single argument list
[3] single argument form
[4] no environment parameter

```
modulo
negative?
newline
not
null?
number->string
number?
odd?
open-input-file
open-output-file
output-port?
pair?
positive?
procedure?
quotient
read
read-char
real?
remainder
reverse
set-car!
set-cdr!
sin
sqrt
string->number
string->symbol
string-append
string-length
string-ref
string-set!
string?
string<?
string<=?
string=?
string>?
string>=?
substring
symbol->string
symbol?
tan
vector
vector->list
vector-length
vector-ref
vector-set!
vector?
write
write-char
zero?
```

Except where indicated, these are identical to their Scheme counterparts. In addition to these, Slip implements SICP's primitive stream functions:

```
stream-null?
stream-car
stream-cdr
```

Additionally, Slip provides the following primitive variables:

`circularity-level`	set to one before each evaluation
`the-empty-stream`	set to the null stream before each evaluation
`false`	set to #f before each evaluation
`true`	set to #t before each evaluation

and primitive functions:

`clock`	time since launch in milliseconds
`collect`	force a garbage collection
`error`	abort evaluation with error message
`pretty`	pretty-print the argument expression
`random`	generate a random integer

The following R5RS primitive functions are (intentionally) ***not*** provided:

```
angle
assv
call-with-input-file
call-with-output-file
```

```
ceiling                       list-ref
char-alphabetic?              list-tail
char-ci<?                     magnitude
char-ci<=?                    make-polar
char-ci=?                     make-rectangular
char-ci>?                     memv
char-ci>=?                    numerator
char-downcase                 peek-char
char-lower-case?              rational?
char-numeric?                 rationalize
char-ready?                   real-part
char-upcase                   round
char-upper-case?              scheme-report-environment
char-whitespace?              string
complex?                      string->list
current-input-port            string-ci<?
current-output-port           string-ci<=?
denominator                   string-ci=?
eqv?                          string-ci>?
exact->inexact                string-ci>=?
exact?                        string-copy
floor                         string-fill!
gcd                           transcript-off
imag-part                     transcript-on
inexact->exact                truncate
inexact?                      vector-fill!
integer->char                 with-input-from-file
list->string                  with-output-to-file
```

Slip semantics:

In this brief description of Slip, semantics are introduced using a meta-circular evaluator. This evaluator is executable by any true Slip interpreter – including itself. In order to manage towers of Slip interpreters, the primitive variable **circularity-level** is used to indicate reflective depth:

```
(begin
  (define old-circularity-level circularity-level)
  (define circularity-level (+ old-circularity-level 1))
  (define meta-level-eval eval)

  (define (loop output environment)
    (define rollback environment)

    (define (eval expression)

      (define (abort message qualifier)
        (display message)
        (loop qualifier rollback))
```

```
(define (load string)
  (eval (read (open-input-file string))))

(define (bind-variable variable value)
  (define binding (cons variable value))
  (set! environment (cons binding environment)))

(define (bind-parameters parameters arguments)
  (if (symbol? parameters)
    (bind-variable parameters arguments)
    (if (pair? parameters)
      (begin
        (define variable (car parameters))
        (define value (car arguments))
        (bind-variable variable value)
        (bind-parameters (cdr parameters) (cdr arguments))))))

(define (thunkify expression)
  (define frozen-environment environment)
  (define value (eval expression))
  (set! environment frozen-environment)
  value)

(define (eval-sequence expressions)
  (define head (car expressions))
  (define tail (cdr expressions))
  (define value (eval head))
  (if (null? tail)
    value
    (eval-sequence tail)))

(define (close parameters expressions)
  (define lexical-environment environment)
  (define (closure . arguments)
    (define dynamic-environment environment)
    (set! environment lexical-environment)
    (bind-parameters parameters arguments)
    (define value (eval-sequence expressions))
    (set! environment dynamic-environment)
    value)
  closure)

(define (eval-application operator)
  (lambda operands
    (apply (eval operator) (map eval operands))))

(define (eval-begin . expressions)
  (eval-sequence expressions))

(define (eval-define pattern . expressions)
```

```
  (if (symbol? pattern)
    (begin
      (define binding (cons pattern ()))
      (set! environment (cons binding environment))
      (define value (eval (car expressions)))
      (set-cdr! binding value)
      value)
    (begin
      (define binding (cons (car pattern) ()))
      (set! environment (cons binding environment))
      (define closure (close (cdr pattern) expressions))
      (set-cdr! binding closure)
      closure)))

(define (eval-if predicate consequent . alternate)
  (if (eval predicate)
    (thunkify consequent)
    (if (null? alternate)
      ()
      (thunkify (car alternate)))))

(define (eval-lambda parameters . expressions)
  (close parameters expressions))

(define (eval-quote expression)
  expression)

(define (eval-set! variable expression)
  (define value (eval expression))
  (define binding (assoc variable environment))
  (if (pair? binding)
     (set-cdr! binding value)
     (abort "inaccessible variable: " variable)))

(define (eval-variable variable)
  (define binding (assoc variable environment))
  (if (pair? binding)
    (cdr binding)
    (meta-level-eval variable)))

(if (symbol? expression)
  (eval-variable expression)
  (if (pair? expression)
    (begin
      (define operator (car expression))
      (define operands (cdr expression))
      (apply
        (if (equal? operator 'begin) eval-begin
          (if (equal? operator 'define) eval-define
            (if (equal? operator 'if) eval-if
              (if (equal? operator 'lambda) eval-lambda
```

```
                        (if (equal? operator 'quote) eval-quote
                          (if (equal? operator 'set!) eval-set!
                            (eval-application operator))))))) operands))
            expression)))

      (display output)
      (newline)
      (display "level ")
      (display circularity-level)
      (display ">")
      (loop (eval (read)) environment))

    (loop "Meta-Circular Slip" ()))
```

All values (including procedures) — and all primitives (except for **`eval`** and **`load`**) instantiated by this meta-circular evaluator are reified meta-level values. Environments are association lists[1] and instead of using an argument to the evaluation functions to hold the environment, it is defined outside **`eval`**.

This meta-circular evaluator implements essential syntax only. Implementations for and, delay, cond, let and or should follow R5RS' approach.

Note that this evaluator can almost trivially be translated into Scheme by substituting **`let*`** forms for **`begin`**/**`define`** combinations.

Slip mechanics:

A Slip interpreter is supported by garbage collection and is properly tail recursive. Slip does not (as yet) provide a macro facility. Whenever added, it will be closer to Lisp than to Scheme.

[1] disallowing global forward referencing