Prototyping a Language Design

Robert Keller
September 2010

Aspects of Language Design

- **Syntax**: How the language **looks** to the user.
- **Semantics**: What the language **means**; how it language behaves when executed.
- **Pragmatics**: **Practical aspects** of working with the language in a development environment.

What’s Most Imporant?

- Semantics, by far
- There can be many different syntaxes for the same semantics.

Syntaxes: “Hello World!” in Various Languages They all mean basically the same thing.

```
; Hello World in Racket

(display "Hello, world!")
(newline)
```

```
// Hello World in Java

class HelloWorld
  {
  static public void main( String args[] )
   {
    System.out.println( "Hello World!" );
   }
}
```

```
// Hello World in C++

#include <iostream>

main()
{
    std::cout << "Hello World!" << std::endl;
    return 0;
}
```

```
% Hello World! in LaTeX
\documentclass{article}
\begin{document}
Hello World!
\end{document}
```

```
% Hello World in Prolog

:- write('Hello World!') , nl .
```

```
# Hello World in Python
print "Hello World!"
```

Racket Syntax

- Racket finesses the issue of syntax by taking a rather **minimalist** approach.
- The syntax of Racket reflects little more than the “**abstract syntax**” of the language.

Abstract Syntax?

- Abstract syntax means recognizing that **various language constructs** have **specific parts** of specific types,

  but giving little attention to how the connection of those parts is represented.

Abstract Syntax Example: "if … then … else" construct

- We'll just call it "if" for short.
- Typical "if" has 3 parts:
  - test part
  - true branch
  - false branch

"if" in C or Java

```
if( … )     /* conditional part */
    {
    . . .   /* true branch */
    }
else
    {
    . . .   /* false branch */
    }
```

"if" in Python

```
if … :    # conditional part
   …      # true branch
else:
   …      # false branch
```

"if" in Racket

```
(if …   ; conditional part
   …    ; true branch
   …    ; false branch
 )
```

Racket's "if" is Abstract

- Racket's "if" looks like everything else in Racket. It is an S-expression:
  (*keyword* … *parts* …)
- This is both good and bad.

The Bad

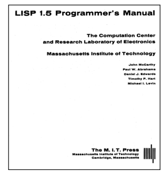
- Since everything looks alike, there are **few visual clues** in Racket, other than keywords and indentation, to differentiate one construct from another.

The Good

- The syntax is very **uniform**, so that part of language compilation is made trivial.
- Adding **new constructs** to the language is easy; we don't have to stop and think up symbols and syntax. We "just do it".
- **Programs can be read as data easily**, allowing new interpreters or transformation systems to be constructed.

Prototyping our Own Racket-Like Language

- Entire **S expressions** can be read with one statement, so no scanning step is necessary.
- An **S expression** (Symbolic Expression) is either:
  - an atom (symbol, numeral, string, etc.)
  - a list (begins and ends with parens)
- S expressions go back to Lisp 1.5, John McCarthy, 1959:


LISP 1.5 Programmer's Manual

The Computation Center
and Research Laboratory of Electronics

Massachusetts Institute of Technology

John McCarthy
Paul W. Abrahams
Daniel J. Edwards
Timothy P. Hart
Michael I. Levin

The M. I. T. Press
Massachusetts Institute of Technology
Cambridge, Massachusetts

Download from:

http://www.softwarepreservation.org/

From http://en.wikipedia.org/wiki/S-expression

- In Lisp, **S-expressions** are used to store both source code and data (see McCarthy Recursive Functions of Symbolic Expressions).
- S-expressions were originally intended only for data to be manipulated by **M-expressions**, but
- the first implementation of Lisp was an **interpreter** of S-expression encodings of M-expressions, and Lisp programmers soon became accustomed to **using S-expressions for both code and data**.

S Expression Examples

- (This is one S expression)
- (So are the following)
  - abcd
  - efg345
  - 678
  - (a (deeply (nested (S expression))))
- (S stands for "symbolic")

Reading S-Expressions

- One command (read) reads a single S-expression, which can then be decomposed using first and rest.

```
Welcome to DrRacket, version 5.0.1 [3m].
Language: racket; memory limit: 128 MB.
> (read)
                                              eof
(This is
     (a fairly complex)
     (multi-line S-expression))
```

returns value

```
'(This is (a fairly complex) (multi-line S-expression))
```

Decomposing a Read S-Expression

```
> (define x (read))              eof  ;; Executed
 (a (big list) with 5 elements)       ;; Entered in box
 (a (big list) with 5 elements)       ;; result
> (first x)
'a

> (rest x)
'((big list) with 5 elements)

> (first (rest x))
'(big list)

> (first (first (rest x)))
'big
```

Stand-Alone Execution

- In stand-alone execution, the box is replaced with just a command-line prompt.

```
$ a03/bin/a03      # Unix command-line

> (* 5 (/ mile hour))
'(2.2352 (meter) (second))

> (/ (* 5 (/ mile hour)) (/ meter second))
'(2.2352 () ())

>
```

Read-Eval-Print Loop (REPL)

- Read-Eval-Print means to repeat this cycle forever, or until "end-of-file" (EOF):
  - Read an expression, E.
  - Let R be the result of evaluating E.
  - Print R.

Example of a REPL

```
(define (repl)
  (begin                                ; sequential execution
   (prompt)                             ; side-effect: print prompt
   (let (
         (expression (read))            ; read 1 S-expression
         )
     (if (eof-object? expression)       ; test for eof
         expression                     ; if eof, just return eof
         (begin
          (top-level expression)        ; top-level eval & printing
          (repl)                        ; tail-recursive call
          ))
     )
   ))
```

top-level?

- The top-level handles things that might be at the top level, but not inside other expressions, such as:
  - (define ...)
- We have it do the Printing part of the REPL too.

Same semantics, Different syntax

- We'll develop a language using abstract syntax.
- If later desired, we can add on a front-end to provide different syntax, with the core processing remaining unchanged.

Example

- Prototyping version:

  (/ (* 5 meter) (* second second))

- Later version:

  5 meter/second^2

- It is the same core program, with a different "skin".

A Mini-Language

- Let's say we want a language to serve as a command-line evaluator based on Unicalc arithmetic.
- The syntax will resemble a subset of Racket.
- The focus is on Unicalc functions and equations.

Unicalc Language Elements

- Domain: **Numbers ∪ Units**
- Functions: * /
- Definable variables: $x, $y, $foo, $bar
- Special forms: **if**, **let**, **let**, **let***
- User-definable functions via **lambda expressions**

Language Semantics

- The **semantics** of the language will be defined by giving its **ueval** function:
- ueval = Unicalc eval
- This will distinguish it from the Racket eval function, which is built-in.

Example Arguments to ueval

- 0
- 3.14
- meter
- foot
- (/ meter foot)
- (* meter meter kg)
- (/ kg (* meter meter))
- (define $c (* 299792458 (/ meter second)))
- $c
- (/ $c (/ mile hour))

- In each case, the result of ueval should be the corresponding Unicalc quantity, unless it is a defintion, in which case the normalization of the quantity being defined should show.

Example Arguments to ueval

- In each case, the result of ueval should be the corresponding Unicalc quantity, unless it is a defintion, in which case the quantity defined should show.
- \> 0
  '(0 ()())
- \> 3.14
  '(3.14 ()())
- \> meter
  '(1 (meter) ())
- \> foot
  '(0.30479449356 (meter) ())
- \> (/ meter foot)
  '(3.2808991669107894 ()())
- \> (* meter meter kg)
  '(1 (kg meter meter) ())
- \> (/ kg (* meter meter))
  '(1 (kg) (meter meter))

Below $c is a user-defined variable:

- \> (define $c (* 299792458 (/ meter second)))
  '(299792458 (meter) (second)))
- \> $c
  '(299792458 (meter) (second))
- \> (/ $c (/ mile hour))
  '(670628744.7943213 ()())

Defining ueval in Racket

(define (ueval *exp env*) ...)

returns the result of evaluating the *expression* argument **exp**,
in the given *environment* **env**.

exp will ultimately be read as an S expression from the command line.

env will be represented as an association list, containing **bindings** for any variables defined by the user.

Basis for ueval

- The **basis** consists of S expressions that are constants and variables (non-lists).
- The **recursion part** consists of lists that represent composites, such as function application, etc.

Avoid "Magic Constants and Functions"

As the definition of what is a constant might change, avoid using built-in constants and functions to manipulate esoteric concepts. Instead, define your own constants and functions for such uses.

```
(define variable-escape #\$)      ; prefixed to designate user variable
(define definition-symbol 'define) ; designates a definition in user input

(define (variable-symbol? exp)
  (and (symbol? exp)
       (let ((string (symbol->string exp)))
         (and (> (string-length string) 1)
              (char=? variable-escape (string-ref string 0))))))
```

"Test-Driven" Design Strategy (crude version): A way to organize development

- **Write a test** that probably won't work.
- Add code to **make that test work**.
- Add another test, that probably won't work.
- Add code to make that test work.

. . .

- until done

"Test-Driven" Design Strategy (refined version)

. . .

- Add code to make that test work, *and* **refactor** the resulting code to make the overall code base better.

. . .

- until done

Beginning ueval

First test:

```
(check-expect
  (ueval 1 '())    ; constant 1 exp, empty env
 '(1 () ()))       ; quantity 1
```

Result:

expand: unbound identifier in module in: ueval

Beginning ueval

Make ueval evaluate units too:

```
(define (ueval expression env)
  (cond
    ((number? expression) (make-numeric-quantity expression))
       . . . other stuff will go in here …
    (else (ueval-error "expression not understood:" expression))))
```

Retry:

```
(check-expect (ueval 1 '()) '(1 () ()))
```

The only test passed!

Don't celebrate yet

```
(check-expect (ueval 1 '()) '(1 () ()))
(check-expect (ueval 'meter '()) '(1 (meter) ()))
```

Result:

expression not understood: meter
Ran 2 checks. 1 of the 2 checks failed.
check-expect encountered the following error instead of the expected value,
(1 (meter) ()).
:: expression not understood: meter   <<< Our message

Beginning ueval

Make ueval evaluate numeric constants:

```
(define (ueval expression env)
  (cond
    ((number? expression) (make-numeric-quantity expression))
    ((symbol? expression) (normalize-unit expression))
        . . . other stuff will go in here ...
    (else (ueval-error "expression not understood:" expression))))
```

Retry:

```
(check-expect (ueval 1 '()) '(1 () ()))
(check-expect (ueval 'meter '()) '(1 (meter) ()))
(generate-report)
```

Both tests passed!

Add Test for User Variable

```
(check-expect (ueval 1 '()) '(1 () ()))
(check-expect (ueval 'meter '()) '(1 (meter) ()))
(check-expect (ueval '$x '(($x (5 () ())))) '(5 () ()))
```
environment (association list)

Result:

Ran 3 checks.
1 of the 3 checks failed.
Actual value (1 ($x) ()) differs from (5 () ()), the expected value.
[Why did we get this? Because $x is a symbol]

Mezzanine ueval

```
(define (ueval expression env)
  (cond                                  ; Note: Ordering below is important!
    ((number? expression)              (make-numeric-quantity expression))
    ((variable-symbol? expression) (get-value expression env))
    ((symbol? expression)              (normalize-unit expression))
    (else (ueval-error "expression not understood:" expression))))
```

```
(check-expect (ueval 1 '()) '(1 () ()))
(check-expect (ueval 'meter '()) '(1 (meter) ()))
(check-expect (ueval '$x '(($x (5 () ())))) '(5 () ()))
```

All 3 tests passed!

Support code

; **get-value** gets the value of a variable in the environment.

```
(define (get-value var env)
  (let ( (found (assoc var env)) )
    (if found
        (second found)
        (ueval-error "unbound variable:" var))))
```

As in

```
*** error: unbound variable: $x
```

Support code

; **ueval-error** is an API interface for error messages.

Currently it just calls the built-in error function, which throws an exception, printing a message, then stopping.

As a first approximation:

```
(define (ueval-error msg exp)   (error msg exp))
```

Although there is an issue with this to be solved later.

Adding Tests for Arithmetic

```
(check-expect (ueval 1 '()) '(1 () ()))
(check-expect (ueval 'meter '()) '(1 (meter) ()))
(check-expect (ueval '$x '(($x (5 () ())))) '(5 () ()))
(check-expect (ueval '(* 2 2) '()) '(4 () ()))
```

Result:

```
   expression not understood (* 2 2)
Ran 4 checks. 1 of the 4 checks failed.
   check-expect encountered the following error instead of the
   expected value, (4 () ()).
    :: expression not understood (* 2 2)
```

ueval with arithmetic

```
(define (ueval expression env)
  (cond
   ((number? expression)          (make-numeric-quantity expression))
   ((variable-symbol? expression) (get-value expression env))
   ((symbol? expression)          (normalize-unit expression))
   ((non-empty-list? expression) ; Assumed to be a function application
     (ueval-operator (first expression) ; operator
                     (rest expression) ; arguments
                     env))
   (else (ueval-error "expression not understood" expression))))
```

ueval with arithmetic

```
(define (ueval-operator operator args expression env)
```

Environmental Issues

Connecting top-level to ueval

```
(define (repl)
  (begin                              ; sequential execution
    (prompt)                          ; side-effect: print prompt
    (let (
         (expression (read))          ; read 1 S-expression
         )
      (if (eof-object? expression)    ; test for eof
         expression                   ; if eof, just return eof
         (begin
          (top-level expression)      ; top-level eval & printing
          (repl)                      ; tail-recursive call
          ))
      )
    ))
```

simplest version of top-level

- (define (top-level expression)
  (ueval expression env))
- This works as long as the environment env never changes.
- However, it **can** change with define in our user language:

```
(define $x (/ meter second))
'(1 (meter) (second))

> (/ (/ mile hour) $x)
'(0.44703999999999994 () ())
```

Two Approaches to User Environment

- Non-Functional approach:

  Destructively modify a global variable, say global-env

- Functional approach:

  top-level returns an environment which is used in subsequent calls to repl.

  No global variable is used.

Common to Both Approaches

We are assuming that the environment is represented as an association list.

```
; Return a new environment in which variable is bound to a value

(define (newenv variable value env)
  (cons (list variable value) env))
```

Note that any old bindings of variable are “shadowed”. They won’t be seen in the new environment.

(So far, everything is still functional.)

Non-Functional Approach

```
; outside of repl:
(define global-environment '())

; called by repl:
(define (top-level expression)
  (if (user-definition? expression)
      (handle-definition expression)
      (handle-evaluation expression)))
```

handle-defintion

```
; Handle a user definition, already established to be
; in the form (define <var> <expression>)

(define (handle-definition definition)
  (let (
        (variable (second definition))
        (result (ueval (third definition) global-environment)) ; RHS
        )
    (begin
     (set! global-environment
           (newenv variable result global-environment))
     (print result)
     result)))
```

set! (“set-bang”)

- (set! <var> <value>)

  destructively assigns <value> to variable <var>

- ! is used in name of destructive commands, by Scheme convention.

Functional Approach

- Call the functional version **pure-repl**
  - The only impurities are these side-effects:
    - reading the input expression
    - printing the result
- pure-repl **takes** environment as argument
- pure-top-level **returns** environment for use in subsequent iterations of pure-repl

pure-repl

```
; A relatively-functional read-eval-print loop
; Only the print part is non-functional.
(define (pure-repl env)
  (begin
   (prompt)
   (let (
        (expression (read))
        )
     (if (eof-object? expression)
        expression
        (pure-repl (pure-top-level expression env))))))
```

Receives current environment

Returns new environment to next call of pure-repl.

pure-top-level

```
; pure-top-level evaluator decides whether
; we have a user definition or not.
; If a definition, pass to handle-definition,
; otherwise pass to handle-evaluation.

(define (pure-top-level expression env)
  (if (user-definition? expression)
    (pure-handle-definition expression env)
    (begin
      (print (ueval expression env))
      env)))                    ; return env unchanged
```

pure-handle-definition

```
; Handle a user definition, already established to be
; in the form (define <var> <expression>)

 (define (pure-handle-definition definition env)
  (let (
       (variable (second definition))
       (result (ueval (third definition) env)) ; RHS value
       )
    (begin
     (print result)
     (newenv variable result env)))) ; return new environment
```

Any Major Differences?

- If the destructive repl is interrupted for some reason, global-environment retains any definitions or redefinitions of variables.
- If pure-repl is interrupted, the accumulated environment is lost.
- If there are lots of redefinitions, the destructive version will be more space efficient.

Handling Forms Such as *let* and *lambda*

- It may be that our users will have no need to define new functions.
- Knowing how to implement functions will still be useful:
  - Maybe it will be a different language.
  - It helps us understand the base language Racket.

Hypothetical User Dialog

```
> (define $sq (lambda($x) (* $x $x)))
... representation of function ...

> ($sq meter)
'(1 (meter meter) ())

> ($sq 3)
'(9 () ())
```

Hypothetical User Dialog

```
> (define $double (lambda ($f) (lambda ($x) ($f ($f $x)))))
... representation of function ...

> (define $f ($double $sq))
... representation of function ...

> ($f 3)
'(81 () ())
```

Representing Functions

- To represent a function, we need three parts:
  - The variables in the function header (also called the **"formal parameters"**)
  - The **body expression** of the function
  - An environment containing bindings of any **imported variables** used in defining the function.

Closures

- The term for such a function representation is "closure".
- Example:

```
> (define $sq (lambda ($x) (* $x $x)))
  '(*closure* ($x) (* $x $x) ())
```

Here the environment was empty.

Continued Example

```
> (define $cube (lambda ($x) (* $x ($sq $x))))

  '(*closure* ($x) (* $x ($sq $x))
              (($sq (*closure* ($x) (* $x $x) ()))))
```

The environment had a binding for $sq, which is an **import** for the function to which $cube is bound.

That is a good thing, because otherwise $sq would be unbound.

Plumbing for Closures

- There are several aspects that need to be considered:
  - Evaluating a lambda expression produces a closure.
  - Applying a user-defined function vs. a built-in function.
  - Applying the closure that originated from a lambda expression.