PLT

Primitive Lisp Technology

Michael R. Gargano
mrg2202@colubmia.edu

COMS W4115 - Spring 2014

February 11, 2014

0.1 Language Description

PLT is a functional language in the LISP family of languages and has a minimalistic design philosophy. It is built out of a tiny set of primitive functions with the idea that everything else can be built upon those primitives as needed. Fundamentally only the concept of lambda is needed to implement all possible computations, but writing useful programs in such a language is probably even less productive than writing code in assembly language. Therefore, PLT will not go to that extreme (we do want to be able to do I/O too). It has enough primitives to make programming enjoyable even if it does not have as many bells and whistles as Scheme, Common Lisp, or the C family of languages.

0.2 Usage

PLT can be used for general purpose programming. One area where PLT shines is in its ability to handle symbolic processing tasks. Most other languages have some kludgy symbolic processing functionality or force the programmer to use string manipulation for such tasks. I wouldn't suggest writing any major enterprise software in this language but it should be great for experimentation and finding solutions to small problems.

0.3 Specifics

PLT works in a similar fashion to other LISPs. All function calls are in prefix notation, e.g. `(+ 2 4)`. Only four data types exist in PLT: integers, floats, symbols, and lists. The only object in the entire language is the S-expression (Symbolic expression). There are no statements or other kinds of constructs, everything is an expression that is evaluated. Two different types of S-expressions exist: atoms and lists. The simple rule is anything that's not a list is an atom with one small exception discussed later. The system, like OCaml and Scheme, has no looping construct. Recursive calls are the way to handle iteration in PLT. Since lists are the primary data structure of the language, recursive algorithms usually produce elegant code. Lists are a very versatile data structure. They allow us to have lists, trees, and a primordial type of record structure in our language.

0.4 Primitives

Operation	Explanation
atom?	Is this S-expression an atom or list?
first	Return the first element of a list.
rest	Return the tail of a list.
cons	Add an S-expression to the head of the list.
eq?	Checks to see if two symbolic atoms are equal.
cond	Switch-like conditional expression. Evaluates tests from top to bottom.
quote	Suppress evaluation of an S-expression.
lambda	Define an anonymous function.
define	Bind a value to a symbol.
symbol?	Is this S-expression a symbol?
+	Add numbers, float or integer.
−	Subtract numbers, float or integer.
∗	Multiply numbers, float or integer.
/	Divide numbers, float or integer.
mod	Modulo of two integers.
expt	Exponentiation, float or integer.
=	Compare two numbers (equality), float or integer (suggested integer only).
<	Compare two numbers (less than), float or integer.
>	Compare two numbers (greater than), float or integer.
<=	Compare two numbers (less than or equals), float or integer.
>=	Compare two numbers (greater than or equals), float or integer.
read	Read in an S-expression.
print	Print out an S-expression.
explode	Explode a symbol into a list of symbols, one for each character.
implode	Condense a list of symbols into a single symbol.

0.5 Examples

Let's start off with some trivial examples to define functions that will be used in our next example.

```
;; define "not", i said this language was minimal :)
(define not
    (lambda (b)
        (cond
            (b nil)    ; if the expression is true, eval to false
```

```
            (t t))))   ; otherwise it's false, eval to true

;; define "and", so minimal
(define and
    (lambda (a b)
        (cond
            (a (cond
                    (b t)    ; a and b are true
                    (t nil)))    ; a is true, b false... false
            (t nil))))   ; a is false, whole thing is false

;; define "or", so very minimal
(define or
    (lambda (a b)
        (cond
            (a t)        ; a is true, it's all true
            (b t)        ; b is true, it's all true
            (t nil))))   ; a and b are false :(

;; check to see if an atom is a number or not
(define number?
    (lambda (n)
        (cond
            ((not (atom? n)) nil)
            ((symbol? n) nil)
            (t t))))

;; get the length of a list, shows recursion
(define length
    (lambda (l)
        (cond
            ((eq? nil l) 0)
            (t (+ 1 (length (rest l)))))))

;; append two lists together
(define append
    (lambda (l1 l2)
        (cond
            ((eq? nil l1) l2)
            (t (cons (first l1) (append (rest l1) l2))))))
```

```
;; get a value from an association list based off a key
(define assoc
    (lambda (key alist)
        (cond
            ((eq? alist nil) nil)  ; list is empty, eval to nil
            ((eq? (first (first alist)) key) (first alist)) ; key is a match
            (t (assoc key (rest alist)))))) ; continue looking for key in rest
```

Below are some of the samples of using these functions.

`(and t nil)` → `nil`
`(or t nil)` → `t`
`(and t t)` → `t`
`(not t)` → `nil`
`(number?  2)` → `t`
`(number?  (quote k))` → `nil`
`(length (quote (a b c d)))` → `4`
`(length (quote (a)))` → `1`
`(append (quote (a b c)) (quote (d e f)))` → `(a b c d e f)`
`(append (quote (a b)) nil)` → `(a b)`
`(accoc (quote k1) (quote ((k1 .  2) (k2 .  5))))` → `(k1 .  2)`
`(accoc (quote k3) (quote ((k1 .  2) (k2 .  5) (k3 .  3))))` → `(k3 . 3)`
`(accoc (quote k4) (quote ((k1 .  2) (k2 .  5))))` → `nil`

The next example is a symbolic differentiation example adapted from the example in SICP[1]. It's a little long, but shows how much can be built up from the primitives to perform some computer algebra.

```
;; is the first atom a "+" sign
(define sum?
    (lambda (exp)
        (eq? (first exp) (quote +))))

;; is the first atom a "*" sign
(define prod?
    (lambda (exp)
```

```
        (eq? (first exp) (quote *))))

;; is the first atom "expt"
(define expt?
    (lambda (exp)
        (eq? (first exp) (quote expt))))

;; ex.  (+ 1 2) -> 1
(define first-term
    (lambda (exp)
        (first (rest (exp)))))  ; get second item in list

;; ex. (+ 1 2) -> 2
(define second-term
    (lambda (exp)
        (first (rest (rest (exp))))))    ; get third item in list

;; create a simplified sum expression with the two given expressions
(define create-sum
    (lambda (exp1 exp2)
        (cond
            ; both exp1 and exp2 can just be added together
            ((and (number? exp1) (number? exp2)) (+ exp1 exp2)) ; sum the digits
            ((and (number? exp1) (= exp1 0)) exp2) ; if adding with zero
            ((and (number? exp2) (= exp2 0)) exp1)
            ; otherwise do the usual
            (t (cons (quote +) (cons exp1 (cons exp2 nil)))))))

;; create a simplified prod expression with the two given expressions
(define create-prod
    (lambda (exp1 exp2)
        (cond
            ; both exp1 and exp2 can just be multiplied together
            ((and (number? exp1) (number? exp2)) (* exp1 exp2)) ; do the prod
            ; if one expression is a number and that number is 0, eval to 0
            ((or
                (and (number? exp1) (= exp1 0))
                (and (number? exp2) (= exp2 0))) 0)
            ; if one expr is a number and that number is 1, eval to the other expr
            ((and (number? exp1) (= exp1 1)) exp2)
```

```
            ((and (number? exp2) (= exp2 1)) exp1)
            ; otherwise do the usual
            (t (cons (quote *) (cons exp1 (cons exp2 nil)))))))

;; negate expression
(define negate
    (lambda (exp)
        (cond
            ((number? exp) (- 0 exp))
            (t (make-prod -1 exp)))))

;; create a simplified difference expression with the two given expressions
(define create-diff
    (lambda (exp1 exp2)
        (cond
            ((and (number? exp1) (number? exp2)) (- exp1 exp2))
            ((and (number? exp1) (= exp1 0)) (negate exp2))
            ((and (number? exp2) (= exp2 0)) exp1)
            (t (cons (quote -) (cons exp1 (cons exp2 nil)))))))

;; create a expt expression
(define create-expt
    (lambda (base expt)
        (cond
            ((and (number? base) (number? expt)) (expt base expt))
            (t (cons (quote expt) (cons base (cons expt nil)))))))

;; the main show, call this with a quoted symbolic expression and the variable
;;  with which the derivative is to be taken with respect to
(define derivative
    (lambda (exp var)
        (cond
            ; it's a constant, dc/dx = 0
            ((number? exp) 0)
            ; it's a variable
            ((symbol? exp)
                (cond
                    ((eq? exp var) 1)    ; if it's with respect to var
                    (t 0)))              ; otherwise the deriv. is 0
            ; it's a sum of two expressions
```

```
((sum? exp)        ; take derivative of each term
                   ;  and leave sum
    (create-sum
        (derivative (first-term exp) var)
        (derivative (second-term exp) var)))
; it's a product of two expressions
((prod? exp)
    (create-sum
        (create-prod
            (derivative (first-term exp) var)
            (second-term exp))
        (create-prod
            (first-term exp)
            (derivative (second-term exp) var))))
; it's an exponentiation
((expt? exp)
    (create-prod
        (create-prod
            (second-term exp)
            (create-expt
                (first-term exp)
                (create-diff (second-term exp) 1)))
        (derivative (first-term exp))))
(t (print (quote "error cannot take derivative"))))))
```

So, in about 100 lines of code, we've created a small computer algebra system that can take derivatives of some polynomial equations. Below are some samples of calling the derivative function shown above.

`(derivative (quote 2) (quote x))` → `0`
`(derivative (quote x) (quote x))` → `1`
`(derivative (quote (+ x 27)) (quote x))` → `1`
`(derivative (quote (* x y)) (quote x))` → `y`
`(derivative (quote (* x y)) (quote y))` → `x`
`(derivative (quote (+ x (* 2 x))) (quote x))` → `3`
`(derivative (quote (expt x 4)) (quote x))` → `(* 4 (expt x 3))`
`(derivative (quote (* (* x y) (+ x 27))) (quote x))` → `(+ (* y (+ x 27)) (* x y))`

0.6 Other Things To Note

Some additional things we can learn about PLT from the examples is that `nil` is equivalent to `()` and they both represent falsehood. They are both atoms and lists in the language. The symbol `t` evaluates to itself, like `nil`, and can be used to represent truth. Symbols can be contained in double quotes (`"`), they are still symbols but can have spaces and other characters normally not permitted in other Lisps. PLT is a lexically scoped Lisp-1 (meaning there is only one namespace for variables and function names). The language contains dotted pairs. When creating a list, each cell has a pointer to a value and a pointer to the next item in the list until the last item in the list points to `nil` as the next list item. This cell structure can be used to point to two values instead of another item in this list, when this happens the structure is represented as a dotted pair [`(a . b)`]. Finally, comments begin with `;` and terminate at the end of the line.

References

[1] Harold Abelson, Gerald Jay Sussman, and Juile Sussman. *Structure and Interpretation of Computer Programs*. 2nd. Cambridge MA: The MIT Press, 1996. ISBN: 0262510871.

[2] Alexander Burger. *The PicoLisp Reference*. URL: `http://software-lab.de/doc/ref.html`.

[3] Paul Graham. *The Roots of Lisp*. May 2001. URL: `http://www.paulgraham.com/rootsoflisp.html`.

[4] John McCarthy. *LISP 1.5 Programmer's Manual*. 2nd. Cambridge MA: The MIT Press, 1962. ISBN: 0262130114.