PLT

Primitive Lisp Technology

Language Reference Manual

Michael R. Gargano
mrg2202@columbia.edu

COMS W4115 - Spring 2014

March 13, 2014

1.1 Introduction

This document will describe the the PLT language. Both its syntax and semantics will be expounded upon. It is intended to be the definitive reference for the language.

1.2 Lexical Conventions

A program is contained entirely in a single file. This file is then processed as a stream of tokens.

1.2.1 Tokens

There a handful of tokens in PLT: symbols, integers, doubles, and delimiters for dotted pairs and lists (a special type of dotted pair). *White space* is considered to be spaces, tabs, newlines, and carriage returns. It is used to delineate tokens, but is otherwise ignored.

1.2.2 Comments

Comments begin with the the semicolon character (`;`) and continue until the end of the line. The semicolon character cannot occur inside of unquoted symbols, otherwise from where it appears, to the end of the line will be considered a comment. Double quoted symbols can contain the semicolon character.

1.2.3 Keywords

PLT does not really contain any keywords. There are many built-in special forms and functions for ease of use, but the symbols those functions are bound to does not make them any more special than any other symbol in the system. They are just processed a little differently than user defined functions and variables. The one symbol that *is* treated specially is the symbol `nil`. `nil` is a symbol, but is also considered to be an empty list. This duality makes it special and it is actually a specific token recognized by the compiler.

1.2.4 Integers

An integer is a sequence of decimal digits that may optionally begin with a sign (`+` or `-`). All integers are signed (whether the sign is specified or not)

and is equivalent in size and range to OCaml's `int` (and the corresponding rules for 32-bit and 64-bit machines).

1.2.5 Doubles

Doubles have an integer part, a decimal point, a fraction part, an exponent part (denoted by `e` or `E`) containing an optionally signed integer. The integer, fraction, and exponent parts contain a sequence of decimal digits. The integer part or fraction part of the the double may be omitted, but not both. Either the decimal point or the exponent part of the double may be omitted, but not both. This IEEE double-precision floating point number is equivalent to OCaml's `float` and C's `double` in size and range.

1.2.6 Symbols

Symbols in PLT can take two different forms, a doubly quoted symbol or a regular unquoted symbol.

Unquoted

Symbols in this form must start with an alpha character (lower or upper) or any of the following characters: `‘ ’ ~ ! @ # $ % ^ & * + = : ? / < > . , - _ + { } [ ] | \` followed by any combination and number (including zero) of the preceding characters mentioned and/or digits (`0 - 9`). One exception is the symbol consisting of only a single dot (`.`). While this is allowed by the above definition, the single dot is used in the definition of dotted pairs and must be ether used in combination with other characters or as part of a double quoted symbol.

Double Quoted

Symbols in this form start with a double quote character (`"`) and end with one as well. Any character inside of the double quotes are allowed except for another double quote character. This gives the user a lot of flexibility in terms of naming, but care must be taken as `"27"` and `"-2.36e+248"` are valid double quoted symbols. In PLT these double quoted symbols take on almost a dual role as strings and can be used as such.

1.3 Syntax Notation

Most of the syntax for PLT has been mostly defined above (believe it or not). Only two other syntactic structures remain to be defined: dotted pairs and lists.

1.3.1 Dotted Pairs

The fundamental underlying structure in PLT is the dotted pair (sometimes called a cons cell or ordered pair). The dotted pair structure has two pointers that can point to symbols or another dotted pair. The dotted pair has the following structure:
`(` *first S-Expression pointed to* `.` *second S-Expression pointed to* `)`
So, after its namesake, we can see a dotted pair is a pointer to two other S-Expressions with a dot (surrounded by whitespace) separating the two of them.

1.3.2 Lists

Lists, at the end of the day, are really just a special case in the possible structures of dotted pairs. A list is just dotted pairs arranged in the form of a singly-linked list. A list has the following structure:
`(` *element1 element2 element3* `)`
This can be equiviallently exapanded into the following dotted list structure:
`(` *element1* `.` `(` *element2* `.` `(` *element3* `.` `nil` `)))`
The singly-linked list show in the structure above, like a lists, is `nil` terminated.

1.4 Objects

PLT only contains one type of object, the S-Expression (Symbolic Expression). There are two types of S-Expressions: atoms and dotted pairs. Atoms consist of the following: symbols, integers, and doubles. Dotted pairs, as we saw before, can contain atoms and/or other dotted pairs.

1.4.1 Forms

A form is any object that is meant to be evaluated: integers, doubles, symbols, compound forms. PLT programs consist of lists of forms, evaluating one after the other.

1.4.2 Self Evaluating Forms

These include integers, doubles, and `nil`. These objects all evaluate to themselves.

1.4.3 Variables

Symbols can be used to name variables. When a symbol is evaluated as a form, the value of the bound variable is the result.
Local variables are lexically scoped in PLT.

1.4.4 Compound Forms

A non-empty list that is a function form, special form, or lambda form.

1.4.5 Function Forms

A non-empty list where the first element is the name of a function that is meant to be called on the arguments (what is obtained after individually evaluating the remaining elements in the list).
`(` *func-name arg1 arg2 ...* `)`

1.4.6 Special Forms

A form with special evaluation rules. Examples in PLT include: `cond`, `quote`, `define`, and `lambda`. It is a list where the first element is the special operator. The other items in the list may or may not be evaluated as they in the other forms.
`(` *special-op arg1 arg2 ...* `)`

1.4.7 Lambda Forms

A non-empty list where the first element is a lambda function meant to be applied to the following arguments (what is obtained after individually evaluating the remaining elements in the list).
`(` `(` `lambda` `(` *arg-id1 arg-id2 ...* `)` *body...* `)` *arg1 arg2 ...* `)`

1.5 Predicates

Predicates answer true/false questions. The value for true in PLT is the symbol `t`, which evaluates to itself. The value for false is `nil`.

Function	Description
`atom?` *sexpr*	Is the S-Expression an atom?
`symbol?` *sexpr*	Is the S-Expression a symbol?
`eq?` *sexpr sexpr*	Are two symbolic atoms or two structures equal?

1.6 Control Structure

Special Operator	Description
`cond (`*test consequent ...*`) ...`	Evaluates test, if it evaluates to true, performs the consequents returning the last value as the result, otherwise it moves to the next test. If no tests succeed, the value of `cond` is `nil`
`define` *id form*	Bind the value of the evaluated form to the identifier.
`quote` *sexpr*	Simply returns the unevaluated form as is. Usually used for defining data.
`lambda (`*arg-id ...*`)` *body*	Create a function that takes the arguments in the arg list and executes the body when called. Returns last value as the result.
`begin` *sexpr ...*	Sequence the S-Expressions, evaluating to the value of the the last one.

In PLT there are no explicit control structures for looping. Looping is done through recursion. Recursively defined functions can provide equivalent functionality to looping.

1.7 Numbers

1.7.1 Comparisons

Predicate	Condition
`=` *arg1 arg2*	Are the two numbers the same.
`<` *arg1 arg2*	First number less than the second.
`>` *arg1 arg2*	First number is greater than the second.
`<=` *arg1 arg2*	First number is less than or equal to the second.
`>=` *arg1 arg2*	First number is greater than or equal to the second.

*All work with integers and doubles. It is suggested that `=` only be used with integers.

1.7.2 Arithmetic Operations

Function	Description
`+` *arg ...*	Sum all of the arguments together. Add the argument to 0 if only one is given.
`-` *arg ...*	Successively subtract the arguments from the first. Subtract the argument from 0 if only one is given.
`*` *args ...*	Multiply all of the arguments together.
`/` *arg ...*	Successively divide the first argument by the following arguments. If only one argument is supplied, its inverse is returned.
`mod` *num div*	Modulus of number and divisor.

*All work with integers and doubles.

1.7.3 Exponential and Logarithmic Functions

Function	Description
`expt` *base pow*	Raises the first number to the power of the second.
`log` *num base*	Log of the first number in the base of the second.

1.8 List/Dotted Pair Operations

Function	Description
`cons` *sexpr dp*	Add an S-Expression to the head of a list. Create new dotted pair with the first pointer pointing to sexpr and the second pointer pointing to dp.
`first` *dp*	Return the first element of the list or first pointer in the dotted pair.
`rest` *dp*	Return the tail of the list of the second pointer in the dotted pair.

1.9 Input/Output

Function	Description
`read`	Read in an S-Expression.
`print` *sexpr*	Print out an S-Expression.

1.10 Symbol Manipulation

Function	Description
`explode` *symbol*	Explode a symbol into a list of symbols, one for each character in the original symbol.
`implode` *list*	Condense a list of symbols into a single symbol.

1.11 Compiler Front-End

1.11.1 ast.mli

```
(* three types of atoms (nil is really a symbol, but
handled specially) *)
type atom =
    Symbol of string
  | Int of int
  | Double of float
  | Nil

(* S-expressions *)
type sexpr =
    Atom of atom
  | DottedPair of sexpr*sexpr

(* the root *)
type prog =
    Prog of sexpr list
```

1.11.2 scanner.mll

```
{ open Parser
  exception SyntaxError of string  (* define a syntax error
exception *)
}

(* predefined regexes *)
(* int and float stuff *)
let digit = ['0'-'9']
let digits = ['0'-'9']+
let sign = ['-''+']
let e = ['e''E']sign?
(* symbol stuff *)
let lalpha = ['a'-'z']
let ualpha = ['A'-'Z']
let oalpha =
['`''~''!''@''#''$''%''^''&''*''+''=''':''?''/''<''>''.',''
-''_''{''}''[']''|''\\''\'']
let symbol_start = (lalpha|ualpha|oalpha)

(* token parsing *)
rule token =
parse [' ' '\t' '\r' '\n'] { token lexbuf }
| ';' { comment lexbuf }  (* comments *)
| '(' { LPAREN }
| ')' { RPAREN }
| '.' { DOT }
| "nil" { NIL }
(* double matching *)
| (sign)?(digits)e(digits) as double
{ DOUBLE(float_of_string double) }
| (sign)?(digit*)'.'(digits)(e(digits))? as double
{ DOUBLE(float_of_string double) }
| (sign)?(digits)'.'(digit*)(e(digits))? as double
{ DOUBLE(float_of_string double) }
(* integer matching *)
| (sign)?(digits) as integer { INTEGER(int_of_string
integer) }
(* symbol matching *)
| '"' { let buf = Buffer.create 27 in
          Buffer.add_char buf '"';
          read_dbl_quoted_symbol buf lexbuf }
```

```
| symbol_start+(symbol_start|digit)* as symbol
{ SYMBOL(symbol) }
| eof { EOF }
| _ { raise (SyntaxError("Syntax error: " ^ Lexing.lexeme
lexbuf)) }

(* ignore comments until end of line or file *)
and comment =
parse ['\n' '\r'] {token lexbuf}
| _ { comment lexbuf}
| eof { EOF }

(* different state for double quoted symbols *)
and read_dbl_quoted_symbol buf =
parse '"' { Buffer.add_char buf '"';
            SYMBOL(Buffer.contents buf) }
| [^'"'] { Buffer.add_string buf (Lexing.lexeme lexbuf);
           read_dbl_quoted_symbol buf lexbuf }
```

1.11.3 parser.mly

```
%{ open Ast %}

%token LPAREN RPAREN DOT NIL EOF
%token <int> INTEGER
%token <float> DOUBLE
%token <string> SYMBOL

%start prog
%type <Ast.prog> prog

%%

prog:
  sexprs EOF            { Prog(List.rev $1) }

sexpr:
  atom                  { Atom($1) }
| LPAREN RPAREN         { Atom(Nil) }
| LPAREN sexprs RPAREN  { let rec buildList = function
                            [] -> Atom(Nil)
                            |h::t ->  DottedPair(h, buildList
t)
                          in buildList (List.rev $2) }
| LPAREN sexpr DOT sexpr RPAREN { DottedPair($2, $4) }

sexprs:
  sexprs sexpr          { $2::$1 }
| sexpr                 { [$1] }

atom:
  SYMBOL                { Symbol($1) }
| INTEGER               { Int($1) }
| DOUBLE                { Double($1) }
| NIL                   { Nil }
```

1.11.4 main.ml (pretty printer)

```
open Ast

type action = Debug | Normal

(* pretty print atoms *)
let ppatom p a = match p with
    Symbol(s) -> (match a with
                   Debug ->  print_string ("sym:" ^ s)
                  |Normal ->  print_string s)
   |Int(i) -> (match a with
               Debug -> print_string "int:"; print_int i
               |Normal -> print_int i)
   |Double(d) -> (match a with
                  Debug -> print_string "dbl:"; print_float
d
                  |Normal -> print_float d)
   |Nil -> (match a with
            Debug -> print_string "sym:nil"
            |Normal -> print_string "nil")

(* pretty print S-expressions *)
let rec ppsexpr p islist act = match p with
    Atom(a) -> ppatom a act
   |DottedPair(se1,se2) -> if islist then () else
print_string "( ";
                    (match se1 with
                      Atom(a) -> ppatom a act
                      |_ -> ppsexpr se1 false act);
                    print_char ' ';
                    (match se2 with
                      Atom(Nil) -> print_char ')'
                      |Atom(_ as a) -> print_string ". ";
                                       ppatom a act;
                                       print_string " )"
                      |_ -> ppsexpr se2 true act)

(* pretty print the program *)
let pprint p a = match p with
    Prog(ss) -> List.iter (fun i -> ppsexpr i false a;
                                    print_newline ();
                                    print_newline ()) ss
```

```
(* starting point *)
let _ =
  let action =
    if Array.length Sys.argv > 1 then
      try
        List.assoc Sys.argv.(1) [ ("-d", Debug) ]
      with Not_found -> Normal
    else
      Normal in
  let lexbuf = Lexing.from_channel stdin in
   let prog = Parser.prog Scanner.token lexbuf in
    pprint prog action
```