PLisp: A Lisp to PostScript Compiler

John C. Peterson
Yale University

ABSTRACT

PostScript is a programming language for the description of images on a printed page. Many laser printers now use PostScript. A very rich set of graphic operations which allow complex images to be described in a simple and concise manner are provided by the PostScript language.

While many systems, such as Tex, pic, or MacDraw, generate PostScript code, none of these allow the user access to the full power of the PostScript language. However, programming directly in PostScript can be difficult. Both the syntax and semantics of the language are oriented more toward simplicity of execution rather than ease of programming. The purpose of this compiler is to take the basic structure of Common Lisp and use it as an alternative representation of a PostScript program. While the basic graphic operators of PostScript are retained, Lisp-like function definition, control flow, and variable binding become available. Macros, defined constants, and function libraries have also been provided. The PLisp compiler generates PostScript code which is guaranteed not to contain errors in stack usage.

1 Introduction

PostScript [1] is a page description language which provides a device independent description of a printed page. Many laser printers now contain PostScript interpreters. Powerful graphic operators allow complex images to be described in a clear and concise manner.

PostScript can also be viewed as a programming language. The problem is that PostScript is meant to be directly interpreted. Programming in PostScript is sort of like programming in assembly language: you have to always keep track of what is where on the stack, there is only a limited notion of variables and environments, and control flow is reduced to a few simple primitives. Features found in high level languages are lacking, such as macros, compile time diagnostics, and complex control structures.

With this in mind, a new language has been created: PLisp (PostScript Lisp). PLisp takes its overall syntax, control structures, and compile time environment from Common Lisp, while its set of operators and datatypes come from PostScript. The PLisp compiler translates PLisp programs to PostScript, which can then be run on any PostScript engine.

2 The PLisp Language

PLisp is a mixture of Common Lisp and PostScript. Some knowledge of these languages is required to understand PLisp. "Common Lisp the Language" [**?**], "PostScript Language Tutorial and Cookbook" [**?**], and "PostScript Language Reference Manual" [**?**] are recomended references on these subjects.

The goal of PLisp is to look as much like Common Lisp as possible: programs are parsed using the Lisp reader, lisp-like function and macro definitions are provided, and many Lisp functions are compiled into PostScript equivalents.

2.1 Syntax

PLisp syntax is Lisp syntax. Numbers, strings, arrays, comments, and identifiers are all read by the Lisp reader. Syntax is entirely inherited from the underlying Lisp system. With a few exceptions, this syntax is completely compatible with PostScript.

[1] PostScript is a trademark of Adobe Systems Incorporated

The primary syntactic difference between PLisp and PostScript is the treatment of case in identifiers. PostScript is case sensitive; Lisp is generally not. PLisp attempts to hide this difference, but more advanced PLisp users may need to know how PLisp treats symbols. Unless alphabetic characters in Lisp identifiers are escaped[2], they are converted to uppercase at read time by the Lisp system. Since most predefined PostScript identifiers are lowercase, Lisp symbols containing no lowercase letters are output in lowercase. If any character in an identifier is lowercase, case is preserved from Lisp to PostScript. Normal Lisp case conversion rules still apply. Any symbol without escaped lowercase letters will be output in lower case only. Symbols with one or more escaped lower case letters will be output to PostScript according to Lisp case rules.

Some commonly used PostScript symbols contain upper case letters. To make case problems more transparent to the user, such symbols have separate PostScript names which supercede all other case conversion rules. This capability has been used to predefine all common mixed case PostScript symbols (like `FontDirectory`) with the proper case. These identifiers need not be specially treated by the user; `FONTDIRECTORY` or `fontdirectory` would be changed to `FontDirectory` by this feature.

One other minor syntactic difference exists. The <, >, and / characters are special to PostScript and should not be used in symbols. The lisp functions <, >, <=, >= , /, and /= are an exception: these are compiled directly to the proper PostScript functions.

Numbers and strings are read in Lisp syntax and converted on output to PostScript syntax without any loss of information. The Lisp character type is converted to integer character codes. Both arrays and lists are transformed into PostScript arrays.

2.2 Functions and Variables

One of the primary contributions of PLisp to PostScript is the addition of Lisp-like functions and variables. PLisp provides dynamic variables (*special* variables in Common Lisp terminology) and Lisp style functions.

Variables in PLisp are essentially the same as variables in Common Lisp. The only difference is that all variables are dynamic (special); lexical scoping is not provided. Variables may be declared and initialized with `defvar`, as in Lisp. Many special forms bind variables, such as `lambda`, `let`, and `do`. As in Common Lisp, evaluating a variable returns its most recent active binding.

[2]Lisp uses backslash as a single character escape or vertical bars as a multi character escape

PLisp functions resemble Common Lisp functions. They are defined with `defun`, support `&optional`, `&rest`, and `&key`, and can return multiple values. Anonymous functions, those defined by `#'(lambda ...)`, cannot use `&optional, &rest,` or `&keyword` parameters. Unlike Lisp functions, PLisp functions must declare exactly how many values are returned. A number specifying the number of values returned can be placed immediately after the argument list of the `defun`, as in

```
(defun foo (x) 1 (+ x 2))
(defun bar (x y) 2 (values x y))
(defun baz (x y) 0 (moveto x y))
```

This allows PLisp to know at compile time exactly how many values are on the stack. If no number is present, 0 will be assumed. The value of the variable `*assume-0-res*` determines whether an error, a warning, or nothing will occur.

All argument processing is done at compile time. At execution time, every function takes a fixed number of arguments. When a call to a function using `&optional`, `&rest`, or `&key` is compiled, extra parameters may be added during generation of code for the caller. This style places a number of restrictions on function calling. Since function processing is done at compile time, all arguments must be provided when `apply` or `funcall` dispatches an unknown function at run time. Keywords do not exist at execution time, precluding the recognition of arguments which evaluate to keywords. A `&rest` argument will not pick up keyword arguments. Since PostScript does not have a list data type, `&rest` arguments are placed in an array instead of a list. Only those keywords declared by the function being called may appear in the calling sequence. Unrecognized keywords are a compile time error.

Multiple values are handled much the same as in Common Lisp. There are, however, a few significant differences. As in Lisp, when only one value is required, as when an expression is an argument to a function, extra values are discarded. Unlike Common Lisp, extra NIL's are never added when fewer values are provided than required. The body of a function must return exactly the number of values specified in the `defun`. Sometimes a `values` function must be added to avoid compiler errors. For example,

```
(defun width (x) 1
    (stringwidth x))
```

would be an error since `stringwidth`, a PostScript function, returns two values: both the width and height. To make the number of returned values work out properly would require

```
(defun width (x) 1
    (values (stringwidth x)))
```

2.3 Program Structure

PLisp programs resemble Common Lisp programs. Constants are declared by `defconstant`, variables by `defvar`, macros by `defmacro`, and functions by `defun`. Forms which are directly interpreted by the PLisp compiler are `top level` forms. Declarations such as `defvar` are examples of top level forms. The main program is formed by wrapping all but the top level forms in a `progn`. Top level forms are not a part of the main program but are instead processed at compile time.

`Defconstant` differs slightly its Common Lisp counterpart. The initial value is a Lisp, not PLisp, expression and is evaluated by the Lisp evaluator. No PLisp items may be referenced in these expressions except previously defined constants. T and NIL are treated as defconstants for values `true` and `false`.

Macros in PLisp are identical to macros in Lisp. The body of the macro is written in Common Lisp instead of PLisp. The value returned by the macro must be valid PLisp code.

2.4 Control Structures

PLisp control structures are completely inherited from Common Lisp. The available structures are `cond`, `do`, `do*`, `dolist`, `dotimes`, `dodict`, `map`, `mapcar`, `if`, `prog1`, `progn`, `unless`, and `when`. All of these work almost exactly as in Lisp. Boolean values used in conditionals must be `true` or `false` instead of non-NIL or NIL. Note that all PostScript relational functions return proper boolean values. `T` and `nil` compile into `true` and `false`. Another restriction is that all consequents of an `if` or `cond` must return the same number of values.

The `map` function is restricted to result types of nil and `array`. Only one sequence may be mapped in `map` and `mapcar`. When the sequence is a string, the variable is bound to integer character codes. The `dodict` function is not in Common Lisp, but has been provided to map over dictionaries. The `while` function is provided but is not properly a part of Common Lisp.

Most of the PostScript control functions are also available. These should not be normally

be used, but may be necessary. The proc argument to these functions is usually a lisp functional in `#'` notation, such as `(loop #'(lambda () ... ))`.

2.5 Other Common Lisp Functions

When a Common Lisp and PostScript function perform the same basic operation, the operation is available to the PLisp user from either language. Often, the Lisp version is preferable to use. For example, the PostScript `add` and the Lisp `+` perform the same operation. Either `(add a b)` or `(+ a b)` could be used in a PLisp program. When using the PostScript version, however, `add` must be given exactly two arguments, while the `+` function takes an arbitrary number of arguments. The PLisp compiler also collects constant terms in the `+` function. Since `+` is more flexible than `add`, the PLisp programmer is encouraged to use the Common Lisp version of this function.

The set of Common Lisp functions compiled into PostScript is presented in section 5.

2.6 Libraries

One of the advantages of PLisp over PostScript is the library. A library is a file of PLisp function, macro, constant, and variable definitions whose contents are loaded on demand. By placing `(library "file")` in a PLisp program, undefined names in the program are looked up in the library file. Only those components of the library file needed to resolve undefined names are actually included in the compiled program. Libraries are searched in the order in which they are declared.

PLisp always attempts to resolve names locally, within either the main program or the current library. When no local definition of a symbol exists, two different resolution strategies are used. For an unresolved symbol in a library function, the main program is searched for a symbol definition. No other libraries are consulted. In the main program, all libraries are searched in the order specified. When an unresolved symbol in variable context is encountered, it is assumed to be an undeclared free variable and initialized to 0. Unresolved functions are a compile time error.

3 PLisp Examples

By far the easiest way to understand PLisp is to look at some examples. The printed output from these examples is included at the end of this report.

3.1 The Title Page

The title page of this report was generated by the following small program:

```
;;;  This function sets the font to Times-Roman
(defun times (size) 0
   (setfont (scalefont (findfont 'Times-Roman) size)))  ; Choose a font

;;;  A simple function to place a string at some given point
(defun showat (str x y) 0
   (moveto x y)
   (show str))

;;; Main program.
(times 150)                     ; select 100 point type
(setlinejoin 1)                 ; rounded line joins
(dolist (width '(12 8 4))       ; Slowly narrow the stroke width
     (moveto 150 450)           ; where to place title
     (charpath "PLisp" nil)     ; Outline of characters
     (setlinewidth width)       ; Set the width
     (setgray (* (- width 4) 0.1)) ; Slowly get blacker
     (stroke))                  ; stroke it

(times 40)                      ; now for the easy part
(showat "A " 130 350)           ; make Lisp and PostScript line up
(let ((savex (currentpoint)))   ; save point at start of "Lisp"
   (show "Lisp to")
   (showat "PostScript Compiler" savex 310)) ; line it up

(times 30)
(showat "John C. Peterson" 250 190)
(showat "University of Arizona" 250 160)
```

```
(showpage)  ; Needed to actually print the page
```

The symbol `Times-Roman` is a known mixed case symbol. It would have been safer to use |`Times-Roman`|. The list `'(12 8 4)` will be converted into a PostScript array since PostScript does not support a list data type.

3.2 A Centering Function

This function centers a line of text. The two optional arguments govern the placement of the centered text. If both are omitted, the text is centered over the current point. When one extra argument is provided, this is y coordinate of the center. The x coordinate is assumed to be the center of the page. When both optionals are provided, they are the x and y coordinate.

Note the nonstandard use of `let` to bind multiple values. This feature is not found in Common Lisp but was added to make multiple values easier to use.

```
(defconstant page-right (* 72 8.5))        ; Right margin of paper
(defconstant page-center (/ page-right 2)) ; Center of paper

(defun center (str &optional (x 0 x-supplied) (y 0 y-supplied)) 0
   (cond (y-supplied (moveto x y))        ; When both are supplied
         (x-supplied (moveto page-center x))); One extra supplied
   (let (((lx ly) (stringwidth str)))
         ;; Stringwidth returns 2 values to lx and ly
      (rmoveto (- (/ lx 2)) (- (/ ly 2)))) ; relative move half way
   (show str))

;;;  Main program begins here

(setfont (scalefont (findfont 'Times-Roman) 20)) ; Choose a font

(moveto 100 100)
```

```
(center "100 100")

(center "At 200" 200)

(center "At 300 300" 300 300)
(showpage)
```

3.3 The Wedge Program

The following example is taken from the "PostScript Tutorial and Cookbook", page 133 [**?**]. Here, the wedge program has been transcribed from PostScript into PLisp.

```
(defun inch (x) 1 (* x 72))

(defun wedge () 0
    (newpath)
    (moveto 0 0)
    (translate 1 0)
    (rotate 15)
    (translate 0 (sin 15))
    (arc 0 0 (sin 15) -90 90)
    (closepath))

(gsave)
(translate (inch 3.75) (inch 7.25))
(scale (inch 1) (inch 1))
(wedge)
(setlinewidth 0.02)
(stroke)
(grestore)

(gsave)
(translate (inch 4.25) (inch 4.25))
(scale (inch 1.75) (inch 1.75))
(setlinewidth 0.02)
(dotimes (i 12)
```

```
        (setgray (/ (1+ i) 12))
        (gsave)
        (wedge)
        (gsave) (fill) (grestore)
        (setgray 0)
        (stroke)
        (grestore)
        (rotate 30))
(grestore)
(showpage)
```

3.4 A Font Catalog

This program prints a page for every every built-in font on the imaging device. Only the page for the "Symbol" font is actually included in this report.

```
;;;;  Program to print out fonts
;;;;  Warning!  This takes a long time!

;;;  These constants locate the printout on an 8.5 x 11 page.
;;;  All scaling is in points.
;;;  A 16 x 16 grid is used to show 256 chars.

(defconstant top 630)    ; top of grid
(defconstant delta 30)   ; distance between lines
(defconstant bottom (- top (* delta 16)))
(defconstant left 66)    ; left side of grid
(defconstant right (+ left (* delta 16)))

; Functions in this library shown at the end of this example
(library "lib.pl")

(defvar ch " ")    ; A scratch 1 character string

(defun show-font (name fnt) 0  ; name is a symbol
```

```
;; start by drawing the header
(font 30 times)          ; selects 30 point times-roman
;; Converting id's to strings requires a buffer.
;; The blank string is used as the buffer.
(setf name (cvs name "                                        "))
(center (concat-str "Sample of Font " name) 720)
(setfont (scalefont fnt 18))
(center name 700)
(font 18 times)
(center "Samples in 18 point type" 680)
;; Now, the grid
(newpath)
(dotimes (i 17)
          (moveto left (+ bottom (* i delta)))
          (lineto right (+ bottom (* i delta)))
          (moveto (+ left (* i delta)) bottom)
          (lineto (+ left (* i delta)) top))
(setlinewidth 1)
(setlinecap 1)
(stroke)

;; Iterate through each of the 16 rows and columns
(dotimes (r 16)
   (dotimes (c 16)
     (let ((charcode (+ (* r 16) c))) ; char to print
       (setf (elt ch 0) charcode)
       (gpreserve   ; in library.  Adds gsave and grestore
        ;; move to box for char in grid
        (translate (+ (* delta c) left)
                   (- top (* delta (1+ r))))
        (font 5 times)  ; for lettering charcode
        (moveto 3 25)
        (show (cvs charcode "    "))  ; integer to string
        (when (/= (stringwidth ch) 0) ; in case no character
           (moveto 10 10)
           (setfont (scalefont fnt 18))
           (show ch)
           (moveto 0 10)  ; centering ticks
           (lineto 4 10)
```

```
                (moveto 10 0)
                (lineto 10 4)
                (setlinewidth 0.3)
                (stroke))))))
    (showpage))

;;; Here is the top level of this program.  Iterate through every font
;;; in the font dictionary and print it.

(dodict (n f (fontdirectory))
     (show-font n f))
```

This example uses the following functions from the library. The `center` function, shown previously, is not listed again.

```
;;; This is a sample Postscript library

;; gpreserve wraps a gsave and grestore around
;; a set of statements

(defmacro gpreserve (&rest statements)
   `(progn (gsave) ,@statements (grestore)))

(defun concat-str (s1 s2) 1
  (let* ((l1 (length s1))
         (l2 (length s2))
         (res (string (+ l1 l2))))
     (putinterval res 0 s1)
     (putinterval res l1 s2)
     res))

;;; To define a function in the compilation environment
;;; for use in macro expansion, the PLisp eval
;;; executes the defun at compile time.

(eval (defun get-font (style)
```

```
    (let ((family
           (cond
            ((member 'times style)
             '(|Times-Roman| |Times-Bold| |Times-Italic|
               |Times-BoldItalic|))
           )))  ; All other fonts deleted
      (if (member 'italic style)
          (setf family (cddr family)))
      (if (member 'bold style)
          (setf family (cdr family)))
      (car family))))

;;; This macro sets the current font.  italic and bold
;;; modifiers are allowed.

(defmacro font (size &rest style)
  `(setfont (scalefont (findfont ',(get-font style)) ,size)))

(defmacro fontname (&rest style)
  `',(get-font style))

(defmacro mkfont (size &rest style)
  `(scalefont (findfont ',(get-font style)) ,size))
```

4 The Lisp to PostScript Translation

This section describes the basic compilation strategy of PLisp; casual users may skip it.

The basic tasks of the compiler are:

- Syntax conversion.
- Name resolution.
- Macro expansion.
- Value analysis and stack management.

- Function call expansion.
- Management of the dictionary stack.
- Open coding of Lisp functions.

This section will discuss each of these subjects in more detail. When appropriate, implementation details of the compiler will be discussed.

4.1 Syntax conversion

Syntactic alteration is the most trivial part of the PLisp compiler. The major problem involves the treatment of case in identifiers, as mentioned previously. Internally, a simple mechanism has been provided to allow PLisp symbols to be renamed as they are being emitted to the output file. When an identifier has the `new-ps-name` property, the value of this property, a string, is substituted for the symbol during the output phase. The file `defps` contains initialization code which defines the initial set of renamings to the compiler. This feature is used to handle PostScript functions with more than one PLisp name as well as hide problems with case sensitivity.

Other syntactic operations are more straightforward. Inside strings, every special character, currently "(" and ")", is preceded by the PostScript escape, "\". Be warned that "\" has no meaning in Lisp but is special to PostScript, so that `"a\222b"` would be a string of 6 characters in the compiler but only 3 characters in the execution environment.

Numbers should go through PLisp unaltered. Numbers can be specified in any valid Lisp fashion, such as `#o100` for octal 100, but will always be translated to decimal on output. The integer character code which character constants compile into is computed in the Lisp environment and may not reflect the encoding in the PostScript environment. Thus, `#\X` may not be the same as `(elt "X" 0)` in cases in which two different character encodings are involved.

4.2 Name Resolution

One of the primary duties of the PLisp compiler is to resolve every symbol in the PLisp program. There are two aspects to this task: locating the appropriate definition of a symbol and determining the type of the symbol. The symbol types inside the PLisp compiler are:

- Primitive PostScript symbols.
- Lisp functions which compile into PostScript.
- Local and global variables.
- Defined constants.
- User functions and macros.

Symbols occur in two contexts: function and variable. The first element of a list is in function context, the rest are in variable context. Primitive PostScript symbols and Lisp function symbols may only occur in function context. These may not be redefined.

Before generating any code, PLisp parses the entire program and creates tables which map symbols onto definitions. A single table is always created for the main program; each library is associated with an additional table. The definition of a symbol within a table must be unique. In the code generation pass, every symbol is resolved as it is compiled.

When functions and variables in libraries are used to resolve undefined symbols, they are brought into the table shared with the main program. When undefined symbols in library functions are encountered, PLisp looks for a definition in the same library. This will occasionally lead to symbol name conflicts when this second library symbol is brought into the main program. Macros and defined constants are not present in the execution environment and do not have this problem.

4.3 Macro Expansion

PLisp uses the underlying Lisp macro expander. Since `apply` cannot be used with macros, the easiest way to expand macros is with the `macroexpand1` function. To use this, however, the macro must actually be defined in the Lisp environment. To avoid name clashes, the function is renamed with a prefix of `macro-`. This will be invisible to the user unless the macro looks at its own name when `&whole` is used. This also is important when debugging if the user traces macros. It is recommended that complex PLisp macros be debugged in the Lisp system to avoid causing errors during PLisp execution.

4.4 Function Call Expansion

Function call expansion involves turning a call to a function which is defined using the `&optional`, `&rest`, and `&key` constructs and replacing it with a call to a function requiring a fixed number of arguments. This involves putting in explicit initializers for `&optional` parameters, explicitly creating the array for `&rest`, and looking for the proper keywords for `&key`. After expansion, each variable bound by the function will have an explicit value. The run-time system supports only functions taking a fixed number of arguments. This expansion is generally trivial except when defining the environment for initializers. Initialization code must be able to refer to other variables previously bound in the parameter list. Since the actual binding is done after the function is invoked, such variables are recognized by the compiler and placed in the caller's environment under a temporary name.

4.5 Value Analysis and Stack Management

A fundamental property of PLisp code is that every expression manipulates the stack in a known manner. This is accomplished by defining exactly how many values each function consumes and returns on the stack. Each PostScript function is predefined to PLisp with these two numbers. Except in `multiple-value-call`, there will be exactly one argument for each value a function will use. When a particular argument to a function returns more than one value, all but the first value are deleted using a `del` operator. When an argument returns no values, a compile time error results. In contexts where no values are required, such as `progn`, all values returned by a function are deleted from the stack.

A few PostScript functions return a variable number of results. The last value returned from such a function is always a boolean. In PLisp, the number of values returned is made constant by adding extra NIL's (`false`'s at run time) when fewer than the maximum number of results are returned. Thus, `search` always returns 4 results, the last of which is always boolean. If this last result is `false`, two other `false`'s pad out the returned values and the original string is returned as the first value in the result. The `where, search, anchorsearch,` and `token` functions all are treated in this manner.

At run-time, user functions are similar to PostScript functions in that they consume and produce a fixed number of values. The compiler keeps the user honest about the values returned from a function by forcing an explicit declaration of this number. While occasionally painful to the programmer, this keeps the PLisp compiler from having to do some potentially difficult analysis.

When using functions such as `apply` which call an unknown function, the number of values returned by the unknown function cannot be determined at compile time. To prevent the stack state from becoming unknown, whenever an unknown function is invoked, the returned values are grouped into a single list. No check is made to ensure that the function being called consumes the proper number of parameters from the stack. Strange things may happen when `(apply #'moveto '(1 2 3))` is executed.

4.6 Management of the Dictionary Stack

The entire run-time environment is maintained using the dictionary stack. Definitions which place names in dictionaries are generated by the PLisp compiler in two situations. All function definitions and global variable initialization is done at startup. These values live in the bottommost user defined dictionary in the dictionary stack at all times. New sets of bindings are added to this stack by the various PLisp binding operators, such as `lambda` and `let`. The generated code uses `begin` and `end` to place new bindings in the environment. References to functions are compiled into simple PostScript identifiers which will be found in the lowest user dictionary. Variables are compiled into quoted names followed by the PostScript `load`. This prevents variables bound to compiled PostScript code from being executed inadvertently by the PostScript interpreter.

Except for the dictionary defining the outermost user block, every dictionary must be allocated by the PLisp program. The exact size needed for each such dictionary is determined by the compiler. When a new block is entered, the associated dictionary may be obtained in two different ways: staticly or dynamicly. Static allocation of dictionaries involves actually creating the dictionary at startup time and constantly re-using the same dictionary every time the associated block is entered. This is efficient in terms of time and space, but will not work if the block is recursively invoked. In this case, a new dictionary must be dynamically allocated every time the block is entered. The PLisp compiler attempts to determine at compile time whether or not each block may recursively invoke itself. This analysis is precise except when unknown functions are being called at run-time in `apply` or `funcall`. Any block containing either of these operators is assumed to be potentially recursive. The only problem with dynamically allocating dictionaries is that PostScript has no garbage collector and undue allocation may cause the program to exhaust memory.

The initialization code which starts any block involves binding every new variable in the block with `def`. After binding, all parameters are consumed from the stack. The requirement that all variables must be bound before any code is executed prevents `let*` from using a single block. Instead, each variable in `let*` is placed in a separate `let`. This is due to

the fact that PostScript dictionaries cannot be cleared out, possibly leading to unwanted shadowing when dictionaries are reused.

4.7 Open Coding of Lisp functions

Many Lisp functions are compiled in a very straightforward manner. The purpose of this section is to discuss those functions whose compilation is non-trivial.

The `function` function, also written as `#'`, compiles to executable code object or a symbol. When the argument to `function` is a symbol itself, then this form works as a `quote`. The only other possibility is that the parameter to `function` is a `lambda`. This causes the body to be compiled inline using the PostScript `{` and `}`. Since explicit compile time calls to such a function are not possible, these functions may not use of `&optional`, `&rest`, or `&key`. Environments are not captured by this function and lexical scoping is not a part of PLisp. Beware of the classic funarg problem.

The lisp math operators all attempt to fold constants. Feel free to use lots of defined constants in math expressions without worrying about slowing down the executing program. The PostScript math operators are compiled without any optimization. If constant math expressions involving operators other than `+, -, *,` and `/` are desired, use `defconstant`, in which any lisp code can be used in the initial value.

The `postscript` function is provided to allow the user to directly insert PostScript code into the program. Arguments may be placed on the stack and values returned on the stack. The actual PostScript code is just a list of symbols and numbers. You may use the lisp ' to generate the corresponding PostScript quote, /. For example, `(postscript (x y) 2 (exch))` would use the PostScript `exch` operator to swap x and y on the execution stack. The 2 is required to inform the compiler how many results are left on the stack.

5 PLisp Reference Manual

5.1 Running PLisp

PLisp has been installed on various systems under the name "plisp". Running PLisp puts you into Common Lisp with the PLisp interpreter loaded. To compile a PLisp program, you type `(ps "file")`. The name of the file being compiled must be `file.pl`, while the

resulting PostScript code will go into `file.ps`. The output is not spooled to a printer. On some systems, the file may be a command line argument to plisp, allowing you to type `plisp file` without having to use the lisp reader.

5.2 PostScript Functions in PLisp

Almost every PostScript function is available to the PLisp user. Only a few functions which deal with control flow or the stack are not directly available. Only those functions not directly available in PLisp will be mentioned here.

The stack operators, `pop`, `dup`, `exch`, `roll`, `index`, `clear`, `count`, `mark`, `cleartomark`, and `counttomark` are not available. PLisp controls the state of the stack at all times.

The `copy` operator should only be used to copy complex objects.

The `[` and `]` operators cannot be used directly. Either `#(` or `vector` should be used.

`aload` and `astore` are not available.

The `begin` and `end` operators will interfere with the PLisp environment. These should not be used.

Some control operators must be obtained using their PLisp counterparts.

All explicit matrix operators (those which use an explicit rather than the implicit matrix) must be selected using the suffix `-matrix`. For example, `transform` in plisp takes two arguments, while `transform-matrix` takes three arguments, the last of which should be a matrix. Also, `token` and `file-token` are treated as separate functions.

5.3 Lisp Functions Translated to PostScript

Many Common Lisp functions can be translated into PostScript. Differences between Common Lisp and PLisp semantics will be noted.

5.3.1 Binding and Control Flow

`and` All but last value must be boolean.

`apply`	The result is returned as a multiple value list. Be careful that the function is passed exactly the expected number of arguments. This is not checked at runtime.
`cond`	All tests must be boolean values. Each alternative must return the same number of values.
`do`	
`do*`	
`dodict`	Not Common Lisp. Like `dolist`, except two variables are bound: the key and the value.
`dolist`	The variable is not bound during the evaluation of the result.
`dotimes`	The variable is not bound during the evaluation of the result.
`funcall`	The result is returned as a multiple value list. Be careful that the function is passed exactly the expected number of arguments. This is not checked at runtime.
`function`	No lexical scoping. An anonymous lambda may not use `&optional`, `&key`, or `&rest`.
`if`	See `cond`.
`let`	A list of variables may appear in place of a single variable to obtain a multiple-value-bind effect.
`let*`	As above.
`map`	the result type must be NIL or an array (list and vector also work). This must be a constant. Only one sequence argument allowed.
`mapcar`	If the mapped function returns multiple values, all values will be concatenated into the result. The function being mapped is not checked to ensure it consumes exactly one value. Only one sequence argument allowed.
`or`	All values must be boolean.
`prog1`	Propagates multiple values. Really `multiple-value-prog1`
`progn`	
`psetq`	No value is returned.
`setf`	No value is returned. Only `values, elt` and `get` are setfable. Implied `values`, as in `setq`
`setq`	No value is returned. List of variables has multiple-value effect.

`unless`

`when`

`while` Not a common lisp function.

5.3.2 Multiple Values

`multiple-value-bind`

`multiple-value-call`

`multiple-value-list`

`values`

5.3.3 Math

`+`

`-`

`*`

`/`

= Also works on non-numerics.

/ = Also works on non-numerics. Only two arguments allowed.

>

<

>=

>=

`1+`

`1-`

`decf`

`incf`

5.3.4 Predicates

`arrayp`

`dictp` Not a Common Lisp function.

`floatp`

`fontp` Not a Common Lisp function.

`integerp`

`numberp`

`stringp`

`symbolp`

`vectorp`

5.3.5 Structures

`elt` Synonym for get.

`vector` Makes a PostScript array.

5.3.6 Other Functions

`postscript` Not a Common Lisp function. Called by (`postscript` *arglist number-results code*). The arglist is a list or arguments to be initially placed on the stack, number-results is the number of values left on the stack (assuming the arguments are consumed), and code is a list of postscript operators. Use the lisp quote (') to get a postscript quote (/).

`quote`

`symbol-value`

5.4 Top Level Forms

These special forms may be found only at the top level.

`defconstant` Initial value is a Lisp expression.

`defun`	Number of values returned must be supplied.
`defmacro`	
`defvar`	
`eval`	The argument to eval is evaluated by the Lisp evaluator at compile time.
`library`	Not a Common Lisp function. The file name must be a string.
`load`	The file name must be a string.

5.5 PLisp Error Messages

Most PLisp error messages are reasonably self explanitory. They all include as much of the surrounding context as possible to help pinpoint the error. Only the most common or cryptic messages will be listed here.

Attempt to redefine symbol
The symbol being defined by `defun`, `defvar`, `defconstant`, or `defmacro` has already been defined.

Can't interpret the following as a function
The argument to `#'` was not an identifier or a lambda expression.

Error in argument list
The argument list in a `defun` could not be parsed according to Common Lisp syntax.

Identifier required
An identifier was not found in a context in which a variable name is required.

Incompatible forms
The alternatives in an `if` or `cond` do not all return the same number of values.

Keyword not valid here
A keyword was encountered in a function call which is not declared by the corresponding function.

Library file must only have definitions
Any form encountered in a library file which is not a top level form causes this error.

No argument for keyword
A value did not follow a keyword in a function call.

PostScript functions must declare values returned
This common error occurs when a `defun` does not have the number of values returned immediately after the argument list.

Symbol can't be used as a variable
A symbol which was defined by `defun`, `defmacro`, or `defconstant` or which is a PostScript primitive was used as a variable.

Symbol required as function
An attempt was made to compile a form which did not have a function name at the front of the form. Note that PLisp does not allow `lambda` to be used in the car of a form.

Undefined function
A definition for the function indicated was not found in a `defun` or `defmacro` and it is not a known PostScript function. Both the main program and all libraries are searched.

Value required
A function which returns no values was used in a context in which a value is required.

Wrong number of arguments
The number of arguments to a function does not match the number expected by the corresponding `defun` or PostScript primitive.

Wrong number of values
In explicit multiple value contexts, the number of values expected does not match the number of values encountered.

Wrong number of values returned
This occurs when the number of values returned by the function body do not match the number of values indicated in the `defun`. This can usually be corrected using `values`.

PLisp programs may also generate runtime errors. The treatment of these errors depends on the PostScript system being used. PostScript allows error handlers to be redefined, thus providing the capability for a PLisp program to deal with its own errors. The dictionary stack will contain the complete environment and is probably the best source of information regarding the program state when the error occured. A fancy runtime error handler is part of the set of future PLisp improvements.

References

[1] Guy L. Steele, Jr. *Common Lisp, the Language* Digital Press, 1984

[2] Adobe Systems Incorporated *The PostScript Cookbook and Tutorial* Addison-Wesley, 1985.

[3] Adobe Systems Incorporated *The PostScript Language Reference Manual* Addison-Wesley, 1985.