# P, AN ALTERNATIVE SYNTAX FOR POSTSCRIPT: USER'S MANUAL

*G. C. Ewing*

*Technical Report COSC 2/88*

# P: An Alternative Syntax for POSTSCRIPT

## 1. Introduction

The POSTSCRIPT[1,2] graphics programming language was devised by Adobe Systems as a device-independent language for communicating with laser printers and other high-resolution graphical output devices. It was intended mainly as a standard intermediate form to be generated by other programs such as word processors and desktop publishing systems. POSTSCRIPT was therefore designed with a very simple, postfix syntax that is easy for programs to generate and parse, but not so easy for people to write and read.

POSTSCRIPT has since been adopted by Sun Microsystems as the basis of their NeWS[3] window server. The programmability of the NeWS server encourages the application programmer to write significant amounts of POSTSCRIPT code for downloading into the server. While doing this, the difficulties of writing large POSTSCRIPT programs by hand become apparent.

This document introduces the basic features of POSTSCRIPT, discusses some of its shortcomings as a programming language for people, and describes a new language, P, which the author has devised to overcome some of these shortcomings.

## 2. POSTSCRIPT

POSTSCRIPT can be regarded in two ways, as a programming language and as a graphics language. These two aspects are complementary and result in a language of great power and expressiveness for describing graphical images. Here, the programming aspects will be discussed first, followed by the graphics aspects.

### 2.1 POSTSCRIPT objects

POSTSCRIPT programs manipulate a number of different types of objects, some of which are familiar from other programming languages, and others which are unique to POSTSCRIPT. The most important types are:

- Numbers
- Strings
- Booleans
- Names

  A name is similar to an identifier in other programming languages.

- Arrays

  An array is constructed by enclosing a list of elements in square brackets.

- Procedures

- Operators (built-in procedures)

- Dictionaries

  A dictionary is an associative table of key/value pairs. Among other things, dictionaries are used to provide binding environments for variables.

All objects also have certain *attributes,* one of which is the *literal/executable* attribute. When a literal object is encountered by the interpreter, it is pushed onto the operand stack, whereas an executable object is treated specially according to its type.

Two types for which the executable attribute is important are *names* and *arrays,* which have different forms depending on this attribute. A literal name is preceded by a slash (which is not part of the name). Procedures are implemented as *executable arrays*, which are distinguished from literal arrays by their being enclosed in curly braces instead of square brackets.

## 2.2 Stacks

The PostScript interpreter maintains four stacks: the *operand stack*, the *dictionary stack*, the *execution stack* and the *graphics state stack.*

The operand stack is used to hold operands and results of PostScript operators. When a literal object is read by the scanner, it is pushed onto the operand stack. When an operator is read, the appropriate number of operands are popped from the operand stack, the operator is executed, and the result is pushed back onto the operand stack.

For example, the fragment

```
3 5 add
```

results in 8 being left on the operand stack.

The dictionary stack holds only dictionaries and establishes the current "scope". When an executable name is read by the interpreter, it is searched for in the dictionary stack, starting with the topmost dictionary. If a matching key is found, its corresponding value is examined. If it is a literal object, it is pushed onto the operand stack. If it is executable, it is executed.

Various operators are provided for managing dictionaries and the dictionary stack. The **dict** operator creates a new dictionary of a given maximum size. The **begin** operator takes a dictionary from the operand stack and pushes it onto the dictionary stack. The **def** operator takes two values from the operand stack, a key and a value, and makes a new entry in the topmost dictionary of the

dictionary stack. The **end** operator pops a dictionary from the dictionary stack and discards it.

The following example uses these operators to illustrate how the dictionary stack can be used to hold local variables. (Everything following a '%' on a line is a comment.)

```
3 dict begin                % push a new dictionary
    /x 6 def                % define x as 6
    /y 7 def                % define y as 7
    /answer x y mul def     % multiply x by y and store in
                            % answer
    end                     % pop dictionary
```

The dictionary stack initially contains two dictionaries, **systemdict** and **usredict**. **systemdict** contains the definitions of all the builtin POSTSCRIPT operators, and **userdict** is provided for the user to make his own definitions in.

The execution stack is used to keep track of procedures currently being executed, and is similar to a return address stack in other language implementations.

The graphics state stack will be discussed later.

### 2.3 Defining procedures

The following code fragment illustrates how a procedure is defined and used.

```
/average {add 2 div} def
40 60 average
```

The first line makes a definition in `userdict` whose key is the name `average` and whose value is the executable array '`{add 2 div}`'. The second line illustrates how this procedure is called.

### 2.4 Control structures

Control structures are provided by operators which take procedures as operands. For example, the fragment

```
a b gt {a} {b} ifelse
```

finds the maximum of the values of `a` and `b`. The values of `a` and `b` are pushed onto the operand stack. The **gt** operator produces a boolean which is `true` if its first operand is greater than the second. The **ifelse** operator takes three operands, a boolean and two procedures. If the boolean is `true` the first procedure is executed, otherwise the second is executed.

Other control structure operators provided include:

- **if**
- **for**

  Iterates over a range of integers.
- **forall**

  Iterates over the elements of a string, array or dictionary.
- **repeat**

  Executes a procedure a given number of times.
- **loop, exit**

  Provide a general iteration mechanism.

### 2.5 Graphics

The POSTSCRIPT imaging model is based on a stencil-and-paint metaphor. Images are built up on a page by applying *paint* through *stencils*. The paint can be coloured, black, white or any shade of grey. The stencils are arbitrary shapes defined in an abstract coordinate system, and may be geometrical shapes, character outlines, halftone renditions of photographs, or any other shape.

There is an implicit *current page* that is used to accumulate the marks made by *painting operators*. The current page starts off completely white, and execution of painting operators places marks on the page. Painting is done as if the paint were opaque, so that each new mark completely obscures any marks that may have been underneath. In the case of a printer, nothing is actually printed until the **showpage** operator is executed, at which time the contents of the current page is printed.

Some of the painting operators use the *current path* to define a stencil through which to apply paint. A path is a sequence of lines and curves (possibly disjoint) which is created using the *path construction operators*. Other painting operators provide facilities for painting text and sampled images.

The POSTSCRIPT interpreter maintains a *graphics state* that holds various items of information used by the graphics operators, such as the *current point,* the*current transformation matrix* which determines the coordinate system, and the current *colour, path, line width, clipping path* and *font.*

The current graphics state can be saved on the *graphics state stack* using the **gsave** operator and restored later with the **grestore** operator. This is very convenient, for it allows a procedure to set up its own graphics environment by modifying the current graphics state, perform some drawing, and then restore the state needed by the calling procedure.

### 2.6 Path construction and painting operators

The **newpath** operator clears the current path in preparation for creating a new one. Lines and curves can be added to the path with operators which construct straight lines, circular arcs and Bézier spline curves. **closepath** can be used to connect the last point of a path to the first point with a straight line.

Once a path has been constructed, the **stroke** operator can be used to draw a line along the path, or the **fill** operator to fill it in.

### 2.7 Transformation operators

Various operators are provided for modifying the current coordinate system by operating on the current transformation matrix. Simple transformations are provided by **translate, rotate** and **scale**, or a general linear transformation can be performed using **transform**.

### 2.8 Text

The **show** operator draws a string of characters starting at the current point using the current font. A font is a specially formatted dictionary which contains all the information necessary to draw characters of a particular size and typeface. The character shapes can be specified in a variety of ways, including arbitrary POSTSCRIPT procedures. The full power of the POSTSCRIPT graphics operators is thus available for drawing text.

### 2.9 NeWS extensions

Described below are some of the extensions to POSTSCRIPT that have been made by Sun Microsystems in order to support the needs of a window system and to allow multiple client programs to use the server at once.

- Canvases

  In place of the "current page" concept, NeWS has a new type of object called a *canvas*. A canvas is a region of the display surface of arbitrary shape on which drawing can be performed. Canvases can be moved around and may overlap. An off-screen bitmap may optionally be kept to speed up redrawing of portions of the canvas which are uncovered by moving another canvas. Canvases thus provide the basis for construction of overlapping windows, pop-up menus, and other user interface components.

- Processes

  The NeWS server supports multiple lightweight processes so that each client can have one or more POSTSCRIPT processes associated with it. If desired, the client can create separate processes to look after different user interface components, which can simplify the code required to deal with each component.

- Events

  User input events (keystrokes, mouse clicks, etc.) generate *event* objects which are distributed to processes. A process can express *interest* in an event or class of events in a number of ways, such as by type and which canvas it occurred on (in the case of mouse events). A process can also post user-defined events to other processes, thus providing a means of interprocess communication and synchronisation.

- Sockets

  Facilities are provided for accessing the socket interprocess communication facilities of BSD Unix, to allow the server process to communicate with client Unix processes.

## 3. POSTSCRIPT problems

The main difficulty encountered in writing POSTSCRIPT programs by hand is the problem of visualising and keeping track of the state of the stack over long sequences of operations. This is made worse by the fact that there is nothing to ensure that the correct number of parameters is passed to a procedure, or that the stack is left in the correct state on return.

Consider the problem of writing a procedure to find the roots of a quadratic equation. It will take the three coefficients as parameters and return a two-element array containing the roots. (For simplicity it will be assumed that the roots are real.) In pseudo-code the algorithm will be:

```
procedure qroots(a,b,c)
        local d,r1,r2
        d := sqrt(b*b-4*a*c)
        r1 := (-b+d)/(2*a)
        r2 := (-b-d)/(2*a)
        return [r1,r2]
```

Providing named parameters and local variables in POSTSCRIPT is somewhat involved, as it involves pushing a new dictionary onto the dictionary stack, popping the parameters (in reverse order) and storing them in the dictionary, and making definitions for any local variables. A direct translation of the above algorithm into POSTSCRIPT would be:

```
/qroots {                 % a b c => [root1 root2]
        6 dict begin
        /c exch def
        /b exch def
        /a exch def
        /d b b mul 4 a c mul mul sub sqrt def
        /r1 b neg d add 2 a mul div def
        /r2 b neg d sub 2 a mul div def
        [r1 r2]
        } def
```

Because of the tedious procedure required to declare named parameters and local variables, and the run-time overhead incurred, it is tempting to dispense

with these, simply leaving parameters and local variables on the stack and using **index** to reference them. The **index** operator takes an integer parameter, counts down the stack that number of items, and pushes a copy of the item found there.

Using this technique, qroots can be rewritten in a more efficient, albeit less readable, style as follows:

```
/qroots {                              % a b c => [root1 root2]
        1 index 2 index mul
                4 5 index 3 index
                mul mul sub  sqrt      % a b c √(b*b-4*a*c)
        [2 index neg 1 index add
                2 4 index mul div      % a b c √(b*b-4*a*c) [ r1
        4 index neg 3 index sub
                2 6 index mul div]     % a b c √(b*b-4*a*c) [r1 r2]
        5 -1 roll                      % [r1 r2] a b c √(b*b-4*a*c)
        4 {pop} repeat                 % [r1 r2]
        } def
```

By now, the underlying structure and purpose has been almost completely obfuscated by the mechanics of manipulating the stack. To maintain any degree of readability it is necessary to insert frequent comments indicating the state of the stack at each stage. Nevertheless, this example is typical of the coding style frequently found in POSTSCRIPT code.

The scope for error, even in this small procedure, is obvious. Moreover, an error which corrupts the stack will often not show up immediately; execution may continue for some time until terminated by a typecheck or stack underflow. By then, much information pertaining to the problem will have been lost, making such errors difficult and time-consuming to diagnose. The problem is compounded by the lack of parameter checking in procedure calls; a procedure will often fail because of an error in some deeply-nested procedure call, which, in the source, is far removed from the offending piece of code.

## 4. Solutions

The problems of programming in POSTSCRIPT have much in common with those of assembly language. The assembly language programmer typically has a stack which can be manipulated with great freedom, and a set of very loosely-typed operators and data structures. This freedom both permits great efficiency and allows tremendous room for obscure errors.

Very effective solutions to the problems of assembly-language programming have been in use for some time in the form of high-level programming languages. Clearly a higher-level language was needed that could be translated easily into POSTSCRIPT.

### 4.1 P - an alternative syntax

Consideration was given to using an existing high-level language, such as C or Pascal, for this purpose. However, each of these languages defines its own set

of abstractions, which cannot always be mapped easily or efficiently onto those of POSTSCRIPT. The intention was not to supplant the control and data structures of POSTSCRIPT, but simply to provide an alternative means of accessing them which is easier for people to deal with.

To that end, a new language was designed. In accordance with the Unix tradition of naming things as concisely as possible, it was christened **P**. It was designed chiefly for use by programmers writing applications for NeWS, although it could be used in any situation requiring hand-writing of POSTSCRIPT code. It is implemented by a translator that converts P source into POSTSCRIPT source. This can then be embedded in a NeWS application program via Sun's *cps* program, which is used to create interfaces between C and POSTSCRIPT.

The control and data structures of P are the same as those of POSTSCRIPT, but the syntax is more like that of languages such as Algol, C and Pascal. Some of the main features of the P syntax are listed below; for a complete description of P, see *The P User's Manual* (Appendix).

- Infix notation for arithmetic, relational and assignment operators:

  ```
  a := b*(c+d)
  ```

- Conventional procedure calling syntax:

  ```
  grobulate(f,g)
  ```

- Conventional array indexing notation (also applicable to strings and dictionaries):

  ```
  a[i] := b[j]
  ```

- Means of declaring parameters and local variables:

  ```
  grobulate ::= {|a,b|
      local c,d;
      ...
      }
  ```

- More conventional control structure syntax, for example:

  ```
  if a>b then {c:=a} else {c:=b}
  ```

- Object-oriented programming facilities, with a message-passing syntax similar to Simula:

  ```
  window.set_title("Hello world")
  ```

As a complete example of a P procedure, here is a P version of the `qroots` procedure presented above.

```
qroots ::= {|a,b,c|          /* Parameters */
    local d,r1,r2;           /* Local variables */
    d := sqrt(b*b-4*a*c);
    r1 := (-b+d)/(2*a);
    r2 := (-b-d)/(2*a);
    [r1,r2]                  /* Return value */
    }
```

The gain in readability over both of the POSTSCRIPT versions should be evident. Although the current implementation of P does not prevent the programmer from misusing the stack or passing incorrect parameters to a procedure, the improved syntax makes it much less likely that this will be done accidentally. In short, it makes it easy to write correct code and more difficult to write incorrect code.

### 4.2 The present

Using the parser-generating tools *lex* and *yacc*, implementation of the P translator took only a few days, mainly due to the very straightforward mapping between P constructs and POSTSCRIPT constructs. Since then, a considerable amount of code has been written in P, including an experimental user interface toolbox for NeWS. This experience has indicated that considerable gains in productivity can be achieved over writing directly in POSTSCRIPT.

### 4.3 The future

Future implementations of P may provide more protection against programmer errors, although, due to the run-time binding of procedure calls, this will probably incur a run-time overhead. A suitable compromise would be to make this a translator option which could be turned off once a program had been debugged.

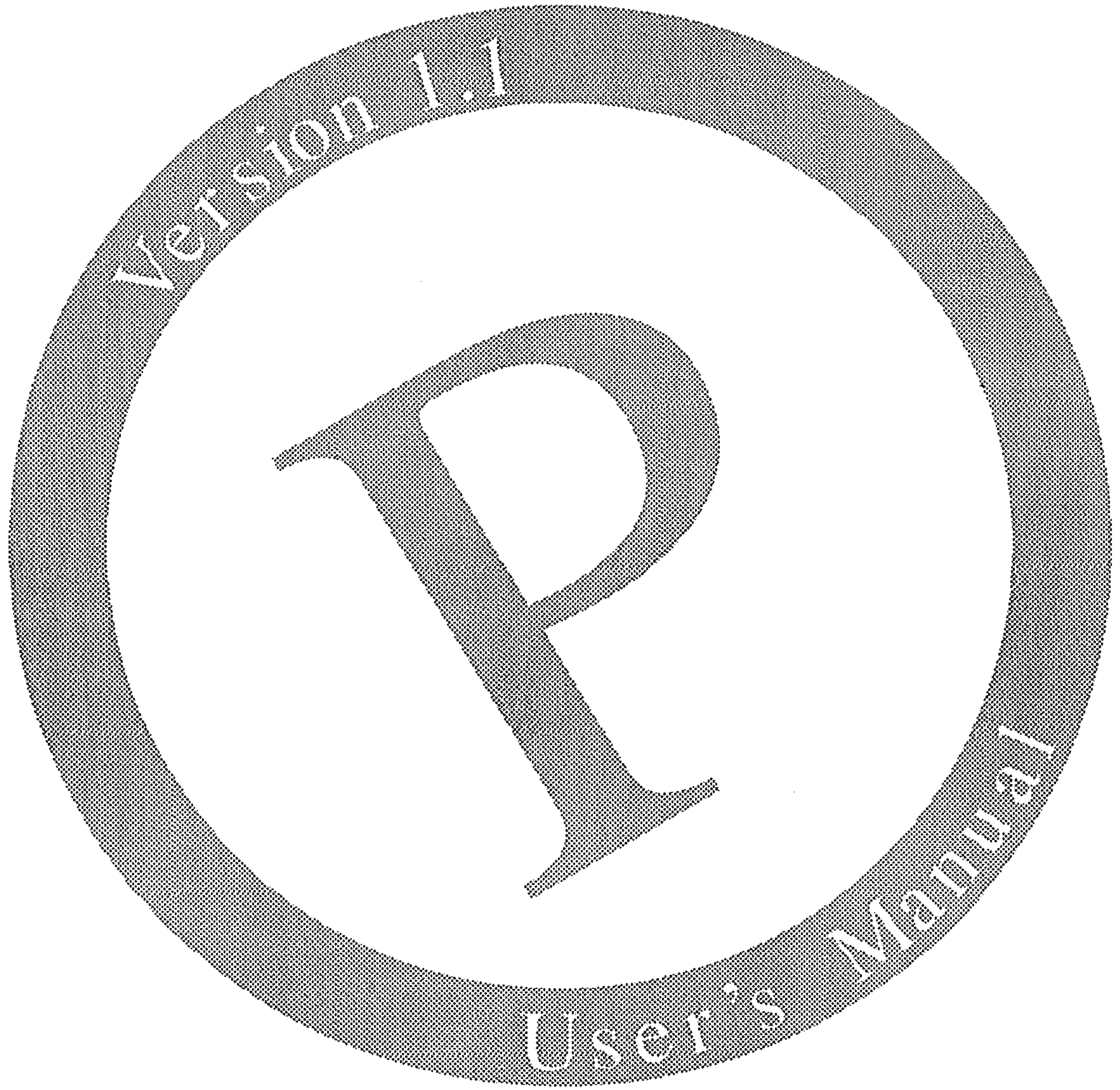
Also planned is support for a run-time source-level debugging environment allowing code to be traced, breakpoints inserted and variables examined during execution. Provision of such a facility is more difficult for POSTSCRIPT, since it is hard to associate parts of an executing POSTSCRIPT program with places in the text of the source code. The P translator should be able to provide a considerable amount of structural information to a debugging system.

REFERENCES

1. Adobe Systems Incorporated, *POSTSCRIPT® Language Tutorial and Cookbook,* Addison-Wesley 1985.

2. Adobe Systems Incorporated, *POSTSCRIPT® Language Reference Manual,* Addison-Wesley 1985.

3. Sun Microsystems, Inc., *NeWS™ Manual.*


Version 1.1
P
User's Manual

# P User's Manual

Version 1.1

# Contents


| Section | Page |
|---|---|
| **1 What is P?** | **1** |
| **2 Using the Translator** | **1** |
| **3 Flavour of the Language** | **1** |
| 3.1 Basic components | 2 |
| 3.2 Assignment operations | 2 |
| 3.3 Expressions | 2 |
| 3.4 Indexing | 3 |
| 3.5 Procedure calls | 3 |
| 3.6 Code blocks | 3 |
| 3.7 Control structures | 4 |
| 3.7.1 if statement | 4 |
| 3.7.2 repeat statement | 4 |
| 3.7.3 for statement | 4 |
| 3.7.4 loop statement | 5 |
| 3.7.5 case statement | 5 |
| 3.8 Object-oriented facilities | 5 |
| 3.8.1 Class definition | 5 |
| 3.8.2 Sending messages | 6 |
| 3.8.3 self and super | 6 |
| 3.8.4 Standard classes | 7 |
| **4 Advanced usage** | **8** |
| 4.1 Including raw PostScript | 8 |
| 4.2 Defeating auto-literals | 8 |
| **5 Translation rules** | **9** |
| **6 Reserved words** | **11** |
| **7 Syntax diagrams** | **12** |

# 1 What is P?

P is a programming language which provides the same facilites and semantics as POSTSCRIPT, but with a syntax which is easier for human beings to write, read and understand. P programs are converted to POSTSCRIPT by the P-to-POSTSCRIPT translator *pt*.

Since a working knowledge of POSTSCRIPT is necessary in order to make use of P, this document assumes that the reader is already familiar with POSTSCRIPT, and describes the semantics of P constructs in terms of their POSTSCRIPT equivalents.

# 2 Using the Translator

The usage of the *pt* command is:

`pt { `*filename*`.p | `*filename*`.ps } ... [ -o `*outfile*` ]`

File names ending in *.p* are assumed to be P source files and are translated into POSTSCRIPT. If no output file is specified, each *.p* file is translated separately and the result left in a corresponding *.ps* file.

The *-o* option can be used to specify an output file. In this case, the results of translating all the *.p* files are concatenated and placed in the output file. File names on the command line ending in *.ps* are assumed to contain POSTSCRIPT code and are incorporated unchanged into the output file. All files are concatenated in the order in which they are specified on the command line.

If an error occurs during translation, an error message and line number is given, along with the previous line and the offending line up to the error. Translation stops as soon as the first error is encountered.

# 3 Flavour of the Language

This section takes an informal look at the main features of P and how they correspond to constructs in POSTSCRIPT. Section 7 contains detailed descriptions of the syntax.

The following fonts are used in this section: `courier` for P fragments, helvetica for POSTSCRIPT fragments. The symbol » means "translates into".

## *3.1 Basic components*

The lexical components of a P program are identifiers, constants, literals and some special symbols and reserved words.

An identifier consists of a letter followed by zero or more letters or digits, and translates into a POSTSCRIPT name. A literal name is preceded by a single quote, for example, `'hello` translates to the POSTSCRIPT literal /hello.

Special characters can be incorporated into an identifier by surrounding it with a backquote and a single quote, for example, `` `Helvetica-Oblique' `` becomes the POSTSCRIPT name Helvetica-Oblique.

Numeric constants have the same syntax as in POSTSCRIPT. String constants, however, are surrounded by double quotes, so that the string constant `"Hello World"` corresponds to the POSTSCRIPT string (Hello World). All the POSTSCRIPT backslash sequences are valid. A double quote can be included in the string by preceding it with a backslash.

All punctuation characters are special characters. Certain identifiers are reserved words - see section 6 for a complete list.

## *3.2 Assignment operations*

There are two assignment operators. `:=` translates to a POSTSCRIPT store, while `::=` translates to a POSTSCRIPT def. For example,

```
greeting := "Hello World"      » /greeting (Hello World) store

TitleSize ::= 12               » /TitleSize 12 def
```

Note that the identifier on the left hand side is automatically made a literal.

## *3.3 Expressions*

The standard arithmetic, relational and boolean operations are available as infix or prefix operators. The operators, from lowest to highest priority (operators with the same priority are on the same line), are :

```
or
and
not
=  <>  <  >  <=  >=
+  -
*  /  %
```

Parentheses can be used in the usual way to group subexpressions.

## 3.4 Indexing

An element of an array or dictionary can be selected using an index expression:

```
a[i]                              » a i get
myDict['Aardvark]                 » myDict /Aardvark get
```

An index of the form m@n selects a subarray *n* elements long starting at *m* :

```
a[3@5]                            » a 3 5 getinterval
```

An index expression can also be used on the left of an assignment to assign to an element of an array or dictionary. Only the store form of assignment is valid in this case.

```
a[i] := 7                         » a i 7 put
a[3@] := [4,5,6]                  » a 3 [4 5 6] putinterval
```

The second form replaces a subsequence of the array.

## 3.5 Procedure calls

A procedure call consists of the name of a procedure optionally followed by a list of parameters enclosed in parentheses:

```
foo(a,b+2)                        » a b 2 add foo
```

The procedure can also be an expression which evaluates to a procedure, but in this case there must be at least an empty pair of parentheses to cause the procedure to be called:

```
myDict['Elephant]()               » myDict /Elephant get exec
```

## 3.6 Code blocks

A code block consists of a list of expressions, separated by semicolons, enclosed in curly braces. It corresponds to an executable array in POSTSCRIPT. However, a code block can also have parameters and local variables associated with it.

This facility is most commonly used when declaring a procedure. An example of a procedure declaration is:

```
foo ::= {| a,b |
     local x,y;
     x := a + b;
     y := a-b;
     x*y
     }
```

This creates a procedure foo with two parameters a and b, and two local variables x and y. The value returned by the procedure is the value of the last expression in the code block.

The POSTSCRIPT generated for this is:

```
/foo {
        4 dict begin
        /b exch def
        /a exch def
        /x null def
        /y null def
        /x a b add store
        /y a b sub store
        x y mul
        } def
```

From this it can be seen that a dictionary is automatically created to hold the parameters and local variables.

Initial values can be given for local variables, for instance:

```
foo ::= {|a,b|
        local x=a+b, y=a-b;
        x*y
        }
```

## 3.7 Control structures

All the standard POSTSCRIPT control structures have equivalents in P.

### 3.7.1 if statement

```
if expr then {body}
if expr then {body1} else {body2}
```

### 3.7.2 repeat statement

```
repeat expr times {body}
```

### 3.7.3 for statement

```
for expr to expr do {|i| body}
for expr to expr by expr do {|i| body}
forall array do {|e| body}
forall dictionary do {|key,val| body}
pathforall
        moves {|x,y| body1}
        lines {|x,y| body2}
        curves {|x1,y1,x2,y2,x3,y3| body3}
        closes {body4}
```

Note the technique of using a code block with parameters to access the values left on the stack by the control structure.

### *3.7.4 loop statement*

```
loop {body}
exit
```

### *3.7.5 case statement*

```
case expr of {
     label1 : {body1};
     label2, label3 : {body2};
     default: {body3}
     }
```

Note: the `case` statement makes use of a nonstandard extension to POSTSCRIPT provided by Sun Microsystems' NeWS window server.

## *3.8 Object-oriented facilities*

A facility is provided for object-oriented programming after the fashion of Smalltalk and Simula.There are may sources which the reader can consult for a detailed discussion of the concepts involved in object-oriented programming; a brief overview is presented here of how these concepts are embodied in P.

Object-oriented programming revolves around *objects* which can be send *messages*. Each object has a set of *instance variables* which are private to the object, and a set of *methods*, or procedures, which are executed in response to messages. While executing, the method has access to the object's instance variables.

Each object is an *instance* of a *class* which defines the object's instance variables and methods. Each class has a *superclass*; it is said to be a *subclass* of its superclass. A class inherits all the instance variables and methods of its superclass, but inherited methods can be overridden by methods of the same name declared in the class.

A class also has a set of *class variables* and *class methods*. Whereas each instance of a class has its own copy of the instance variables, class variables are shared between all instances of the class. Class methods are invoked by passing messages to the class itself, and are useful for such things as creating new instances of the class and initialising them.

### *3.8.1 Class definition*

Here is an example of a class definition.

```
class Object.Platypus;
cvars
```

```
        numberOfLegs = 4;
    cmeths
        new ::= {| name |
            local p;
            p := super.new;
            p.setName(name);
            p
            };
    ivars
        name;
    imeths
        setName ::= {| n |
            myName := n;
            };
        whoAreYou ::= {
            moveto(10,10);
            show(myName);
            };
    endclass
```

This creates a class called `Platypus` which is a subclass of class `Object`. It has one class variable called `numberOfLegs` with an initial value of `4`. Since it is a class variable, it will be shared between all instances of `Platypus`.

There is a class method declared called `new` which will override the default one inherited from class `Object`. There is one instance variable called `name`; since no initial value is specified, it will default to `null`. There are two instance methods calles `setName` and `whoAreYou`.

### *3.8.2 Sending messages*

A message is sent using the notation

```
receiver.message(parameters)
```

So we can create a new Platypus from the example above by writing:

```
fred := Platypus.new("Fred")
```

Then we can send a message to the newly-created instance of Platypus:

```
fred.whoAreYou
```

### *3.8.3* `self` *and* `super`

There are two special names which can be used inside a method. `self` always refers to the object which received the message which caused the current method to be executed. `super` is a special name to which a message may be sent in order to invoke a method from the superclass of the class to which the current method belongs. This is used to gain access to methods which have been overridden. In the example above, the `new` method of class `Platypus` uses the `new` method of its

superclass, `Object`, to create a "raw" `Platypus`, which it then initialises by sending it the message `setName`.

Note that `self` is a real identifier which may be used in any context, whereas `super` is a pseudo-identifier whose only valid use is to be sent a message.

### 3.8.4 Standard classes

There are two predefined classes called `Object` and `Class`. Class `Object` is the ultimate superclass of all other classes. Class `Class` is the class to which all classes belong. Below are listed the methods provided by these classes.

Methods of class `Object`

| Method | Description |
|---|---|
| `new` | Creates an new instance of a class. |

Methods of class `Class`

| Method | Description |
|---|---|
| `new(`*superclass, instance_vars, methods*`)` | Creates a new class.<br>*superclass* = class which new class is to be a subclass of.<br>*instance_vars* = a dictionary of instance variables and initial values.<br>*methods* = a dictionary of methods. |
| `newWithMetaclass(`*superclass, class_vars, class_methods, instance_vars, instance_methods*`)` | Creates a new class with class vars and methods.<br>*Superclass* = class which new class is to be a subclass of.<br>*class_vars* = a dictionary of class variables and initial values.<br>*class_methods* = a dictionary of class methods.<br>*instance_vars* = a dictionary of instance variables and initial values.<br>*instance_methods* = a dictionary of instance methods. |

# 4 Advanced usage

This section describes some techniques which are rarely used but are sometimes useful.

## 4.1 Including raw PostScript

Occasionally a construct is required in PostScript which does not map easily onto P. An example is the where operator, which can return a different number of values depending on whether it succeeded.

P allows a list of expressions separated by semicolons to be enclosed in parentheses and used wherever an expression is allowed. The following example illustrates how this can be used to include a piece of "raw" PostScript. It examines the result of the where operator, and if it returned true, stores the result in a variable, otherwise it stores the value /NotFound. Note the use of `...' to include PostScript operators which would otherwise be reserved words in P.

```
fred := ('mary; where; `not'; {'NotFound}; `if')
```

## 4.2 Defeating auto-literals

In some places in P, an identifier automatically has a slash character prepended to it when it is translated to PostScript. These places are:

- When an identifier appears on the left hand side of an assignment
- When an identifier is used as a message name in a message send

Sometimes this is not desirable; for instance, it may be desired to look up the value of a variable and use the result as a name to assign to, i.e. an indirect assignment. This can be accomplished by enclosing the identifier in parentheses. Here is an example.

```
fred := 'hello            » /fred /hello store
(fred) := 'hello          » fred /hello store
```

This works because the slash is only inserted if the left hand side consists of a single identifier; any more complex expression is left unaltered. In the same way, an indirect message send can be performed:

```
fred.blarg                » /blarg fred send
fred.(blarg)              » blarg fred send
```

# 5 Translation rules

This section defines how each language construct is translated into POSTSCRIPT.

*Expression lists and sequences*

| | | |
|---|---|---|
| `expr1 ; expr2` | » | *expr1 expr2* |
| `expr1 , expr2` | » | *expr1 expr2* |

*Assignments*

| | | |
|---|---|---|
| `ident ::= expr` | » | /ident *expr* def |
| `expr1 ::= expr2` | » | *expr1 expr2* def |
| `ident := expr` | » | /ident *expr* store |
| `expr1 := expr2` | » | expr1 *expr2* store |
| `expr1 [expr2 ] := expr3` | » | *expr1 expr2 expr3* put |
| `expr1 [expr2 @] := expr3` | » | *expr1 expr2 expr3* putinterval |

*Unary operators*

| | | |
|---|---|---|
| `not expr` | » | *expr* not |
| `- expr` | » | *expr* neg |

*Binary operators*

| | | |
|---|---|---|
| `expr1 or expr2` | » | *expr1 expr2* or |
| `expr1 and expr2` | » | *expr1 expr2* and |
| `expr1 = expr2` | » | *expr1 expr2* eq |
| `expr1 <> expr2` | » | *expr1 expr2* ne |
| `expr1 < expr2` | » | *expr1 expr2* lt |
| `expr1 > expr2` | » | *expr1 expr2* gt |
| `expr1 <= expr2` | » | *expr1 expr2* le |
| `expr1 >= expr2` | » | *expr1 expr2* ge |
| `expr1 + expr2` | » | *expr1 expr2* add |
| `expr1 - expr2` | » | *expr1 expr2* sub |
| `expr1 * expr2` | » | *expr1 expr2* mul |
| `expr1 / expr2` | » | *expr1 expr2* div |
| `expr1 % expr2` | » | *expr1 expr2* mod |

*Procedure calls and message sends*

| | | |
|---|---|---|
| `ident(exprlist)` | » | *exprlist ident* |
| `expr(exprlist)` | » | *exprlist expr* exec |
| `expr.ident` | » | *expr /ident* send |
| `expr.ident(exprlist)` | » | *exprlist expr /ident* send |
| `expr1.expr2` | » | *expr1 expr2* send |
| `expr1.expr2(exprlist)` | » | *exprlist expr1 expr2* send |

*Elements*

| | | |
|---|---|---|
| *ident* | » | *ident* |
| '*ident* | » | /*ident* |
| "*string*" | » | (*string*) |
| *number* | » | *number* |
| (*exprseq*) | » | *exprseq* |
| [*exprlist*] | » | [*exprlist*] |
| expr1 [*expr2*] | » | *expr1 expr2* get |
| expr1 [*expr2*@*expr3*] | » | *expr1 expr2 expr3* getinterval |

*Blocks*

| | | |
|---|---|---|
| {*exprseq*} | » | {*exprseq*} |
| {\|*ident1*, *ident2*...\|<br>local *ident3* = *expr3*,<br>*ident4* = *expr4*...,<br>*identn* = *exprn*;<br>*exprseq*} | » | {*n* dict begin<br>/*ident1* exch def<br>/*ident2* exch def ...<br>/*ident3* *expr3* def<br>/*ident4* *expr4* def ...<br>/*identn* *exprn* def<br>*exprseq* end} |
| {... *label1*, *label2*: ...} | » | {... /*label1* /*label2* ...} |

*Control structures*

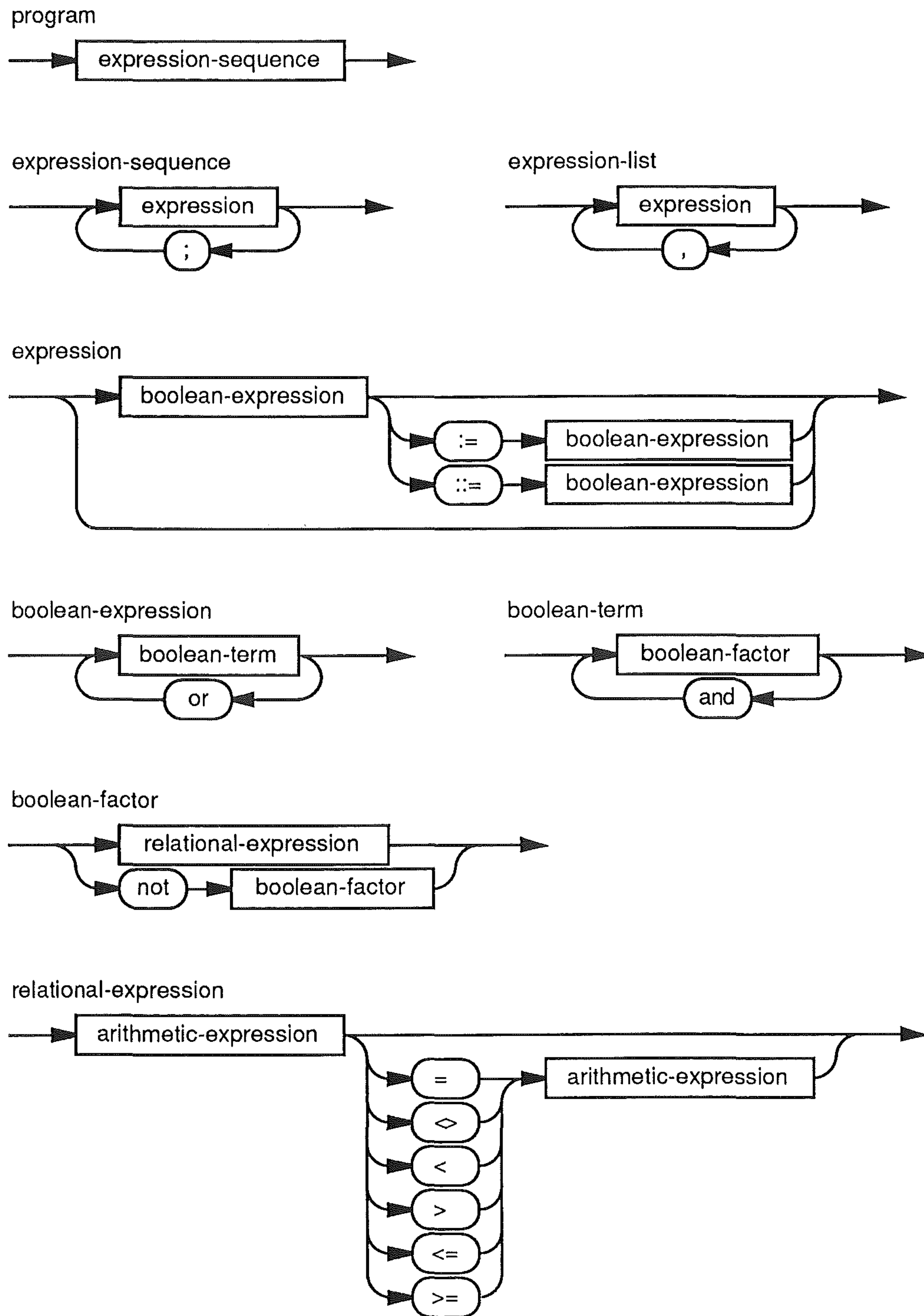
| | | |
|---|---|---|
| if *expr1* then *expr2* | » | *expr1 expr2* if |
| if *expr1* then *expr2*<br>else *expr3* | » | *expr1 expr2 expr3* ifelse |
| for *expr1* to *expr2* do<br>*expr3* | » | *expr1 expr2 expr3* for |
| for *expr1* to *expr2* by<br>*expr3* do *expr4* | » | *expr1 expr2 expr3 expr4* for |
| forall *expr1* do *expr2* | » | *expr1 expr2* forall |
| pathforall<br>moves *expr1*<br>lines *expr2*<br>curves *expr3*<br>closes *expr4* | » | *expr1 expr2 expr3 expr4*<br>pathforall |
| loop *expr* | » | *expr* loop |
| repeat *expr1* times *expr2* | » | *expr1 expr2* repeat |
| with *expr1* do *expr2* | » | *expr1* begin *expr2* end |
| case *expr1* of *expr2* | » | *expr1 expr2* case |

# 6 Reserved words

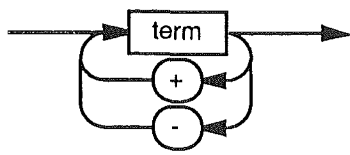
Here is a complete list of the reserved words in P.

```
and       curves    imeths    of          times
by        cvars     ivars     or          to
case      do        lines     pathforall  with
class     else      loop      repeat
closes    for       moves     super
cmeths    if        not       then
```

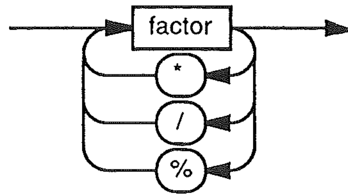
# 7 Syntax diagrams



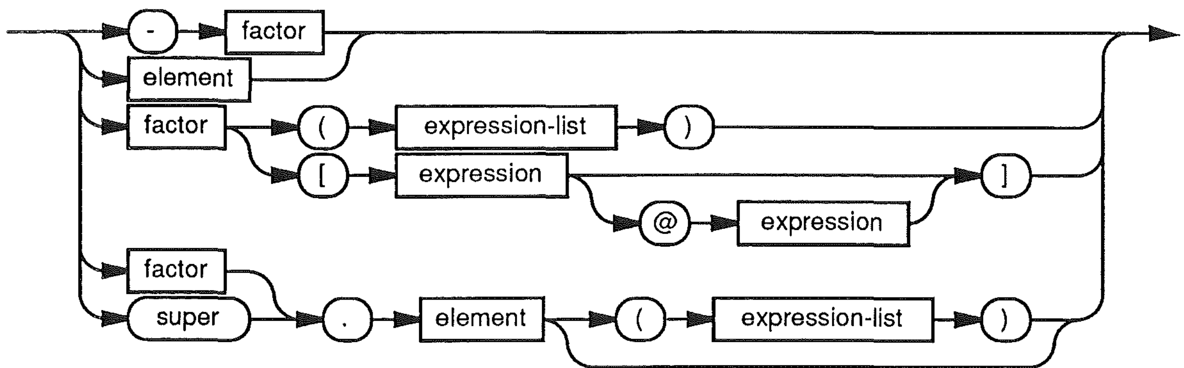
arithmetic-expression



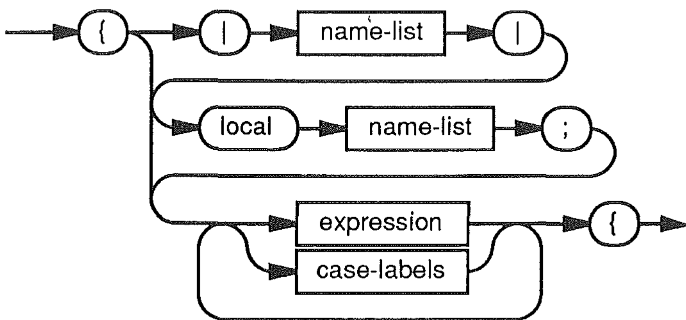

term



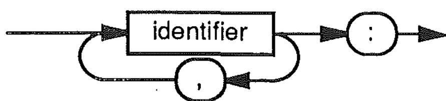

factor



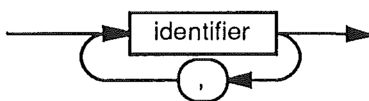

block



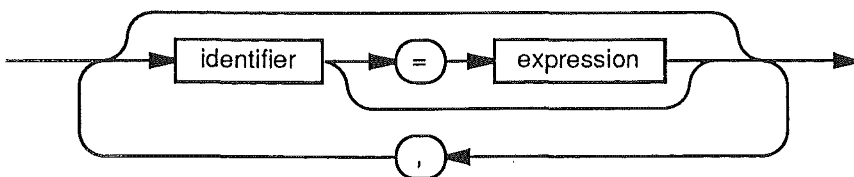

case-labels



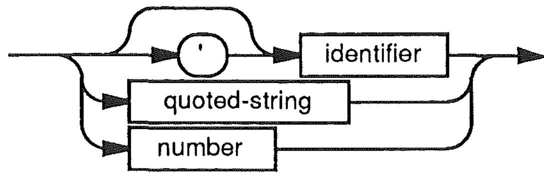

name-list



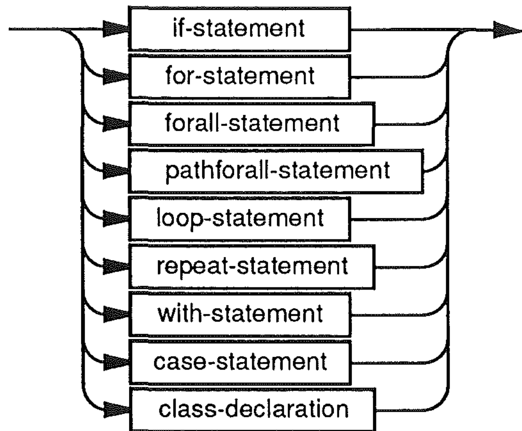

locals



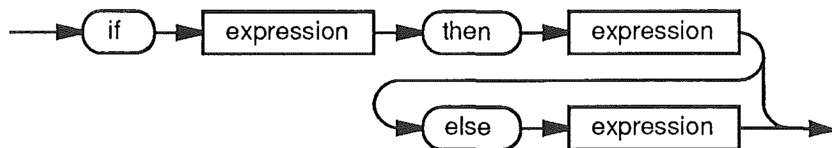
literal



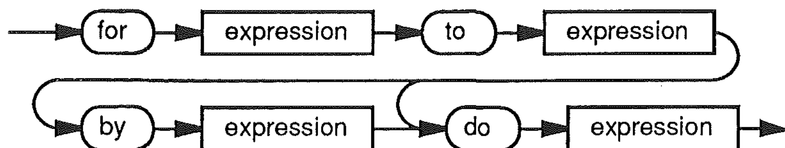

control-structure



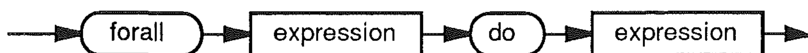

if-statement



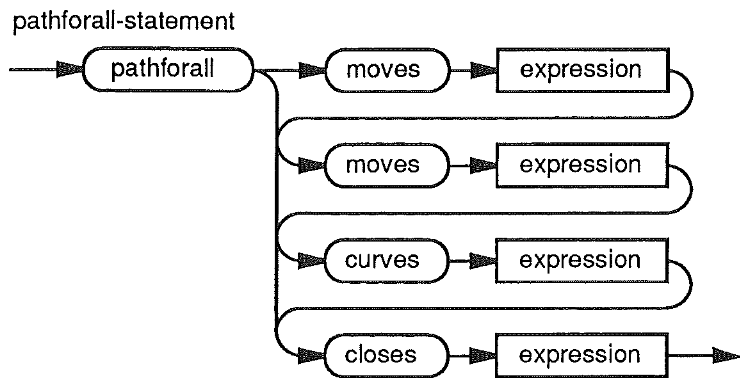

for-statement



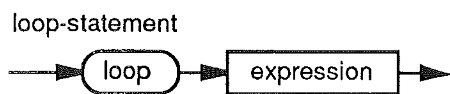

forall-statement



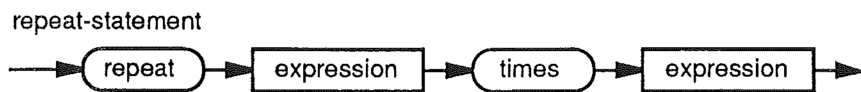
pathforall-statement



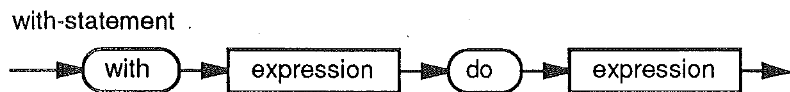

loop-statement



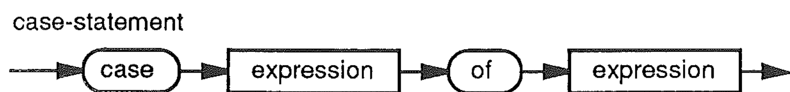

repeat-statement




with-statement




case-statement




class-declaration