P5

ISO 7185 Basic interpreter

Scott A. Franco

1 Introduction

This manual covers the Basic language interpreter that comes with the Pascal-P5 porting kit. The interpreter is written entirely in ISO 7185 Pascal, but contains extensions that are normally commented out to provide named file I/O.

The language of the basic interpreter was in common use in the late 1970's/early 1980's. When the first microprocessors were used widely to create hobby computers, the Basic language was popular as the programming language. Because the most popular previous implementation was DEC PDP-11 Basic, those Basic implementations often appeared as cut down versions of that Basic interpreter to enable porting of various programs from that machine. This includes implementations such as MBasic-80 by Microsoft, and others.

P5 Basic roughly implements MBasic-80, but a subset of that. The test material for it is the "Creative Computing" set of Basic language games that were published in the 1970s. Please see:

http://www.classicbasicgames.org/

Thus P5 Basic, for the most part, implements the set of features necessary to implement that set of programs. However, P5 Basic implements several features to enable a better programming style than those original Basics, including:

- Programs without leading line numbers.
- Full alphanumeric identifiers, including goto targets.
- Structured statements.
- Free form, multiline constructs.

This has been implemented in a manner backwards compatible to the base language.

2 Beta software warning

P5 Basic is beta software at this time (version < 1). It should not be considered reliable for production use.

3 Syntax notation used

The notation used to introduce language syntax elements is:

[option]	An optional item that may or may not exist.
repeated…	An item that may be repeated any number of times from 1 to n.
[option]…	An item that may be repeated any number of times from 0 to n.
a|b	Two alternatives, IE, a or b but not both.
'literal'	Literally, the characters in single quotes.
"string"	A constant string of characters.

keyword	A language keyword.
(group)	A group of items, usually for purposes of repeating.
(group)…	A repeated group, 1 to n times.

4 Format of source

A P5 basic program consists of a series of text lines. Each line may or may not have a leading number, which is ignored while the program is being executed. Lines can be blank.

```
10 print "hi"
20
30 end
```

Case is not significant except within strings.

4.1.1 Line entry

When lines are entered on the console, the "line oriented editor" mode is used. The format of a console line is:

[number] program text…

When a line is entered, and it has a leading number, that line is matched to the lines in the current program store. If the line number order is:

- Between existing lines: it is inserted between those lines.
- Equal to an existing line: the line is replaced.
- After any line in the program: the new line is added above the program.
- Before any line in the program: the new line is added before the program.

If a line number is entered alone, it signifies that any existing line is to be deleted. If a space or spaces exist after the number, then a blank line is assumed.

All leading spaces before the line number are removed. This is the only time the formatting of source lines is changed.

If no line number appears on the entered line, then the line is considered to be immediately executable.

4.1.2 File entry

If a source file is read, each line is treated as if it were directly entered on the console. However, if no line number appears, the line is simply appended to the current program. In this manner, it is possible to enter a program without line numbers.

When a source file is written, it is output as is, with out without a leading line number.

Thus directly saving and restoring program files allows free form programming.

4.1.3 Identifiers

Identifiers in P5 Basic are of the form:

[_,a-z,A-Z][_,a-z,A-Z]…

An identifier starts with any of _, a through z and A through Z, followed by any number of characters in _, a though z, A through Z, or 0 through 9. Note that this also means the legacy basic identifiers are valid labels, thus:

a

a1

a10

MyVariable

Are all valid identifiers.

Note that case is irrelevant in P5 Basic. Thus:

MyVar

And

Myvar

Are the same identifier.

4.1.4 Numeric constants

Numberic constants can be simple integer, or real:

1

1.2

1.2e5

.5

-0.5

Integer constants can also be input in hex, octal or binary:

$10	Hexadecimal 16
&10	Octal 8
%0101	Binary 5

4.1.5 String constants

String constants are characters between double quotes:

“hi there”

If a double quote is wanted within the string, a “quote image” is used, consisting of two back to back quotes:

“and he said “”what is going on”” there?”

4.1.6 Comments

P5 Pascal features both a comment statement, **rem**, as well as a comment character, '!'. The difference is, besides being shorter, ! is not a statement and can appear anywhere on the line. Like the **rem** statement, it ignores all further text on the line.

4.1.7 Internal format

P5 Basic reformats the source language into an internal, tolkenized format that is more efficient, both to store as well as to interpret. It tries hard to maintain the original format of the source. For example, the spacing of the original program is maintained. However, there is one change, and that is of numeric formats. Numbers are simplified, with leading zeros removed, and floating point constants are normalized.

5 Variables

P5 Basic allows for 7 different kinds of variables

x	Real (floating point)
x%	Integer
x$	String
x(n[,n]…)	Real array
x%(n,[n]…)	Integer array
x$(n,[n]…)	String array
x.y	Record

Each form of variable contains different data. Thus:

x

And

x$

Refer to two completely different variables.

5.1 Real

Real valued variables can take any value, with any fraction. They can be used for integer math, but take longer to calculate than integer based variables.

5.2 Integer

Integer valued variables always assume integer only values, no fractional part. They are generally faster and more compact than reals.

5.3 String

String valued variables take a series of characters and can be any length up to 250 characters.

5.4 Arrays

Each of real, integer and string can be expressed as an array, and that array can have any number of dimensions. For example:

x(10, 14, 12, 31, 42)

Is a valid array. Index values range from 0 to n, where n is any number between 0 and 1000.

Arrays are distinct from other variable types, but not from each other. Once an array with a given identifier is used with a given number of dimensions, the number of dimensions used to access it cannot change. Thus:

a(10)

Cannot later be used as:

a(10, 10)

Arrays can be dimensioned by the dim statement, but more typically the size and index characteristics of the array are determined automatically from its first encountered form.

5.4.1 Sparse arrays

In P5 Basic, arrays are automatically sparse. If the statements appear:

A$(1) = “hi”

A$(10) = “there”

The resulting array is not 10 units long, but 2. P5 Basic does not allocate array elements until they are used.

5.4.2 Function precedence over arrays

In Basic, array notation a(x) and function notation f(y) are inherently ambiguous with respect to each other. When a name is used to introduce a function, any array definition by that same name is essentially masked by the function definition. The previous contents of the array will not be usable.

Naming functions and arrays the same is to be avoided for this reason.

5.5 Records

A record is a collection of named fields, each of which can have a different type:

a.b = 1

a.b$ = “hi there”

a.b(5) = 62

a.c% = 7

There is no limit to the number of fields each record can have. Records can contain records:

a.b.c$ = “hello world”

is a record a, with field b which is a record, which contains field c.

However, in this version of P5 Basic, arrays of records are not possible.

5.6 Files

Separate from variables, P5 Basic can operate on a number of text files referenced by number, from 1 to n where n is a maximum of 100. **input** or **print** statements can be used with files. Two file numbers, 1 and 2 are reserved:

1	Standard input file.
2	Standard output file.

So the default file for input statements is #1, and the default file for print statements is #2.

Files are text only, and structured into lines. **input** statements read whole lines from a file. **print** statements write whole lines to a file.

6 Statements

6.1 Input statement

input [#file,] [prompt[;|,]] variable[,variable]

The input statement reads some number of variables from either a file or the standard input. The file number can be specified, or the default file #1 used (standard input). If a prompt string is specified, then that will be output to the standard output before the input is performed. This mode is not valid if a file number was specified (even if it is the standard input). The prompt string is a simple constant string. No string expression can be specified.

If no prompt string is not present, then "? " will be output as the standard prompt (question mark followed by space). The prompt is terminated by either ';' or ','. Which character terminates the prompt determines:

;	Specifies to output the standard prompt ("? ") *after* the prompt string is output.
,	Specifies to *not* output the standard prompt after the prompt string is output.

The input statement reads an entire line from the text file or standard input. If multiple variables appear, then the input line is separated into fields by the ',' character. For example:

input a, b, c

Given an input line "1, 2, 3" would give the results:

a = 1

b = 2

c = 3

String values are also delimited by ',' if more than one variable appears on the input statement. However, if only one string variable appears, the entire input line will be assigned to the given variable without regard to embedded ',' characters.

Note that input from a file is not available if named file handling is not enabled.

6.2 print statement

print [#file] ([**tab**(number)] [**using** "format"] [expression [,|;] [expression[,|;]])…

Print a series of expressions, of any type, integer, string or real, on one output line. The print statement starts with an optional file number, the a series of "print groups", containing **tab**, **using**, and then expressions, ending with a print format character.

If a file number is specified, then the output is redirected to that file. The default file is #2, the standard output.

Each group must be separated from the next by ',' or ';'. The separation of each printed field from the next is determined by the character used to separate the fields. The ',' character causes each field to be placed in 15 character spaces. The ';' character causes each field to be placed separated by 2 spaces. Thus:

```
print 1,2

 1        2

print 1;2

 1  2
```

If the print group starts with a **tab** expression, it will contain a numeric expression that evaluates to a tab location on the line, from 1 to n, where n is the character location on the line. The print location will be padded with that number of spaces to reach the tab position. Thus:

```
print tab(20) "hi"

                   hi
```

To allow more flexibility in output formatting the **using** facility is available. Using takes a string expression. The string specifies a series of characters to determine how the expression to be printed is laid out on the line. The following character meanings exist:

&	Print a string
$	Print '$' if at the left of the most significant digit in number, otherwise ' ' (space).
,	Divide digits in a number. Prints ',' if within significant digits of the number, otherwise ' ' (space).
#	Print digit in this position or space if not significant.
0	Print digit or zero if not significant.
-	Print sign if negative before most significant digit.

+	Print any sign before non-zero digits.
.	Place decimal point.
^	Print exponent.

Any other character than what appears above is simply printed in place. The idea of using is to present an "image" of what the number looks like in its field. Each character of the format indicates what happens in that digit position. Thus:

print using "The string is: &" "hi"

The string is: hi

```
print using "####" 5
   5

print using "0000" 5
0005

print using "$$$#" 5
  $5
```

The formatter starts by finding the decimal point in the format. This is either explicitly the '.' decimal point format character or the position to the right of any of the digit format characters. If the number is integer, it is considered to be a whole number with only 0's as the fraction.

Having found the decimal position, each number format character to the left of the decimal position is matched to a digit in the whole part of the number, and each number format character to the right of the decimal position is matched to a a digit in the fractional part of the number. Thus:

```
print using "###.###" 5
  5.000

print usign "###.###" 5.123
  5.123

print using "###,###.###" 5.123
      5.123

print using "###,###.###" 1465.123
  1,465.123

print using "000,000.###" 1465.123
001,465.123

print using "$$$,$$#.###" 1465.123
 $1,465.123

print using "---,--#.###" 1465.123
  1,465.123

print using "---,--#.###" -1465.123
```

```
 -1,465.123
```

It is an error if there are more than one '.' (decimal) format characters in the **using** string.

When real numbers are printed, the most space efficient format is used. Generally if the number can be represented without an exponent in 10 characters, it is printed that way. If it takes more than 10 characters, the exponent format is used.

To get the exact format you want, it is recommended that **print using** be utilized.

6.3 goto statement

goto [line number|label]

label:

The goto statement causes program execution to stop and then continue at a new location. If a line number is specified, then that is found and executed. If, instead an identifier appears, then the execution continues after the indicated label. It is an error if the line number or label is not found.

The ability to specify an identifier instead of a line number enables free format code with or without line numbers. To enable this, a label must be present in the code. It must appear first on the line before any statements. The line containing the label may or may not also have a line number on it.

6.4 on statement

on expression **goto** label|lineno [,label|lineno]…
on expression **gosub** label|lineno [,label|lineno]…

The **on** statement parses an expression and uses the value to select from a list of line numbers or labels. If the expression is 1, the first in list is matched. If 2, the second, etc. If no item in the list is matched, execution simply continues.

The target line may be a **goto** or **gosub**. If **goto**, the execution simply transfers to the target line (see the **goto** statement). If **gosub**, the target statement is executed, and a ensuing **return** statement will cause it to return to the next statement after the **on** (see **gosub** statement).

6.5 if statement

if condition **then** statement [**else** statement]

if condition **goto** [line|identifier]

if condition **then**

[else]

Endif

The if statement controls execution according to a given condition. It comes in several forms, and may either complete on a single line, or span many lines.

The condition must evaluate to 0 or non-zero. If the condition is zero, then the condition is false. If the condition is non-zero, then it is true.

The first form gives a statement to execute if the condition is true, optionally followed by a statement to execute if the condition is false. A side effect is that the rest of the line after the statement is executed along with the last active part of the **if** statement. Thus:

if 1 **then print** “hi”: **print** “how are you”

Prints:

hi

how are you

Because the last part of the if statement is the true clause. If the condition is not true, none of the rest of the line is executed.

If this if is executed:

if 1 **then print** “hi” **else print** “lo”: **print** “how are you”

Prints:

hi

And then terminates the line, because the else or false clause controls the rest of the line. The statement:

if 0 **then print** “hi” **else print** “lo”: **print** “how are you”

Prints:

lo

How are you

Bcause the else or false clause enables the rest of the line.

[note: I don’t think this is correct behavior according to MBasic-80]

The rules for continuation on the same line using **if** are a bit complex. The recommendation is to only use **if** for a single statement on the line.

The second form of if gives shorthand for gotos. If the condition is true, then the goto is executed (see the goto statement). The rest of the line is not executed.

In the final form, the **if** statement spans multiple lines. This form of **if** is more appropriate for free-form program code. The if statement multiline mode is activated if there is nothing on the line past the then keyword. When this occurs, the if is not terminated until **endif** is encountered. If an **else** is encountered and the condition is false, then that is executed. In both true and false cases, the **if** statement ends after the **endif** is encountered.

6.6 rem statement

rem some text…

The rem statement causes the rest of the line to be ignored, and execution continues with the next line. Thus:

```
rem this is a comment
```

Another way to introduce a comment is with the '!' character. This can be placed anywhere on the line, not just where a statement is expected:

```
print "hi there" ! do my print here
```

6.7 stop statement

stop

Causes the program to halt and enter command mode. In this version of P5, it is functionally identical to the **end** statement (see **end** statement).

6.8 run statement

run [filename]

Runs the current program or a program from a file. The variables, current tabbing, and the random number generator are cleared and reset, and the program runs from the first line in the program. If a filename is specified, then that program file is loaded and the current one completely cleared.

It is an error if a filename is specified and named file I/O is not enabled.

6.9 list statement

list [start line[,end line]]

Lists the current program to the console. If no starting or ending line is specified, then the entire program from start to finish is listed. If only a starting line is specified, then that line from the end of the program is listed. If both the start and end are specified, then the program is listed from the starting line to the ending line, inclusive.

To be able to specify starting and ending lines, the source lines indicated must have numbers.

6.10 new statement

new

Clears the program store and variables. Used to start editing on a new program.

6.11 let statement

[let] variable = expression

Assigns an expression to a variable. The variable assumes the value of the expression. The let keyword is optional. Examples:

```
Let a = 10.67
A$(10) = "hi"
a.b%(5) = 3
```

6.12 load statement

load filename

Loads a new program. The file by the given filename is loaded into the current program store. Any previous program is erased.

If is an error if named file I/O is not enabled.

6.13 save statement

save filename

Saves the current program to the given file by filename. The current program in store is unmodified.

If is an error if named file I/O is not enabled.

6.14 data statement

data [value[,value]]

Introduce data into the program. The data statement introduces a series of data values into the program:

```
data 1, "hi", 3.1414
```

Data statements can be placed anywhere in the program, and are simply no-ops when run. The values in the data statement are placed in variables by the **read** statement (see **read** statement). All of the data statements in the program are considered a list from start to finish. The **read** statement will read each in turn until there are no more data statements in the program.

6.15 read statement

read variable[,variable]…

Reads from the current **data** statement (see **data** statement) into the list of given variables. The data types must match in type, just as if an assignment were being made. Integers can be assigned to real, and vice versa, but strings cannot be read to integer or real, and integer or real cannot be read to a string.

The current data position to read from starts from the first **data** statement in the program and moves sequentially to the last data item on the last **data** statement in the program. The position of where data is to be read can be set with the **restore** statement (see the **restore** statement).

It is an error if there are no more data values in the program to read to the current variable.

6.16 restore statement

restore [linenumber|label]

Restores the data position used by the next **read** statement (see **read** statement) to either the beginning of the program, or to a specified line by number or label.

It is not an error to restore the data position if there are no **data** statements in the program. However, the next **read** statement will cause an error.

6.17 gosub statement

gosub linenumber|label

Transfers control to the line specified by the linenumber or label. The position of the program execution after the gosub is saved, and restored by a later return statement (see return statement). **gosub**s can be nested to any level.

Example:

```
10 gosub 30: print "done"
20 end
30 print "hi": return
hi
done
```

6.18 return statement

return

Returns to the program position just after the last **gosub** statement was executed (see **gosub** statement).

It is an error if there is no last **gosub** executed.

6.19 For/next statement

for variable = start expression **to** end expression [**step** expression]

for variable = expression [,expression]

next [variable]

Loops a code sequence with a series of values in a variable. The variable is assigned a series of values from the starting expression to the ending expression. The following statements are executed with that variable value until a **next** statement is encountered (see **next** statement). Then the loop is repeated with the variable incremented by either the default 1, or by the increment specified in the optional step expression.

The **step** can be any value, including negative and fractional. The loop will continue as long as the variable value is between the starting and ending expressions. The start and end expressions are evaluated only when the **for** loop is started.

Examples:

```
for i = 1 to 10: print i: next
 1
 2
 3
 4
 5
 6
 7
 8
 9
 10

for i = 1 to -10 step -0.45: print i: next i
 1
 .55
 .1
 -.35
 -.8
 -1.25
 -1.7
 -2.15
 -2.6
 -3.05
 -3.5
 -3.95
 -4.4
 -4.85
 -5.3
 -5.75
 -6.2
 -6.65
 -7.1
 -7.55
 -8
 -8.45
 -8.9
 -9.35
 -9.8
```

A second form of the **for** loop can be used called “sequence of values”. Instead of specifying how the variable is to be incremented, a series of values are given:

```
for i = 1, 3, 5, 2, 0: print i: next
 1
 3
 5
 2
 0
```

This form of the **for** loop also allows string variables to be iterated:

```
for i$ = "one", "two", "three", "four", "five": print i$: next
one
two
three
four
five
```

A **next** statement can be specified with or without a variable. With no variable, it terminates whatever for statement was last executed. The **for** statement is executed again, and that continues until the values of the **for** loop are satisfied. Then execution continues after the **next** statement.

If the **next** statement has a variable, then only the **for** statement that matches that variable is reexecuted. If the last **for** statement does not match, then all previous active for statements that do not match the variable are simply canceled. This allows for special situations:

```
for y := 1 to 10
    for x = 1 to 10
        print x, y
        if x > 5 goto exit
    next x
    exit:
next y
```

Here the inner loop is terminated early. If the **next** statements were anonymous (no variable specified), then the last next would simply continue the inner loop. Instead, the inner loop is cancelled and the outer loop continues.

for statements can be nested to any level, and be spread over any number of lines.

The rule should be followed that next statements follow the **for** statements they are matched with. This example of "tricky code":

```
10 goto 30
20 next
30 for I = 1 to 10: print i: goto 20
```

Would function in an interpretive environment, but would not work in a compiler, in case you ever want to generate fast binaries. Thus these kinds of styles should be avoided.

6.20 while statement

while expression … **wend**

The **while** and **wend** statements form a loop controlled by the expression of the **while** statement. Unlike a **for** statement, the controlling expression is evaluated every time the loop iterates. As long as the expression evaluates to true or non-zero, the statements in the **while** loop are executed. When the expression value is false or 0, the body of statements in the **while** loop are skipped and execution continues after the **wend** statement. Thus a while loop may not execute the contents of the loop at all.

Example:

```
10 i = 1
20 while i <= 10
30
40     print i
50     i = i+1
60
70 wend

 1
 2
 3
 4
 5
 6
 7
 8
 9
 10
```

while loops can be nested with other while loops and other control structures to any level. The **wend** statement must be located in the program source after the **while** statement. This is because if the **while** condition is false, statements are skipped, not executed, until the **wend** statement is found.

6.21 repeat statement

repeat … **until** expression

The **repeat** and **until** statements form a loop that is controlled by the **until** expression. Unlike a **for** statement, the controlling expression is evaluated every time the loop iterates. As long as the expression evaluates to false or zero, the statements in the **repeat** loop are executed. When the expression value is true or non-zero, the **repeat** loop exits and execution continues after the **until** statement.

Example:

```
10 i = 1
20 repeat
30
40    print i
50    i = i+1
60
70 until i = 10

1
2
3
4
5
6
7
8
9
```

repeat loops can be nested with other while loops and other control structures to any level. The **until** statement should located in the program source after the **repeat** statement.

6.22 select statement

select expression .. **case** expression: [**other**:] **endsel**

A **select** statement gives a means to execute different alternative program paths depending on the value of an expression. It is like the **on** … **goto** statement except that it is a true multiline capable programming structure.

The **select** expression is evaluated, then each ensuing **case** statement is examined to see if it contains the matching value by evaluating the **case** expression and checking for equal with the **select** expression result. If the **case** statement is matched, then the statements following the **case** statement are executed until the next **case** statement is found. The rest of the **case** statements are then skipped until **endsel** is found, and execution continues with the next statement after that.

If an **other** clause exists, then that will match any **select** expression result. This allows a single case to handle all other possible **select** expression values. The **other** clause behaves just as a **case** statement.

select and **case** expressions can be integer, real or string.

Examples:

```
select 1

    case 1: print "one"
    case 2: print "two"
    case 3: print "three"
    other:  print "something"

endsel

one

select 1.2

    case 1.1: print "point 1"
    case 1.2: print "point 2"
    case 1.3: print "point 3"

endsel

point 2

select "fork"

    case "knife": print "its a knife"
    case "spoon": print "its a spoon"
    case "fork":  print "its a fork"
    other:        print "don't know what it is"

endsel

its a fork
```

6.23 open statement

open filename **for input**|**output as** #number

Opens a text file by filename for either input or output using the given file number as a handle. File numbers can be in the range 3 to 100. The file numbers 1 and 2 are reserved for standard input an standard output.

When a file is open for reading, it is positioned to the start of the file and made ready to read. When a file is open for writing, it is set to zero length and writing starts at the beginning. P5 Basic only allows reading or writing to a text file, not both at the same time.

It is an error if the file is opened for reading, but a file by that name does not exist.

If the file by number is already open, it is automatically closed, then reopened under the new name.

6.24 close statement

close #number

Closes a text file by filenumber. The file number is then free to be used for other files. It is an error to close either the standard input file (1), or the standard output file (2).

6.25 end statement

end

The **end** statement terminates the program and causes it to stop and enter interactive mode.

It is considered poor form not to have the **end** statement as the final statement in a program, from a program readability standpoint. If you wish to have the program stop in the middle, the normal practice is to use the **stop** statement (see **stop** statement), or **goto** the final line containing the **end** statement.

6.26 dim statement

dim variable[,variable]

The **dim** statement defines the exact dimensions of arrays in P5 Basic. P5 Basic automatically declares variables when they are used. However, the default string length of 250 characters may be excessive, especially in large arrays of strings. Thus it is often useful to set the exact string length used.

For each variable defined as an array, the subscript used with dim is the total size of the array, not an index value:

```
10 dim a$(100), b.c(10, 10)
```

Here the size of the string a$ is set at 100 characters, and the size of the matrix c in record b is set to 10x10.

Using a dim statement can also help code size and speed when P5 Basic is complied (sometimes dramatically). The dynamic creation of variables in code without dim statements can be a source of considerable runtime overhead.

It is an error to **dim** variables after their use. It is an error to dim the same variable multiple times.

6.27 def statement

def functionname[(parameter[,parameter]…) = expression

Defines a single line function. The function name, which can be any identifier (not "fn…" of older Basics) is registered as a function with optional parameters. Each parameter is a variable declaration for a variable that exists only while the function is being executed.

The expression on the right hand side supplies the definition of the function. Each parameter may appear in the expression and behaves as a variable that was loaded with the data passed by the caller. Thus:

```
10 def add(a, b) = a+b
20 print add(1, 2)
 3
```

The result returned by the function is the result of the expression that defines it.

The function definition must be executed or appear before it is called. In a compiled environment, it must appear first in source regardless of execution order.

The appearance of a function overrides the function name use as an array. If an array was defined when the function is defined, the contents of the array become unusable. Naming functions and arrays the same is to be avoided.

6.28 Function/endfunc statement

function functionname[(parameter[,parameter]…) … **endfunc** expression

Introduces a multiline function. The function name, which can be any identifier is registered as a function with optional parameters. Each parameter is a variable declaration for a variable that exists only while the function is being executed.

The statements defining the function are terminated by an endfunc statement. Any number of lines can constitute the function. The expression evaluated by the endfunc statement defines the result of the function:

```
10 function min(a, b)
20
30     if a > b then a = b
40
50 endfunc a
```

The function definition must be executed or appear before it is called. In a compiled environment, it must appear first in source regardless of execution order.

The appearance of a function overrides the function name use as an array. If an array was defined when the function is defined, the contents of the array become unusable. Naming functions and arrays the same is to be avoided.

6.29 procedure statement

procedure procedurename[(parameter[,parameter]…) … **endproc**

Introduces a multiline procedure. The procedure name, which can be any identifier is registered as a procedure with optional parameters. Each parameter is a variable declaration for a variable that exists only while the procedure is being executed.

The statements defining the procedure are terminated by an endproc statement. Any number of lines can constitute the procedure:

```
10 procedure printit(a, b)
20
30     print “a”; a; “b”; b
40
50 endproc
```

The procedure definition must be executed or appear before it is called. In a compiled environment, it must appear first in source regardless of execution order.

The appearance of a procedure overrides the procedure name use as an array. If an array was defined when the procedure is defined, the contents of the array become unusable. Naming procedures and arrays the same is to be avoided.

6.30 randomize statement

randomize

Overrides the reset of the random number generator used in **rnd**(). The pseudo-random number generator normally gives a series of random numbers that is the same series each time the program is run. This can be overridden by the **randomize** statement, which causes it to continue with the series instead of restarting.

6.31 trace statement

trace

Turns on trace mode. In trace mode, each time a statement is executed, the line the statement comes from is printed. This can help debug programs.

6.32 notrace statement

notrace

Turns off **trace** mode. See the **trace** statement.

6.33 bye statement

bye

Exits the P5 Basic interpreter.

7 Expressions

Expressions in P5 Basic consist of standard operators, standard functions, and user defined functions. The operators are, in order from lowest to highest precedence, in groups of equal precidence:

a **and** b	Find the bit or logical and of a and b.
a **or** b	Find the bit or logical or of a and b.
xor	Find the bit or logical xor of a and b.
a = b	Equals. Returns -1 if a = b, otherwise 0.
a <> b	Not equals. Returns -1 if a <> b, otherwise 0.
a < b	Less than. Returns -1 if a < b, otherwise 0.
a > b	Greater than. Returns -1 if a > b, otherwise 0.
a <= b	Less than or equal. Returns -1 if a <= b, otherwise 0.
a >= b	Greater than or equal. Returns -1 if a >= b, otherwise 0.
a + b	Add a and b
a – b	Subtract b from a.

a*b	Multiply a and b.
a/b	Divide a by b.
a **mod** b	Find a modulo b
a **div** b	Divide a by b in integer mode.
a^b	Exponential. Find a to the b power.
+a	Positivition. Find a.
-a	Negation. Find –a.
not a	Logical bit not of a.
(a)	Subexpression. Find a.
func(a)	Find function.

Logical values are always returned as true = -1 and false = 0, because -1 is 2's complement notation for all bits high in the integer result. This is convienent in conjuction with Boolean math. For example:

1 = 1 and 1 <> 2	is 0, because -1 and 0 are 0.
1 = 1 and 2 = 2	is -1, because -1 and -1 is -1.
1 = 1 and 2 `	is 2 because -1 and any value is the same value.

7.1 Standard functions

left$(string, length)

Finds the length number of characters at the left of the string.

right$(string, length)

Finds the length number of characters at the right of the string.

mid$(string, position, length)

Finds length number of characters starting at the position, which is 1 to n.

chr$(value)

Finds the ASCII character corresponding to the character code value.

asc(string)

Finds the ASCII character code corresponding to the first character in the string.

len(string)

Finds the length, in number of characters, of the string.

val(string)

Finds the value of a number in the string (parses and converts the number). The value can be integer or real, and may contain an exponent.

str$(value)

Converts the number value to its equivalent string. Real or integer can be converted. The same comments as **print** values apply here (see **print** statement).

rnd(value)

Finds the next random number from the pseudo-random number generator in the range 0 to 1.

int(value)

Finds the integer value of real. The fraction is truncated.

sqr(value)

Finds the square root of the value.

abs(value)

Finds the absolute value.

sgn(value)

Find sign of value, which is 1 if the value is positive, or -1 if the value is negative.

sin(value)

Find the sine of the value.

cos(value)

Finds the cosine of the value.

tan(value)

Finds the tangent of the value.

atn(value)

Finds the arctangent of the value.

log(value)

Finds the log of the value.

exp(value)

Finds the exponential of the value.

lcase(string)

Finds the lower case equivalent of the string.

ucase(string)

Finds the upper case equivalent of the string.

eof(#filenumber)

Finds if the file by filenumber is at the end of that file. Returns -1 if so, otherwise 0.

8 ISO 7185 extension routines

P5 Basic provides for several extentions to the ISO 7185 standard for use in the language. These include named file I/O and more efficient Boolean bit math. Most ISO 7185 Pascal implementations (actually at this writing all) have this functionality. The routines to enable this functionality are arranged at the top of the Basic source program and are easily configured for the target Pascal implementation.

For named file I/O, the routines enable the entire set of:

- Basic program read and save from/to a file.
- File I/O operations within Basic.

The Boolean math routines that are standard with P5 Basic are functional, but not efficient. You can replace them, but this is for the purposes of efficiency only.

8.1 Routines

8.1.1 Named file input

For all of the following procedure/functions, filnam is declared:

```
Filnam = packed array [1..maxfln] of char;
```

and is an array of characters containing the filename, which is padded on the right side with spaces, and the filename starts at array position 1. The exact formatting of the filename, what characters, what path or other special characters is system specific. P5 Basic specifies filenames in quotes and will pass all characters given down to the following routines unmodified.

`function existsfile(var fn: filnam): boolean;`
Checks if the file exists. Returns true if the file exists, otherwise false.

`procedure openread(var f: text; var fn: filnam);`
Open file for reading. The given text file is opened for reading with the filename. Usually this means that reset() or equivalent is applied to the file, which sets the file at the start and ready for reading.

`procedure openwrite(var f: text; var fn: filnam);`
Open file for writing. The given text file is opened for writing with the filename. Usually this means that rewrite() or equivalent is applied to the file, which sets the file as empty and ready for writing.

`procedure closefile(var f: text);`
Closes the file. The file is disassociated from its file by name, and the file is made ready for another openread() or openwrite() call.

8.1.2 Boolean bit operations

`function bnot(a: integer): integer;`
Finds the bitwise 'not' of the bits in the integer. This is then returned. Must be able to perform this operation on negative, 2's complement format integers.

`function bor(a, b: integer): integer;`
Finds the bitwise 'or' of the bits in two integers. The result is then returned. Must be able to perform this operation on negative, 2's complement format integers.

`function band(a, b: integer): integer;`
Finds the bitwise 'or' of the bits in two integers. The result is then returned. Must be able to perform this operation on negative, 2's complement format integers.

`function bxor(a, b: integer): integer;`
Finds the bitwise 'or' of the bits in two integers. The result is then returned. Must be able to perform this operation on negative, 2's complement format integers.

9 Using ISO 7185 mode

In ISO 7185 strict mode, there is no named file I/O, and program files cannot be directly loaded and saved to disk. However, there are two ways in which programs can be loaded and run, and also saved to disk.

9.1 Batch mode

By completely running P5 Basic in batch mode, a fully automated program run can be made. The input is redirected from a file, and the output is redirected to a file (or simply printed to console):

P5 basic < infile > outfile

Thus a sample input file would be:

10 print "hi"
20 end

Run

Then the output file would be:

hi

9.2 Cut and paste

A program can also be entered by copying it to the clipboard, then pasting it onto the window running P5 Basic. The program will need line numbers to do this, since console line input mode is being used.

Similarly, the program can be saved by using list to print it, then copying the result to the clipboard, then pasting to an editor and saving.

In this way a fully interactive session can be created.

10 BNF Grammar Specification

This chapter provides a complete BNF (Backus-Naur Form) grammar specification for the P5 BASIC language. The grammar is organized hierarchically, starting with program structure and proceeding through statements, expressions, variables, and lexical elements.

10.1 Program Structure

```
<program> ::= <line>*
```

```
<line> ::= [<line-number>] <statement-list> <end-of-line>
| [<line-number>] <end-of-line>
| <label> ":" <statement-list> <end-of-line>


<line-number> ::= <digit>+


<label> ::= <identifier>


<statement-list> ::= <statement> [":" <statement>]*


<end-of-line> ::= <newline> | <eof>
```

10.2 Statements

The complete statement syntax includes:

```
<statement> ::= <input-stmt>
| <print-stmt>
| <goto-stmt>
| <on-stmt>
| <if-stmt>
| <rem-stmt>
| <stop-stmt>
| <run-stmt>
| <list-stmt>
| <new-stmt>
| <let-stmt>
| <load-stmt>
| <save-stmt>
| <data-stmt>
| <read-stmt>
| <restore-stmt>
| <gosub-stmt>
| <return-stmt>
| <for-stmt>
| <next-stmt>
| <while-stmt>
| <wend-stmt>
| <repeat-stmt>
| <until-stmt>
| <select-stmt>
| <case-stmt>
| <other-stmt>
| <endsel-stmt>
| <open-stmt>
| <close-stmt>
| <end-stmt>
| <dim-stmt>
| <def-stmt>
| <function-stmt>
| <endfunc-stmt>
| <procedure-stmt>
| <endproc-stmt>
| <randomize-stmt>
| <trace-stmt>
| <notrace-stmt>
| <bye-stmt>
| <procedure-call>
```

10.3 Expression Hierarchy

Expressions are parsed according to the following precedence hierarchy (from lowest to highest precedence):

1. Logical operations (and, or, xor)
2. Relational operations (=, <>, <, >, <=, >=)
3. Additive operations (+, -)
4. Multiplicative operations (*, /, mod, div)
5. Exponentiation (^)
6. Unary operations (+, -, not)
7. Primary expressions (literals, variables, functions, parentheses)

```
<expression> ::= <logical-expr>

<logical-expr> ::= <relational-expr> [<logical-op> <relational-
expr>]*

<logical-op> ::= "and" | "or" | "xor"

<relational-expr> ::= <additive-expr> [<relational-op> <additive-
expr>]*

<relational-op> ::= "=" | "<>" | "<" | ">" | "<=" | ">="

<additive-expr> ::= <multiplicative-expr> [<additive-op>
<multiplicative-expr>]*

<additive-op> ::= "+" | "-"
```

```
<multiplicative-expr> ::= <exponential-expr> [<multiplicative-op>
<exponential-expr>]*


<multiplicative-op> ::= "*" | "/" | "mod" | "div"


<exponential-expr> ::= <unary-expr> ["^" <unary-expr>]*


<unary-expr> ::= [<unary-op>] <primary-expr>


<unary-op> ::= "+" | "-" | "not"
```

10.4 Variables and Literals

```
<variable> ::= <simple-variable> | <array-variable> | <record-
variable>


<simple-variable> ::= <identifier> [<type-suffix>]


<array-variable> ::= <identifier> [<type-suffix>] "(" <expression>
["," <expression>]* ")"


<record-variable> ::= <identifier> "." <identifier> ["."
<identifier>]*


<type-suffix> ::= "%" | "$"
```

Where % indicates integer type, $ indicates string type, and no suffix indicates real type.

10.4.1 Literals

```
<integer-literal> ::= [<sign>] <digit>+
| "$" <hex-digit>+ (* hexadecimal *)
| "&" <octal-digit>+ (* octal *)
| "%" <binary-digit>+ (* binary *)


<real-literal> ::= [<sign>] <digit>+ "." [<digit>*] [<exponent>]
| [<sign>] [<digit>*] "." <digit>+ [<exponent>]
| [<sign>] <digit>+ <exponent>


<string-literal> ::= '"' <string-char>* '"'


<identifier> ::= <letter> [<letter> | <digit>]*
```

10.5 Grammar Notes

• Case is not significant except within string literals

• Line numbers range from 0 to 9999

• String maximum length is 250 characters

• Maximum 255 variables in a program

• File numbers range from 1 to 100, with 1=stdin and 2=stdout

• Identifiers can be fully alphanumeric with underscores, unlike classic BASIC

For complete syntax details of individual statements, see the corresponding sections in earlier chapters.