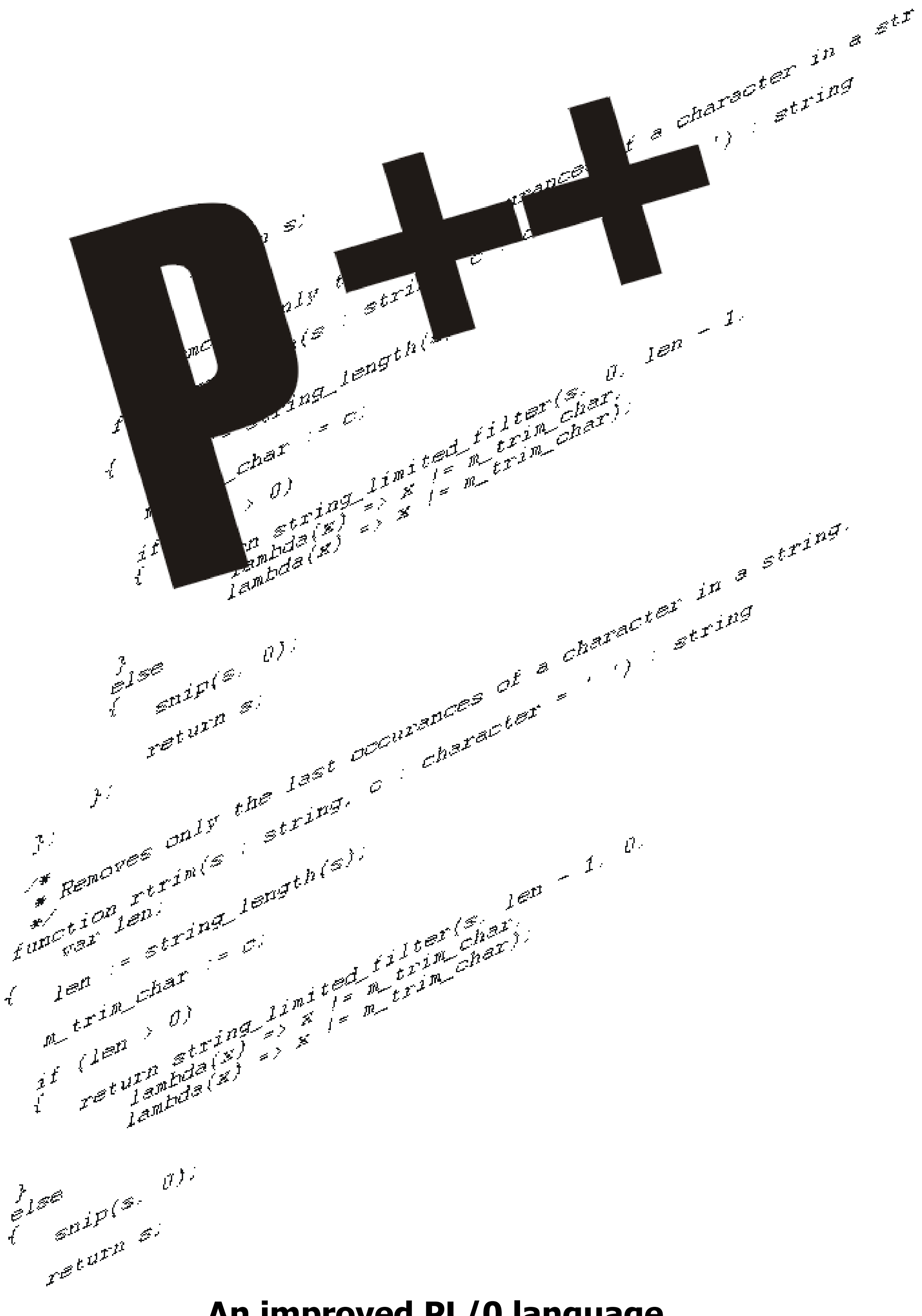


An improved PL/0 language

Programmer's Guide

By Thông Nguyen

Revision 1.0

Table of contents

Introduction

This programmer's guide to P++ presumes some familiarity with programming languages like C and Pascal.

Line number in examples aren't part of the source code, but simply markers.

2.0 Language Basics

2.1 Source code

P++ source code must be written with ASCII standard text, all commands and statement blocks are terminated the C style way, with a semicolon.

2.2 Source code commenting

There are two styles of comments that can be inserted into P++ source code.

2.2.1 C style block commenting

C style block comments, may be inserted anywhere in the source code and can be embedded. Block comments are started with a `/*` and terminated with a `*/`.

Example:

```
/*
 * This is a comment
 * /* this is an embedded block comment */
 */
```

2.2.2 C++ style line commenting

C++ style line comments whenever inserted into a line, will remark out the rest of the line as a comment.

Example:

```
var x : integer // This is a line comment
```

2.3 Numbers

In source code, when working with numbers, simply type out the number as needed. There are 4 formats of numbers you can use, integers, floats, characters and hexadecimal.

They are all demonstrated here:

```
x := 65;  // integer
x := 65.0;     // float
x := 'A';      // character
x := 0x41;     // hexadecimal
```

2.3 Strings

Strings in P++ must be declared within speech marks and can include escape characters.

2.4 Escape characters

It is often desirable to use special characters in strings that aren't immediately representable in source code such as new lines, beeps, tabs and speech marks.
P++ supports doing this by using C style escape characters. Each special character is represented by an escape character. To insert these inside strings or character declarations, use the backslash character proceeded by the escape character.

Escape character	ASCII	Description
n	10 or 0x0a	New line.
r	13 or 0x0d	Carriage return.
"	34 or 0x22	Double quotation (speech mark).
'	39 or 0x27	Quotation.
\	92 or 0x5c	Backslash.
t	09 or 0x09	Horizontal tab.
v	11 or 0x0b	Veritical tab.
?	63 or 0x3f	Question mark.
a	07 or 0x07	Bell.
b	08 or 0x08	Backspace.
XXX	--	XXX is a decimal number.
oXXX	--	XXX is an octal number.
xXXX	--	XXX is a hexadecimal number.
0	0	Null character.

2.4.1 Escape character examples.

The following statements all assign 255 to the variable x.

```
var x : character;

x := 255;
x := 0xff;
x := '\255';
x := '\xff';
x := '\o377';
```

The following statement will produce a string with a smiley face (\1), a tab (\t) and a new line (\n).

```
var s : string;

s := "\1\thello\n";
```

The variable s will now contain:

```
☺       hello
```

2.6 Expressions

P++ treats any non zero number as true, and 0 as false.

2.7 Include

You can include files anywhere outside functions by using the include keyword.

2.8 Identifier naming

Identifiers in P++ include variable, constant and function names. The following are the set of restrictions on identifier names.

- Identifiers consist of alphanumeric characters and underscores.
- Identifiers can be 1 to 255 characters long.
- Identifiers must start with an alphabetical character.

2.9 Reserved keywords

The following is a table of reserved keywords that can't be used as identifiers.

begin	end	call	const	do
do	if	else	odd	procedure
then	var	while	integer	for
to	step	exit	function	return
goto	include	lambda	repeat	which
case	ref	val	asm	machine_code
float	void	main	handle	string
array	character	boolean	stream	byte
not	or	and	pragma	boundary_check
continue	null			

3.0 Data types and variables

3.1 Primitive data types

P++ currently only supports 3 primitive data types, these are:

- **Integers**

  Integers are 32bit unsigned integers.

  P++ type name: integer

  The have a range from -2147483648 to +2147483648.

- **Characters**

  Just integers.

- **Floats**

  Floats are 32bit unsigned floating-point numbers represented in IEEE 754 format (single precision floats).

  P++ type name: float

  They have a range from 1.40239846E-45 to 3.40282347E+38.

3.2 Structures

P++ has no support for structures (aka records) yet.

3.3 Advanced types

- **Functions.**

  P++ supports using functions and procedures as types. These will be covered in the chapter on functions.

- **Handles**

  Handles are used as a way to pass arrays around without reference counting or type checking. They are reserved for use by low-level support libraries (like in array.p++).

3.4 Arrays

P++ supports one-dimensional arrays of any type, except of other arrays.
Arrays in P++ are automatically garbage collected using transparent reference counting.
Arrays in P++ are dynamic, they automatically grow beyond their boundaries when needed!

See section 9.0 on arrays.

3.6 Variable declarations

Variables in P++ can be must be declared before use, and unlike low-level languages such as C, all variables have default initial values. Primitive number types default to 0, strings default to "", and array references default to null.
Variable declarations must appear before statement blocks, they can not be declared along with statements (see 5.0 functions).
Variables are declared by using the "var" statement, followed by a variable name, a colon, and a type. You can declare a variable with the same name many times, and the most recently declared variable will be used.

3.6.1 Primitive declaration examples:

```
/*
 * Declares a variable called "x" that's implicitly an integer.
 */
var x;

/*
 * Declares a variable called "x", explicitly of the type integer.
 */
var y : integer;

/*
 * Declares an variable called "x", explicitly of the type integer.
 * And sets the initial value to be 10.
 */
var z : integer = 10;

/*
 * Declares two variables, "a" is an integer with "a" default of 0
 * and "b" float with "a" default of 10.
 *
 * Notice how "a" is NOT a float, and does not default to 10.
 */
var a, b : float = 10;

/*
 * Declares two floats, both with the default value 10.
 */
var a : float = 10, b : float = 10;
```

3.5.2 Array declaration examples

```
/*
 * Declares variable x to be an array of 100 integers.
 * The array will grow as it is used.
 */
var xx : integer[100];

/*
 * Declares variable x to be an integer array of any size.
 * The array will grow as it is used.
 */
var xx : float[..];
```

3.5.3 Variable naming conventions

Variables should have lower case names, and use the underscore _ character between logical words.

3.6 Constant declarations

P++ supports the idea of constants, these are variables whose value never changes, and at compile time, the values are substituted. Constants saves you having to search and replace numbers or strings, you can just change them where ever you want. They have an advantage over declaring variables with default values because they tend to be faster.

3.6.1 Example of declaring constants

```
const PI : float = 3.1415926;
const MESSAGE : string = "P++ rocks";
```

As you can see, declaring constants is almost identical to declaring variables, but you use the *const* keyword rather than the *var* keyword.
Using constants is identical to using variables, just substitute the name of the constant.

3.6.3 Constant naming conventions

Constants should always have names in UPPERCASE to distinguish them from variables.

3.6.4 Using constant strings

Generally, constants are faster than using variables, because the values can be known at compile time. But there are some implementation issues that need to be known with constant strings if you want to do advanced optimisation.
Constant strings are dynamically created whenever the name of the constant is used. So in the above case with the constant string “P++ Rocks”, every time the constant MESSAGE is used, a new string containing “P++ Rocks” is created. The reason there is not one instance of the string is to allow the string to be modified once it is created. If you passed the constant MESSAGE to a function, and the function modified the string, the constant MESSAGE will still represent the original string. This makes constant strings more consistent, as they work exactly the same way as if you had used the literal string “P++ Rocks” in place of the constant MESSAGE.
There is obviously a performance issue if you know your functions will never modify the string, so in some cases it might be faster to create a new variable that holds the constant string, and pass that variable around. In this case, the same string will be passed to each function, only one string will ever be created, and it will be much faster (and produce less code).

4.0 Your first P++ program

In order to continue further, it’s best to just throw in a simple P++ to get an idea of what a P++ program looks like.

```
1:  // sample1.p++
2:  include "io.p++";
3:
4:  var x;
5:
6:  main
7:  {
8:        x := 10;
9:        write(x + 1);
10:       write(x * 2);
11: };
```

Program output:

```
11
20
```

Lets examine this program line by line.

2: Include the file “io.p++”. This file contains more P++ source code, that defines basic input output functions, we need this so we can use the write statement.

4: Declare a variable called x. This variable will default to 0, and is of the default type, integer.

6: Start the main block. This is not a function! And is not necessary, but makes code tidier to read.

7: Start the main block. This is necessary. P++ uses curly braces to define the start and end of statements blocks.

8: Uses the Pascal style assignment operator “:=”, and puts the number 10 inside x.

9: Writes the number 11 (10 + 1) to standard output.

10: Writes the number 20 (10 * 2) to standard output.

5.0 Operators

5.1 Assignment

P++ uses the Pascal style := operator for assignment. However, unlike Pascal, the assignment operation can be aggregated like in C because P++ assignments are expressions.

For example, you can go:

```
x := y := z = 100;
```

Which sets x, y and z to 100.

5.1 Mathematical

5.1.1 Table of mathematical operators.

Operator	Description	Example
-	Subtract	2 + 2
+	Add	2 - 2
*	Multiply	2 * 2
/	Divide	2 / 2
%	Modulus	2 % 2
@	Absolute (Unary)	@2

5.2 Logical

Logical operators always return 0 or 1 (true or false).

5.2.1 Table of logical operators.

Operator	Description	Example
!	Logical not	!x
&&	Logical and	x && y
||	Logical or	x || y
>	More than	x > y
<	Less than	x < y
>=	More than or equal to	x >= y
<=	Less than or equal to	x <= y

5.3 Bitwise

Bit wise operators only work on integers (use the f2i_nocast function if you need to work on floats).

5.3.1 Table of bit wise operators

Operator	Description	Example
&	Bit wise and	x & y
|	Bit wise or	x | y
^	Bit wise xor	x ^ y
~	Bit wise not (unary)	~x

5.4 Unary

5.4.1 Table of unary operators

Operator	Description	Example
++	Adds one	x++; or ++x;
--	Subtracts one	x--; or --x;
+=	Inline add	x += 2;
-=	Inline subtract	x -= 2;
*=	Inline multiply	x *= 2;
/=	Inline divide	x /= 2;
&=	Inline bit wise and	x &= 2;
|=	Inline bit wise or	x |= 2;
^=	Inline bit wise xor	x ^= 2;

When using ++ and --, there are special considerations.
When the ++ or – operator is placed before a variable, the variable is incremented or decremented and then the value of the variable loaded.
When the ++ or – operator is placed after a variable, the variable is loaded, and then the value of the variable is incremented or decremented.

Note: Doing unary operations on array elements that contain numeric types is now supported.

```
x := 10;
x-- + --x;
```

x = 8
expression = 18

x = 9
expression = 10

6 Expressions

6.1 Introduction to expressions

Expressions always evaluate to a result, and almost anything can be an expression – even assignment statements.

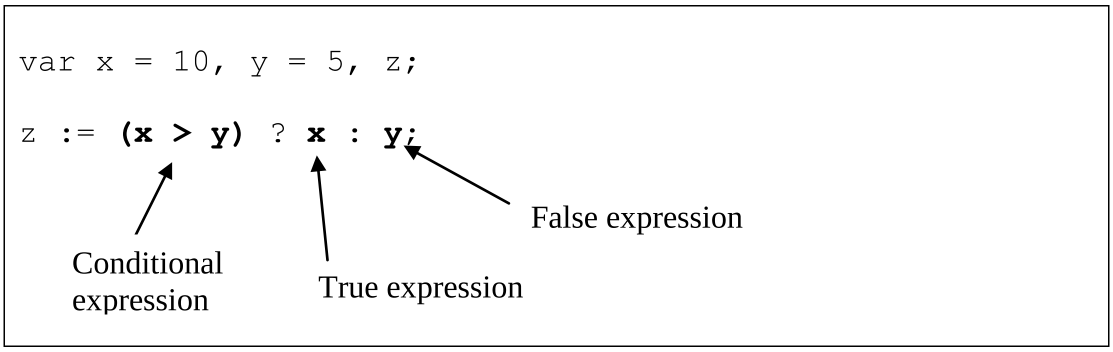
6.1.1 Table of basic expressions.

This is a table of some simple expressions in P++.

Expression type	**Example**	**Comments**
Mathematical	`x + y * 2`	
Logical	`x && y`	Always either true or false.
Bit wise	`x | y`	Always works on integers.
Assignment	`x := y := z := 3`	
Function calls	`foo(100)`	
Lambda	`lambda(x) => x + 1`	
Variables	`X`	True if non zero.
Inline if	`(x > y) ? x : y`	This returns the max of x/y.

6.2 Ternary conditional expression (inline if)

6.2.1 Ternary conditional expression example



```
var x = 10, y = 5, z;

z := (x > y) ? x : y;
```

Z will become 10 after the above expression has been evaluated.
The way a TCE expression works is like this.

1) The condition expression is evaluated to true or false (0 is false, anything else is true).
2) If the conditional expression is true, the true expression is the result of the expression overall.
3) If the conditional expression is false, the false expression is the result of the expression overall.

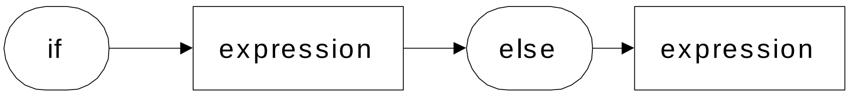
7.0 Statements

7.1 Introduction to statements

Statements never evaluate to any results, they basically "do" things, rather than "do" and "give" things.

7.1 If

If statements allow certain statements to be carried out if an expressions is either true or false.



7.1.1 If statement example 1

```
if (x = 0) ←———— Expression (don't need parenthesis)
{
    print("x is 0");
}
else
{
    print("x isn't 0");
};
```

In the above example, "x is 0" will be printed if x = 0, or in other words if the expression is true. Remember, any expression is true if it's non zero, so the expression could be changed to (!x) and it would still work exactly the same.

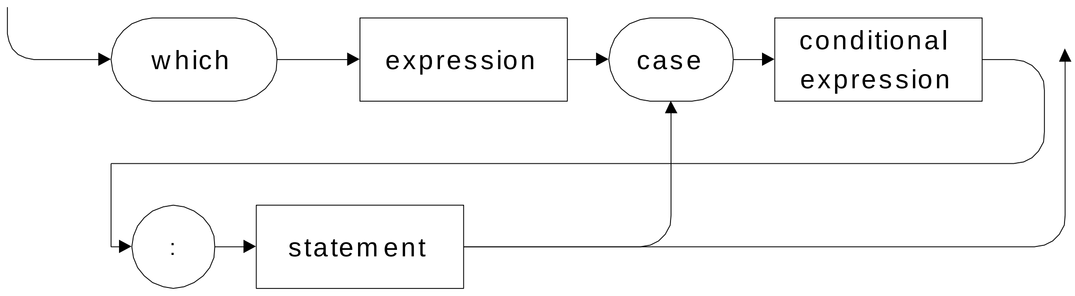
7.1.3 If statement example 2

```
if (x) then print("x is 0") else print("x isn't 0");
```

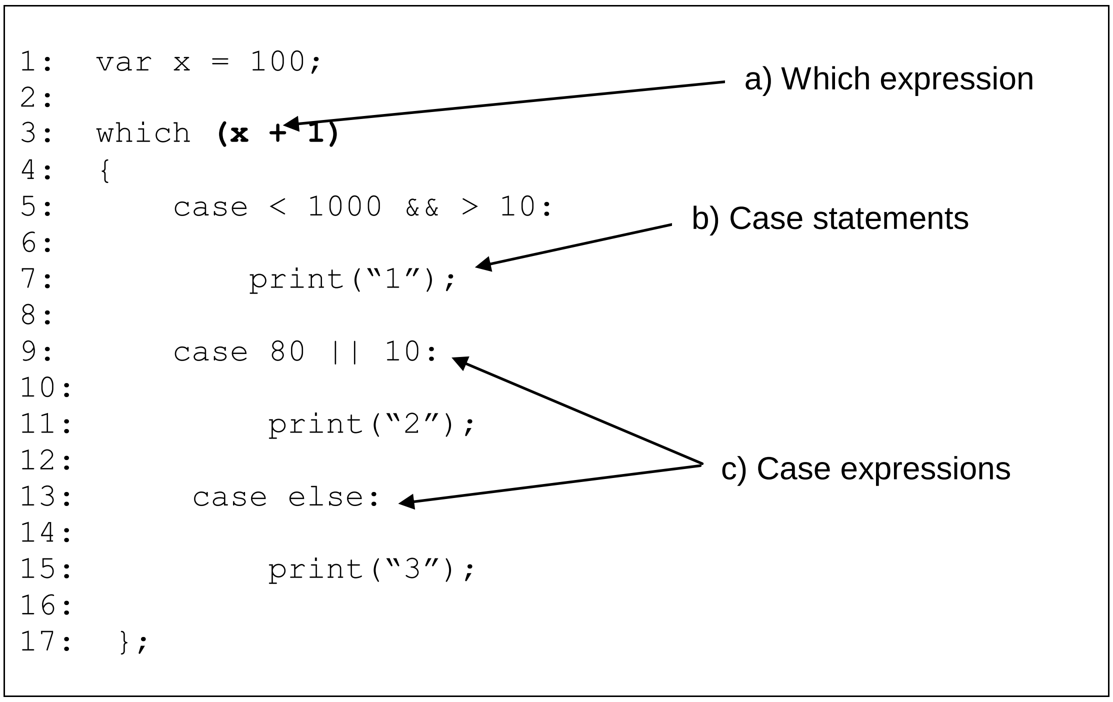
The above example shows doing an if statement on one line without braces. Only one statement for each true/false condition is allowed, and you need to use the *then* keyword after the *if expression*.

7.2 Which

The which statement a block control structure very similar to C's switch statement, and BASIC's select case. Unlike C's switch statement, which can be more useful because it allows multiple conditions per case, and works on floats and strings (not just integers).
Which statements essentially let you compare a series of expressions with one initial expression, one after the other, and execute statements based on a match.



7.2.1 which statement example


a) Which expression
b) Case statements
c) Case expressions

```
var x = 100;

which (x + 1)
{
    case < 1000 && > 10:

        print(“1”);

    case 80 || 10:

         print(“2”);

     case else:

         print(“3”);

 };
```

Each which statement needs a which expression (a), and a series of case statements (b). There is no need to surround multiple statements for each case with braces. Each case exists of either an expression, or multiple expressions separated by the logical `&&` and `||` operators. There is a special case expression called *else*, this case executes if no other case applies.

To examine how case statements handle their expressions, we can look at the case at line 5.

```
case < 1000 && > 10:
```

This expression of the case would expand to the following expression:

```
((x + 1) < 1000) && ((x + 1 )> 10):
```

7.2.1 Use of logical AND and logical OR in which case statements

Since which case statements use the `&&` and `||` operators to bind together partial expressions, it's possible to have confusion arise over the use of these logical operations of separate expressions in a case.

For example:

```
case > y && z && > 2:
```

The above case statement expands to:

```
((x + 1) > y) && (x = z) && (x > 2)
```

When the actual intended expression might have been:

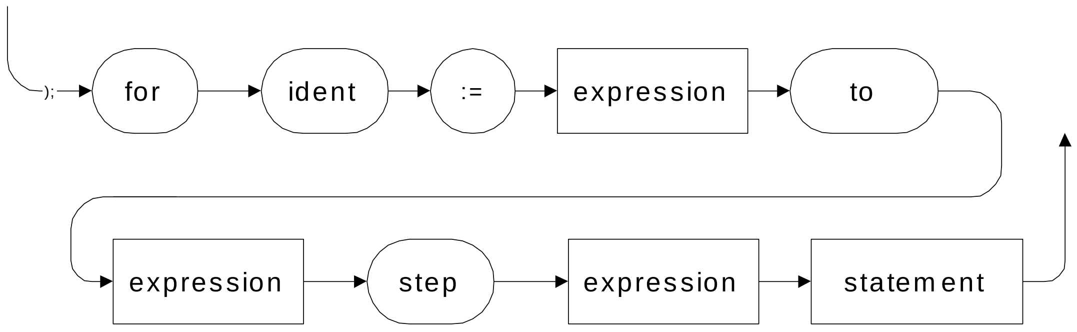
```
((x + 1) > (y && z)) && (x > 2)
```

In order to fix this, you should use parenthesis to separate expressions clearly. All logical operators inside parenthesis will be considered part of the inner expression, and not special case directives for AND and OR.

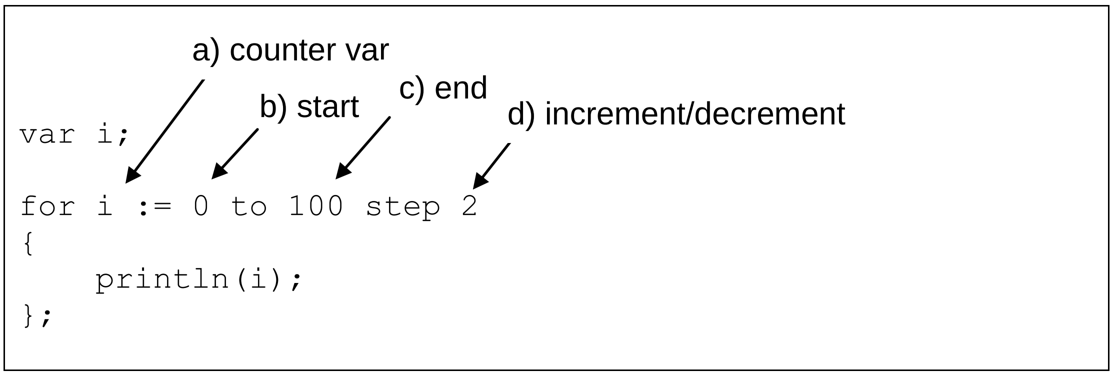
```
case (> (y && z)) && (> (2)):
```

7.3 For

P++ for statements are very similar to Pascal and BASIC for statements, and they allow you to count up and down in various steps.



7.3.1 for statement example



```
var i;

for i := 0 to 100 step 2
{
    println(i);
};
```

This example will print out all the even numbers from 0 to 100 (inclusive) .

All for loops need a loop counter, which has to be a numeric type (integer or float), in this example, the loop counter is the integer “i”. Each for loop starts by initialising the counter to a starting value, in this case “0”. Then you specify the end to stop counting at, in this case “100”. All of these must be specified and are not optional. There is an optional step statement you can make after the end of the loop, which specifies how much the counter is increment each loop. The default step is 1.

7.3.2 counting backwards

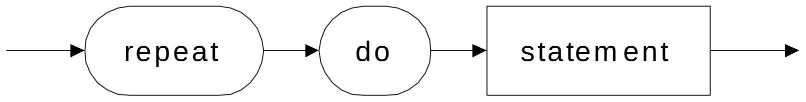
To have a loop count backwards, just have the start larger than the end, and specify a negative step. Make sure that you have a negative step or else the for loop will loop forever! Also note, that like BASIC for loops, a loop from the x to x (eg. `for i := 10 to 10`) will loop exactly once no matter what the ‘step’ is.

7.3.3 Loop interruption

See section 7.8 on the exit statement.

7.4 Repeat

Repeat statements are a simple mechanism for repeating for an undetermined amount of time.



7.4.1 Repeat statement example

```
repeat
{
    println(“Pokwer!”);
};
```

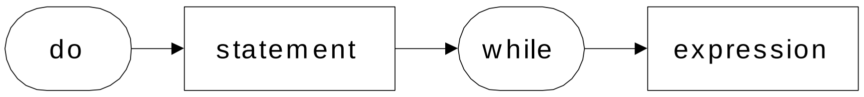
This above program will keep printing the line “Pokwer!” until it is terminated by the user.

7.4.2 Loop interruption

See section 7.8 on the exit statement.

7.5 Do

Do statements in P++ are identical to do statements in C. They are a form of loop statement that loops while an expression is true. The validation of the expression is done at the bottom of a do loop, so do loops always run at least once.



7.5.1 Do statement example

```
var x;

do
{
    write(x);
}
while (++x < 10);
```

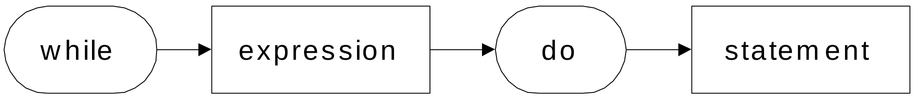
This program will print out all the numbers from 0 to 9 (inclusive).

7.5.2 Loop interruption

See section 7.8 on the exit statement.

7.6 While

While statements in P++ are identical to while statements in C. While statements are almost the same as do statements, they loop around while a specified condition is true. However, while statements evaluate the expression at the start of a loop, so unlike "do", they may not loop round even once.



7.6.1 While statement example

```
while (x < 10)
{
    write(x++);
};
```

This program will print out all numbers from 0 to 9 (inclusive).

7.6.2 Loop interruption

See section 7.8 on the exit statement.

7.7 Call

Call statements are left over from the original PL/0 compiler. They are used only to call procedures. You can't use them to call functions, and can you call procedures without using call. Using call is straight forward, as the following example demonstrates.

7.7.1 Call statement example.

```
procedure beep()
{
    write_byte('\a');
};

main
{
    call beep();
};
```

7.8 Exit

Exit statements allow you to break out of the flow of the program, whether it is a function, a loop or a which statement. They will bring you to the end of that logical block (exiting a which statement will take you to the statement right after the end of the which statement etc).

Exit able blocks.

Block type	**Statement used**
Function	`exit function;`
Procedure	`exit procedure;`
Which case	`exit which;`
Repeat loop	`exit repeat` **X**`;`
Do loop	`exit do` **X**`;`
While loop	`exit while` **X**`;`
For loop	`exit for` **X**`;`

The exit statements including **X**, mean that with these block types, you can specify the number of blocks to exit at once. This can be a powerful and fast mechanism for jumping out of multiple loops. This is demonstrated in the following example.

7.8.1 Example of exit X

```
do
{
    for i := 0 to 10
    {
        for i := 0 to 10
        {
            exit for;    // A
            exit for 2;  // B
            exit do;     // C
            exit do 2;   // D Error
        };
        // A'
    };
    // B'
}
while (true);
// C'
```

This diagram shows where an exit statement will take the program flow. A jumps to A', B jumps to B' etc. Notice how at point **D**, an attempt to exit 2 do loops was made, but there is only one do loop applicable. This will cause a compilation error.

7.8.2 Limitations of exit

Because of the complexity of some of the higher level constructs available in P++, there are some limits to the exit statement.

If you are using a *which* statement and the 'which expression' is a string, then no code embedded inside the which statement is allowed to break out of the which statement (using goto, exit or otherwise). The only exception is that you are allowed to do a one level exit which, and continue which (these statements are in the scope of the which statement, and gives it a chance to do garbage collection).

```
do
{
    which("hello")
    {
    case "hello":
        exit do;
    };
}
while (true);
```

In the above example, the exit do statement breaks out of the which and the do statement. This will cause the string "hello" to never be garbage collected. This problem does not occur with integers or floats.

This limitation will be fixed in a future release.

7.8 Continue

The continue statement is for loops and which statements. It's purpose in loops is to instruct the loop to skip all the remanning statements for the loop and continue to the next iteration of the loop. The continue statement makes sure that conditions for loops are evaluated, so for example, in a do/while loop, continue will jump to the bottom where the while is evaluated, but in a standard while loop, continue will jump to the top where the while is evaluated.

Continuable blocks.

Block type	**Statement used**
Which case	`continue which;`
Repeat loop	`continue repeat` **`X`**`;`
Do loop	`continue do` **`X`**`;`
While loop	`continue while` **`X`**`;`
For loop	`continue for` **`X`**`;`

The continue statements including **X**, mean that with these block types, you can specify the number of blocks to exit at once. This can be a powerful and fast mechanism for jumping and continuing within multiple loops. This is demonstrated in the following example.

7.8.1 Example of continue X

```
for i := 0 to 10
{
    // B'
    for j := 0 to 10
    {
        // A'
        continue for;      // A
        continue for 2;    // B
    };
};
```

This diagram shows where a continue statement will take the program flow. A jumps to A', B jumps to B' etc.

7.10 Return

Return statements are for exiting out of a function, and you can specify an expression which will be the return value of the function. The expression must be the same type as the function’s return value. Not specifying the return value will simply exit out of the function, and the return value will be the current return value (this can be accessed by the stock variable *retval*).

See section 8.8 on returning from functions.

7.12 Goto

Goto statements allow you to jump from to another portion of code. You can only jump from one part of a function to another part. You can not jump across functional boundaries (even between children/parent functional boundaries).
To jump to another portion of code, you need to define a goto label above the portion of code where the jump will jump to. Labels can only be letter labels, and must follow the identifier naming rules as described in section 2.8.

10.12.1 Example of using goto

```
var x = 10;

if (x = 10) then goto foo;

print(“wibble”);

foo:

print(“wobble”);
```

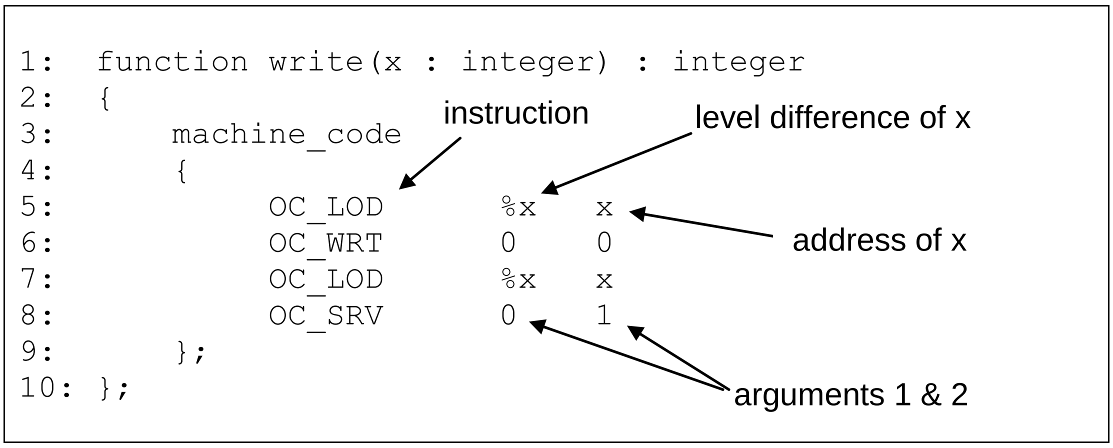
In the above example, if the variable *x* is 10, then only “wobble” will be printed, however if x isn’t 10, then both “wibble” and “wobble” is printed.
See section 7.8.2 on using goto with which statements.

7.13 machine_code

The machine code block allows you to literally enter P++ machine code instructions. It is similar to the ability in C to embed assembly code inside C code.

7.13.1 Example of using machine_code

This example is of the write statement, and can be found in *"io.p++"*.



```
function write(x : integer) : integer
{
    machine_code
    {
         OC_LOD      %x   x
         OC_WRT      0    0
         OC_LOD      %x   x
         OC_SRV      0    1
    };
};
```

As you can see, a *machine_code* block starts with the keyword *machine_code* and all machine code statements must lie within the open and close curly braces.
Each machine code instruction must occupy one line, and consists of 3 parts.
The first part is the instruction, this can either be a constant integer, or a literal integer. It can not be a variable since no dynamic code generation is supported.
The second and third parts are the two instructional arguments each P++ machine instruction takes. These can be literal integers, floats, or constants of integers and floats. Additionally, the arguments can also have special flags that make it possible to use variables for the arguments.

7.13.1 Table of advanced machine code instructional arguments

This table demonstrates advanced machine code instructional arguments. Assume that x is a variable, and X is a constant.

Instructional Argument	**Description**
`%x`	Level difference of x
`x`	Address of x
`&0`	The address of this instruction
`&1`	The address of the next instruction
`&x`	The address of the *xth* instruction after this one.
`X`	The value of the constant X.

8.0 Functions and procedures

8.1 Introduction to functions and procedures

P++ makes a distinction between functions and procedures primarily for compatibility with old style PL/0. It is recommendedthat in new programs functions are always used. There are two major limitations to procedures; they can’t return values, and they must be called with the call statement.
There is a third type of function that P++ also supports; this is the anonymous functions or “lambda expression”, which will also be covered later.

Functions in P++ are very flexible; they can take and return any type, including arrays, integers, floats and even other functions.

8.1 Declaring procedures

Procedures need a name, and an parameter list.

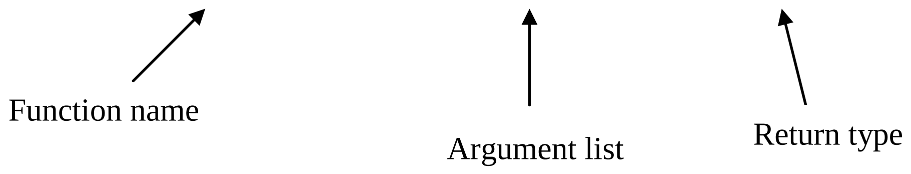
```
procedure write_plusone(x : integer)
```

This declaration is for a procedure that takes one integer.

8.2 Declaring functions

Functions need a name, and an parameter list and a type.

```
function write_plusone(x : integer) : integer
```



This declaration is for a function that takes one integer and returns an integer. Note that if the return type is not specified, the return type defaults to integer.

8.3 Calling functions

Calling functions is almost as simple as decalring them. You just need to supply the function *arguments* for each non-optional parameter. Since functions are expressions, and return values, you can embed function calls within expressions. You can optionally just call a function without assigning the return value if it's not needed.

8.3.1 Example of calling functions

```
function write_plusone(x : integer) : integer
{
    return write(x + 1);
};

var y;

main
{
    write_plusone(100);
    y := write_plusone(100);
};
```

In this example, the function *write_plusone* is called twice both are valid. In the second call, the return value (101) is put into the variable y.

8.4 Special parameter directives

The variable declarations in the parameter list for a function can include several special directives that change the way the parameters behave for the function and the caller.

8.4.1 Passing by reference

ref must be placed before the variable name, and specifies that the parameter is an outgoing parameter, that is, all values are passed indirectly by reference (a pointer argument in C terminology). This lets you change the values of any arguments passed.
The P++ compiler will automatically determine how a variable should be passed (by reference or by value) by looking at the signature of the function being called. There is no need for an address operator like in C.

Here's an example of passing by reference:

```
function foo(ref x : integer)
{
    x := 10;
};


var y;

main
{
    foo(y);
};
```

After the call to *foo*, the variable *y* will contain 10.

Reference parameters that are defaults (e.g. optional) can only have defaultsof *null*. You can test to see if an optional parameter is not supplied by using the *ismissing* function declared inside *lang.p++*.

You cannot pass arrays as reference variable types, instead, pass the array, and an index as normal parameter; or assign the array element to a var and pass that.

8.4.2 Using reference variables

Whenever you try to use a variable that is a reference variable (such as *x* in the example above), the value of the variable will be retrieved, and not the address.

When you pass a reference variable to a function that requires a reference variable, a reference to the reference variable is not passed, but rather the reference to the original variable being referenced is passed! This behaviour is shown in the following example.

```
function A(ref x)
{
};

function B(ref x)
{
   A(x);
};

var y;

main
{
   B(y);
};
```

8.4.3 Optional parameters

The = operator can be places after the type of a variable, and lets you specify optional default values for a parameter.

8.4.4 Example of optional parameters

```
include “io.p++”;

function foo(x = 1, y = 2)
{
    write(x);
    write(y);
};

main
{
    foo();
    foo(10);
    foo(10, 20);
    foo(, 20);
};
```

Program output:

```
1
2
10
2
10
20
1
20
```

8.4.5 Example of optional float and string parameters.

```
function foo(x : float = 3.14, s : string =
“hello”)
{
    write_float(x);
    println(s);
};

main
{
    foo(, “pok”);
};
```

Program output:

```
3.14
pok
```

8.6 Special functions

In P++, there is a special reserved function called *static_init*. You can declare any function called *static_init* anywhere in your source code, and this function will be automatically run before any functions following it. This is similar to Java's *static blocks* and allows you to do advanced initialisation of variables such as arrays.

The signature for the *static_init* function is shown in example 8.4.1.

8.6.1 Example of static_init

Here is an example of using *static_init*.

```
include "array.p++";

var xx : integer[..];

function static_init() : integer
    var i;
{
    for i := 0 to 10
    {
        xx[0] := i;
    };
};

main
{
    write(xx[10]);
};
```

Program output:

```
10
```

In the above example, even thought the function *static_init* was never explicitly called, the array xx was filled with the numbers from 0 to 10. This is because *static_init* is automatically called by P++.

Another example of using *static_init* can be found in the time library *time.p++*.

8.7 Embedded functions and scoping

P++, like PL/0 allows embedding of functions. You can declare functions inside functions, but before the statement blocks (like variables).

```
function A()
 var a, b;
  function B()
     var c, d;
  {
  };
{
};

var x, y;

function C()
{
};
```

In the above example, B is an embedded function of A. A and C are both sibling functions.

Here are the scopes for the above example:

- A, a, b, c, d are visible from B.
- a, B are visible from A.
- A, x, y are visible from C.

Essentially, functions can see all sibling and parent functions, but not children of sibling functions. This rule also applies to variables.

8.8 Returning from functions and procedures

In P++, you can return from functions any time you want, without specifying a return value. The default *return value* (usually 0) is returned. This is different from C, and is made possible because in P++, the space for the return value is allocated on the stack before a function is called, and is available throughout the lifetime of a function call.

8.8.1 Setting the return value with retval

To set the return value you want, you can use the "retval variable, which actually a variable that exists in every function. "retval" is of the same type as the return type of the function, and it's address is the address of the return value space for function. You can use retval like any other variable, this means you can change the return value anywhere in the code for your function.

Note: retval will also be available in embedded functions.

Here's an example of setting the return value using the retval reserved variable.

```
function write_plusone(x : integer) : integer
{
    retval := x + 1;
};
```

8.8.2 Returning with exit

The first way to return from a function is by using the exit statement. P++'s exit statement is very powerful, and allows termination of almost all blocks of code. In this case, you can use exit to exit from a function or procedure. Exit only exits from the function, and does not give you the opportunity to set a return value.
7
Here are some examples of using *exit* to exit functions and procedures.

```
procedure write_plusone(x : integer)
{
    exit procedure;
};

function write_plusone(x : integer) : integer
{
    exit function;
};
```

8.8.3 Returning with return

The second and final way to return from a function is to use the return statement. The return statement allows you to set the return value, and exit in one statement (basically a coupling of 5.2.1 and 5.3.2).

Here's an example of using return.

```
function write_plusone(x : integer) : integer
{
    return write(x + 1);
};
```

8.9 Function pointers and high order functions

All P++ functions can be passed to other functions, or returned from functions. Functions that take other functions, or return functions are considered to be high order functions.
The rules for passing and returning functions are simple. You can pass or return any function as long as the signature of the function expected is the same. That is, the arguments and return types match.

8.9.1 An example of high order functions

```
1:  include "io.p++";
2:
3:  function takes_one(x);
4:
5:  function A(x)
6:  {
7:     write(x);
8:  };
9:
10: function high_order(x : takes_one) : takes_one
11: {
12:     x(100);
13:
14:     retval := x;
15: };
16:
17: main
18: {
19:     high_order(A)(101);
20: };
```

Program output:

```
100
101
```

Program Explanation

3: A function called takes_one is decalred with no body.

5: A function with the same signature as takes_one called A is declared. It takes one integer, and writes it out.

10: A function called high order is decalred to take a function of the type takes_one (or any function with the same signature as takes_one). The function is also declared to return a function of the same type as takes_one.

12: The function high_order makes a call to the function it was passed, and passes that function the number 100.

13: The function high_order sets it's return value to be the function it was given. In other words, it returns the function it was given.

19: The function high_order is called, it is passed the function A, and then the function that is returned from high_order is passed the number 101.

8.9.2 Working with function pointers

Just like functions can have function as a return type, normal variables can have a type of a function. For example:

```
var myfunc : takes_one;

myfunc := A;
```

This code above code is based on example 5.6.1. myfunc is a variable, that is a function pointer, and that now points to the function A.

Line 19 of the sample, can then be modified to pass myfunc rather than A to the high_order function.

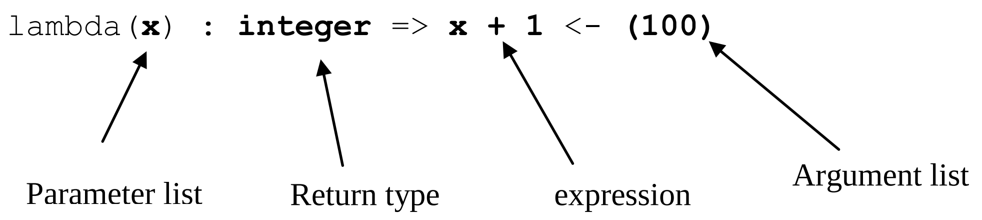
```
19:     high_order(myfunc)(101);
```

8.10 Lambda expressions

Lambda expressions are basically anonymous functions, and they allow for extremely powerful generalization of functions. Lambda expressions help encourage function reuse, and many demonstrations of this can be seen in the string libraries string.p++.

8.10.1 Declaring a lambda expression

Lambda expressions are simple to declare, they can be immediately evaluated, or not evaluated and passed as a function pointer.



```
lambda(x) : integer => x + 1 <- (100)
```

The above lambda expression will result in the value 101. The power to lambda expressions comes when you don’t pass in an argument list (missing out the `<- (100)`), the result of the expression then will be a function pointer to a function that represents the lambda expression. This function pointer can be passed and returned to functions.

8.10.2 An example of lambda expressions

Note: This example makes use of arrays, which will be covered in the next section

```
function char_char_fn(char : character) : character;

/*
 * Transforms a string on a character - character basis.
 *
 * @returns The passed string transformed.
 */
function string_transformer(s : string, cc :
char_char_fn) : string
     var i, len;
{
     len := string_length(s);

     for i := 0 to len - 1
     {
          s[i] := cc(s[i]);
     };

     return s;
};
```

This function, string_transformer, takes a string and a function pointer to a function that takes and returns a character. string_transformer then goes through every character in the string, it calls the function it was passed (cc), passing cc every character in the string, and setting every character in the string to the return value of the function call.
Basically, this lets you generally examine each character in a string and change it according to a pattern.

To move all letters in a string across 1 character, you would make a call like this:

```
s := string_transformer("hello", lambda(x : character) :
character => x + 1);
```

After this call, the string s will contain the string "ifmmp" (every character is offset by 1).

8.10.2 ROT-13 encryption with lambda expressions

ROT-13 encryption is a simple encryption for alphabetical characters; it involves rotating all characters in a string by shifting them across 13 positions. The advantage of ROT-13 is that doing ROT-13 encryption on an encrypted string will decrypt it.

Here's the P++ code for doing ROT-13 encryption!

```
s := string_transformer("hello", lambda(c) => ((c - 'a'
+ m_rot_rotations) % 26) + 'a');
```

After this call, the string s will contain "uryyb".

8.10.3 Lambda expressions summary

As you have seen, lambda expressions make writing reusable functions much more practical because it lets you create functions 'on the fly' when and where you want them. To see more examples of functional reuse, see the string.p++ library, many of the string functions are generalised using lambda expressions, resulting in much less code that needs to be written.

8.11 Limitations of function pointers

Unfortunately, there are some limits to function pointers, and lambda expressions. They however aren't serious, but you none the less have to watch out for them. Because P++ allows embedding of functions, it is not possible to statically keep track of the level changes of function pointers. This means, calls on function pointers requires that the function pointers hold pointers to functions at the same level the function pointer.

```
1:  var y; // level 0
2:
3:  function takes_one(x);
4:
5:  var z : takes_one;
6:
7:  function high_order(x : takes_one)
8:  { // level 1
9:      write(x(100));
10:     z := x;
11:     write(z(100)); // error
12: };
13:
14:  function test()
15:  { // level 1
16:      high_order(lambda(x) => x * y);
17:  };
18:
19:  main
20:  { // level 0 (main isn't a function)
21:      y := 2;
22:      test();
23:  };
```

In the above example, the lambda expression being passed at line 12 uses the 'global' variable y (global meaning any variable at a lower level).

At line 9, the call to x(100) is correct, and will work since the function pointer x declared in the same level as the lambda expression (level 1). At line 11, the call to z(100) is incorrect (even though z references the same function as x) because the function pointer z is declared at a different level to the lambda expression.

In the future, it may be possible to dynamically track level information along with function pointers.

9.0 Arrays

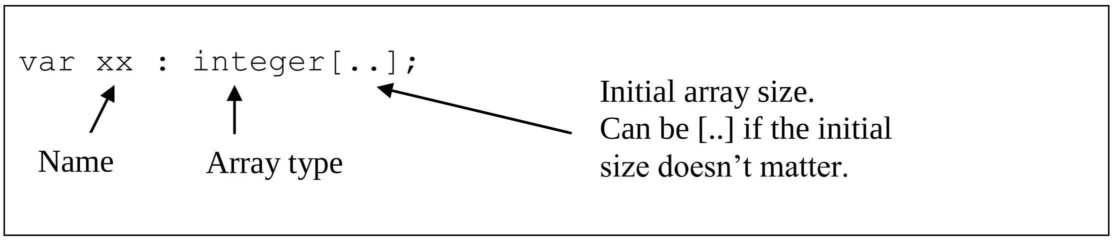
9.1 Introduction to arrays

As described mentioned in section 3.4, arrays in P++ are more advanced than C arrays. This chapter will go into depth arrays, how to use them, how they work, and common functions and operators used with arrays and strings.

Things to remember about arrays:

- Arrays in P++ are always zero based (indexes start at 0).
- Arrays will dynamically grow when you read beyond their boundaries.
- Arrays are automatically garbage collected when no longer used.
- The source file array.p++ contains all the support functions and garbage collection routines.

9.2 Declaring arrays



```
var xx : integer[..];
```

This declares an integer array reference called xx, and immediately assigns it an integer array of an unspecified (default) size. All array elements default to 0 when the array is initialised.

9.3 Getting and setting array elements

As you would have seen by now, putting an index between two square brackets retrieves array elements. The index can be any valid numeric expression (when strings are used, they are cast into integers).

9.2.1 Array getting and setting examples

```
xx[10] := 10;
```

Puts the number 10 inside the 11th element of the array xx.

```
y := xx[5 * 2];
```

Puts the value of the 11th array element of xx into y.

9.4 Aggregate array getting example

Because functions can return arrays, it is possible to actually directly access an array element from an array returned by a function. This is called aggregation; the following program demonstrates this concept.

9.4.1 Example of using arrays and aggregation

```
1:  include "io.p++";
2:  include "array.p++";
3:
4:  function gives_array() : integer[]
5:      var yy : integer[..];
6:  {
7:      yy[0] := 1;
8:
9:     return yy;
10:  };
11:
12:  main
13:  {
14:      write(gives_array()[0]);
15:  };
```

The key statement in this program is at line 14. Notice how the function gives_array() is called, an immediately after, the brackets are used to extract the first element of the array returned by the function. Also notice that each time gives_array() is called, it will return a new array since yy is a private array to gives_array, it is created every time gives_array is called. If yy had been declared before gives_array, gives_array would return the same array every time it is called (consider yy to then be a global variable).

The program will obviously write “1” to standard output.

10.5 Array references

Arrays are always passed and used through references. Whenever an array is passed to a function, returned from a function, or assigned to another reference, it is done with references. This can have implications that you need to be aware of, when working with strings, which will be covered later on in this chapter.

10.5.1 Multiple references example

```
1:   include “io.p++”;
2:   include “array.p++”;
3:
4:   var xx : integer[];
5:   var yy : integer[100];
6:
7:   main
8:   {
9:       xx := yy;
10:      xx[0] := 1;
11:
12:      write(xx[0]);
13:      write(yy[0]);
14:  };
```

Program output:

```
1
1
```

Lets examine the program line by line:

- **1**: Include the io libraries
- **2**: Include the array libraries
- **4**: Declare an array reference “xx” that initially references null (no array). Trying to access xx without first assigning it to anything will cause an error.
- **5**: Declare an array reference “yy” that initially references an array of 100 integers.
- **9**: Make the array reference xx reference the same array as yy.
- **10**: Make the first element of the array to be 1.
- **12**: Write the array’s first element through the xx reference.

- **13**: Write the array’s first element through the yy reference.

As you can see from the program output, both xx and yy reference the exact same array! Changing the array through xx, will affect all variables that reference the array (such as yy).
Notice how at line 4, when you use the := operator between two arrays, you’re actually assigning references, not a copy of the array.

10.5.2 Comparing references

You can compare two array references with the ‘=’ operator; this will let you test to see if two variables (references) refer to the same array.

```
var xx : integer[..];
var yy : integer[];

main
{
    xx := yy
    write(xx = yy); // will output 1 (true)
};
```

10.5.3 Null references

By default arrays that are initialised with no size reference nothing or ‘null’.

```
var xx : integer[];
```

You can check if an array reference is null, or assign an array reference to nothing, by using the reserved keyword *null*, or by using the function *isnull* (defined in *array.p++*).

```
xx := null;

if xx = null then print(“I am null”);

if isnull(xx) then print(“I am null”);
```

Trying to read elements from a null array reference will cause an error to be generated by the interpretor!

11.0 Strings

11.1 Introduction to strings

Strings in P++ are arrays of characters (which are characters). P++ explicit support for strings, so that you can use literal strings inside source code.

Most of the functions relating to strings are defined in the library string.p++.

11.2 Hello World!

Finally! With string support, a hello world program can be written in P++.

```
include "string.p++";

main
{
     print("Hello World!\n");
};
```

The print statement is defined in string.p++, it takes a string and prints it to standard output.

11.3 Declaring strings

11.3.1 Declaring an empty string.

Declaring a string in P++ is like declaring any other type of variable.

```
var s : string; // same as character[..];
```

11.3.1 Declaring a null character array reference

Whenever a string is declared, a new strings is always created. It is not a null reference. If a null reference is needed (you know you'll assign it to an existing string you already have), you can declare a character array reference, or you can assign the string to an initial null value:

```
var s : character[];
var s : string = null;
```

This will be faster because no memory for an array has been allocated, but is dangerous if you don't assign the reference to something before you use the string.

11.4 Defining strings

Once you have declared a string variable, you can work with it exactly like it was an array. You can modify single characters inside a string using the bracket [ ] element notation.

Warning:
Since strings are arrays and always passed by reference, you have to be careful when passing strings to functions, it is entirely possible that the function will modify the string. Either expect your string to be modified, or pass a copy of the string to the function using the new_string_copy function.

11.4.1 Defining strings by character

Strings, like arrays will grow as needed, so it's a simple matter of just working with each character on a one by one basis to fill up the string and not worry about the size.

```
var s : string;

s[0] := 'h';
s[0] := 'i';
```

11.4.2 Defining strings literally

Defining strings character by character gets boring after the second character. P++ lets you define entire strings literally with strings. You must enclose strings inside double quotation marks.

Note: The only limitation on using literal strings, is that you can not assign or pass literal strings to the handle type (no garbage collection is performed on handles).

```
var s : string

s := "hi";
```

11.5 Strings and functions

Strings and functions are superbly supported by P++. Strings can be passed and returned from functions without worrying about the scope of strings. Strings will exist until they are no longer referenced by any variables.

11.5.1 Examples of strings and functions

Here's an example of using strings with functions. This is the print function, takien straight out of string.p++.
The print function takes a string, prints out the string, and then returns the string it was given.

```
1:  /*
2:   * Prints a string to standard output.
3:   *
4:   * @returns The string supplied.
5:   */
6:  function print(s : string) : string
7:      var i, max;
8:  {
9:      if (isnull(s))
10:     {
11:         return s;
12:     };
13:
14:     if ((max := string_length(s) - 1) >= 0)
15:     {
16:         for i := 0 to max
17:         {
18:             write_byte(s[i]);
19:         };
20:     };
21:
22:     return s;
23: };
```

6: The function is declared, it is called print, it takes one argument called s of the type string, and the function is of the type string (returns a string).

7: Two worker variables are declared, i, the for loop counter, and max, the maximum index of the string.

9: The first thing the function does is check at line 9 is check to see if the string supplied is actually just a null reference. If it is, then just return the null/

14: The variable max is assigned to be one less than the length of the string, and also this value of max is then tested to see if it’s more than 0. This is made possible because in P++ assignments can also be used as expressions!

16: If the string actually contains something, then a for loop is used to print out each character in the string as a raw byte to standard output, using the write_byte function.

23: The string that was passed is returned.

12 String operators

12.1 Introduction to string operators

P++ supports some operators than can be performed on strings, this section explains those operators.

12.2 String concatenation

String concatenation is done using the *string_append* function, which takes two strings and appends the second string to the first string. P++ however allows you to use the "+" operator to add two strings or a string and a number together, the compiler requires that array.p++ be included since it also uses integer_array_append function directly.

In P++ with ever operator except for +, when strings and numbers are operands, numbers take priority, and the string is first converted to a number. When you use the + operator, the number is converted into a string, and the two strings are concatenated. This behaviour was a simply design decision I made because I believe that it is more useful to be able to add strings with + and also add strings and numbers (as strings) with +. The s2i function can be used to explicitly convert a string to an integer if it is needed.

12.1.1 Examples of concatenation of strings

Assume s is a string.

Operation	**Result**
`s := "hello" + " world";`	`"hello world"`
`s := 1 + "2";`	`"12"`
`s := "1" + 2;`	`"12"`
`s := 1 + s2i("2");`	`"3"`
`s := s2i("2") + 1;`	`"3"`

12.3 String comparisons

Strings comparisons can be performed with the *string_compare* function; however, P++ lets you use the standard mathematical comparison operators to compare strings.

12.3.1 Table of string comparators

Operator	**Description**
>	More than.
<	Less than.
>=	More than or equal to.
<=	Less than or equal to.
$=	Equal to.

These operators use the string function, *string_compare*, to compare each character in the two strings, which are being operated on. A more than or less than comparion done on a string works in the same way as it does in C and BASIC, each character is compared, and the ASCII code of each character is used to determine which string is smaller or larger.

Note: If you use the = operator on two strings, it will compare string *handles*, not each character in the string. So it can tell you if two variables point to the same string, but not if two variables point to strings which are identical character-for-character.
See section 10.5.2 on comparing array references.

12.3.2 Sample on comparing strings

```
var s1 : string = "h";
var s2 : string = "i";
var s3 : string = "h";

main
{
    write(s1 < s2);  // "h" <  "i"
    write(s1 = s3);  // "h" =  "h"
    write(s1 $= s3); // "h" $= "h"
};
```

Program output:

```
1
0
1
```

13.0 Type casting in P++

13.1 Introduction to type casting

In P++, there is a lot of implicit type casting in order to make P++ easy to use. This makes P++ a relatively weakly typed language. This section will explain how the P++ compiler will handle casting requirements, and how you can manually do casting with the inbuilt functions.

13.2 Generic implicit casting

P++ will automatically do implicit casts whenever you try to use any type as another type.
For example, if you try to assign a string to an integer, the string will be cast into an integer. If you try to assign an integer to a string, the integer a new string will be created containing a string representation of the integer. Similar casting will be done between integers and floats.

13.2.1 Implicit casting table

This table shows all the types that will be implicitly cast by P++.

FROM	TO
integer	float
float	integer
string	float
float	string
string	integer
integer	string

13.3 Implicit casting with strings

The following source code demonstrates some typical casting between strings and integers. The same applies for strings and floats.

```
var x;
var s : string;

main
{
    /* s = "", x = 0 */

    s := "100";

    /* s = "100", x = 0 */

    x := s * 2; // cast from string to integer

    /* s = "100", x = 200 */

    s := x; // cast from integer to string

    /* s = "200", x = 200 */
};
```

Remember that implicit casting also affects functional arguments. For example, the string function print, prints out a string to standard output. But if you supply it an integer, the integer will automatically be cast to a string before passing it to print.

```
var x := 100;

A: print(“hello”);             // “hello”

B: print(x);                   // “100”

C: print(“hello ” + x);        // “hello100”

D: print(x + x + “hello”);     // “200hello”

E: print(“hello” + x + x);     // “hello100100”

F: print(“hello” + (x + x)); // “hello200”
```

The above example uses the print statement that requires a string. If a number or expression that results in a number is passe, then the compiler will automatically generate code to do dynamic casting to a string and then pass that string to print.

A: Passing a string to print is fine.

B: Passing an integer to print, the integer will be converted to a string containing “100”.

C: Adding a string and an integer, the integer will be converted to a string, and then appended to the end of the string. This produces the string “hello1000”.

D: The expression (x + x) is evaluated first, which comes out to the number 200, this number is then added with the string “hello”. So the number 200 is converted into the string “200” and then is pre-pended to the string “hello”. This produces the string “200hello” being sent to print.

E: The expression (“hello” + x) is evaluated first, so the number x (100) is converted into the string “100” and is appended to the string “hello”. This results in the string “hello100”. The next expression will be “hello100” + x, and by the same process as the previous expression, the string “hello” is appended to make “hello100100”.

F: The expression (x + x) is evaluated first, which results in the number 200. This number is then converted to the string “200” so it can be appended to the string “hello”. This produces the string “hello200”.

13.4 Implicit casting with floats

When doing mathematical operations between floats and integers, all operations will be carried out in floating point if necessary to keep floating point precision.

Code 8.4.1

```
var i : integer;

i := 2
i := i * 1.5;
```

In the above example, *i* will become 3, because the operation (2 * 2.5) is carried out in floating point, then the float is converted to an integer and put back into i.

Code 8.4.2

```
var i : integer;

i := 10;

println(i / 3);       // A

println(i2f(i) / 3); // B

println(i / 3.0);    // C
```

In the above example, there are 3 println statements. The println function requires a string, but because of implicit casting, it's perfectly valid to pass println a number.

A: The expression `(i / 3)` is passed to println. Since both *i* and *3* are integers, the division is done using integer division with truncation – therefore the result is the number 3, which is converted to a string and then sent to println.
The output is "3".

B: The expression `(i2f(i) / 3)` is passed to println. In this case, *i* is converted into a floating point number using the i2f casting function. Because *i* is a float, the integer 3 is implicitly casted into a float, and the division is carried out in floating point. The floating point number 3.333' is converted to a string which is sent to println.
The output is "3.333".

C: The expression `(i / 3.0)` is passed to println. This time, *i* is still an integer, but *3.0* is explicitly declared as a floating point number by putting in the decimal point. Since 3.0 is a float, the value of *i* is implicitly cast into a float and the division is carried out in floating point.
The output is again "3.333".

13.5 Explicit casting and conversion functions

math.p++, lang.p++ and string.p++ all define functions for casting between different data types.
These functions are typically named x2y, where x is what is being cast from (this is the argument to the function), and y is the destination type (this is the return type & value of the function).

Math function	**Description**
`round`	Converts a float into an integer with rounding.
Lang function	
`i2f`	Converts an integer to a float.
`f2i`	Converts a float into an integer.
`nocast_i2f`	Puts an integer into a float bit for bit.
`nocast_f2i`	Puts a float into an integer bit for bit.
String function	
`c2s`	Converts a character into a string.
`i2s`	Converts an integer into a string (you can specify the radix/base, which is 10 by default).
`f2s`	Converts a float into a string.
`hex`	Converts an integer into a hexidecimal string.
`oct`	Converts an integer into an octal string.

14.0 Array implementation

14.1 Array implementation introduction

This section covers my implementation of arrays in P++.

14.2 Arrays with the P++ machine

The P++ machine supports using portions of dynamic memory using the basic instructions OC_MAL, OC_FRE, OC_RAL and OC_MSZ. These functions take and return sizes of memory in chunks of integers. The P++ machine garuntees that any rallocation of memory (OC_RAL) will not invalidate any handles given out by (OC_MAL). This allows arrays in P++ to dynamically grow with no troubles at all. The P++ machine does this by handing out handles (pointers to pointers), and not direct pointers to memory, so that if the pointer changes, the P++ program doesn't need to know.
This is all that's made available by the P++ machine. I chose to implement garbage collection using reference counting, and I handle all the boundary checks and garbage collection all using the P++ language (not relying on the interpreter).

14.3 Array header anatomy

Every array allocated has a 4 byte header, so when OC_MAL is called, it is always given the size of the array + 4.
Here is the anatomy of the arrays in memory (sizes are all in integers not bytes).

Length	RefCount	Res	Res	Array data

Length takes up one integer, and holds the length of the array.
Refcount takes up one integer and holds number of references refer to the array.
Res are reserved for future use.
Array data is the space taken up by the array.

14.4 Analysis of a sample array program

This section analyses the instructions and program stack for a simple program that uses arrays.

14.4.1 Sample program using arrays

```
include "array.p++";

var xx : integer[..];

main
{
     xx[0] := 10;

     write(xx[0]);
};
```

14.4.2 Commented P++ instructions for the array sample

This is a partial list of the instructions generated by the program. It represents the portion of code that makes up the *main* block (which includes tha variable initialisation).
Most of the instructions generated because of the support functions in *array.p++* have been removed.
All numbers in the left column are in hexadecimal.

493: INC 0x0 0x4	Increment the stack by 4. 3 for the offset, and 1 for the return value.
494: LIT 0x0 0x4	Load 4 (the array header size).
495: MAL 0x0 0x3	Allocate 4 integers of memory and store the handle to this memory in the variable xx (address 3).
The array is about to have it's reference count incremented.	
496: LIT 0x0 0x0	Load up 0. This actually allocates return space for the upcoming function call.
497: LOD 0x0 0x3	Load up the memory handle xx.
498: STP 0x0 0x3	Store the memory handle as a parameter.
499: CAL 0x0 0x65	Call the function __array_addref. This function increments the reference count of the array.
49a: DEC 0x0 0x1	Decrement 1 off the stack. This gets rid of the return value from __array_addref.

The array is about to undergo a boundary check.	
49b: LIT 0x0 0x0	Load up 0. This actually allocates return space for the upcoming function call.
49c: LIT 0x0 0x0	Load up 0, this is the index of the array we want to access.
49d: LOD 0x0 0x3	Load up the memory handle xx.
49e: SLD 0x0 0x2	Load up a copy of the index of the array we want to access, this is 2 positions down.
49f: STP 0x0 0x3	Store the index as a parameter.
4a0: STP 0x0 0x3	Store the array handle (xx) as a parameter.
4a1: CAL 0x0 0xf3	Call the __array_check function to ensure the array size is ok for a access to the index.
4a2: DEC 0x0 0x1	Decrement 1 off the stack. This gets rid of the return value from __array_check.
The array is about to have the number 10 put in index 0.	
4a3: LIT 0x0 0x4	Load 4 (the array header size).
4a4: OPR 0x0 0x1010	Do an add operation on the array header and the index (which is just under it on the stack at this point). This gives us the real memory offset so we can pass it to the P++ machine.
4a5: LIT 0x0 0xa	Load up 10, the value we want to store in the array at index 0.
4a6: SWS 0x0 0x0	Swap the memory offset and the value to be stored.
4a7: POP 0x0 0x3	Store the memory offset into the CX register (3) and pop it off the stack.
4a8: PUS 0x0 0x4	Push onto the stack the current value of the DX register (4).
4a9: SWS 0x0 0x0	Swap this value of DX with the value to store (10).
4aa: RIO 0x4 0x4	Do a bitwise OR on the DX register (4) with the memory mode flag (4) to tell the P++ machine all operations from now on will be done with memory.
4ab: SLD 0x0 0x0	Make a copy of the value to store (10).
4ac: STO 0x0 0x3	Store this value (10) as a parameter.
4ad: SWS 0x0 0x0	Swap the other copy of the value (10) with the item underneath it (the backup of DX previously pushed).

	The backup of DX is now on top.
4ae: POP 0x0 0x4	Store the original value of DX back into the DX register (4).
4af: DEC 0x0 0x1	Decrement the stack to get rid of the copy of the value to be stored we previously SLDed. This copy was never used, but was loaded because the xx[0] := 10 is an expression which returns 10! But we never assigned it to anything. It would have been valid to go (a := xx[0] := 10).
The number 10 has been stored in the array at index 0. The number 10 at array index 0 is now going to be retreived and printed out.	
4b0: LIT 0x0 0x0	Load up 0. This actually allocates return space for the upcoming function call to write.
4b1: LIT 0x0 0x0	Load up 0. This actually allocates return space for the upcoming function call to __array_check.
4b2: LIT 0x0 0x0	Load up 0, the index of the array.
4b3: LOD 0x0 0x3	Load up the array handle (xx).
4b4: SLD 0x0 0x2	Load up a copy of the index of the array.
4b5: STP 0x0 0x3	Store the array index as a parameter.
4b6: STP 0x0 0x3	Store the array handle as a parameter.
4b7: CAL 0x0 0xf3	Call the __array_check function to ensure the array size is ok for a access to the index.
4b8: DEC 0x0 0x1	Decrement 1 off the stack. This gets rid of the return value from __array_check
4b9: LIT 0x0 0x4	Load 4 (the array header size).
4ba: OPR 0x0 0x1010	Do an add operation on the array header and the index (which is just under it on the stack at this point). This gives us the real memory offset so we can pass it to the P++ machine.
4bb: SLD 0x0 0x0	Make a copy of this memory offset.
4bc: POP 0x0 0x3	Store the memory offset into the CX register (3) and pop it off the stack.
4bd: PUS 0x0 0x4	Push onto the stack the current value of the DX register (4).
4be: RIO 0x4 0x4	Do a bitwise OR on the DX register (4) with the memory mode flag (4) to tell the P++ machine all operations from now on will be done

	with memory.
4bf: LOD 0x0 0x3	Load up the element of the array (xx) at the memory offset stored in CX.
4c0: SWS 0x0 0x0	Swap the array handle with the original value of DX.
4c1: POP 0x0 0x4	Pop the original value of DX back into the DX register.
4c2: SWS 0x0 0x0	Swap the element of the array just loaded, with the copy of the memory offset.
4c3: DEC 0x0 0x1	Get rid of the copy of the memory offset.
4c4: STP 0x0 0x3	Store the array element now retrieved (10) as a parameter for the write statement.
4c5: CAL 0x0 0x17	Call the write statement to write this element (10).
4c6: DEC 0x0 0x1	Decrement 1 off the stack. This gets rid of the return value from write.
4c7: LIT 0x0 0x0	Load up 0. This actually allocates return space for the upcoming function call to write.
4c8: LOD 0x0 0x3	Load up the array handle (xx).
4c9: STP 0x0 0x3	Store the array handle (xx) as a parameter.
4ca: CAL 0x0 0x7e	Call the __array_release function to release the array. Since there was ever only one reference to the array xx, it is now deallocated automatically by __array_release.
4cb: DEC 0x0 0x1	Get rid of the return value from __array_release.
4cc: OPR 0x0 0x0	End the data segment. This is also the end of the program.

Because of the amount of code generated, a stack trace for this program would be futile, a better understanding of how the stack changes over time, can be seen in section 15.2.3.

15.0 Which statement implementation

This section analyses a simple which statement.

15.1 Which statement implementation introduction

The implementation of which statements is very simple, and is best understood by following the sample program.

15.2 Analysis of a sample which statement program

15.2.1 Sample program using a which statement.

```
include "io.p++";

main
{
            which (5 + 5)
            {
                 case > 55 || < 8:

                       write(1);

                 case 10:

                       write(2);
            };
};
```

15.2.2 Commented P++ instructions for the which sample

All numbers in the left column are in hexadecimal.

24: INC 0x0 0x3	Load up 3, this is the datasegment overhead.
25: LIT 0x0 0x5	Load up 5.
26: LIT 0x0 0x5	Load up 5.
27: OPR 0x0 0x1010	Add the previous 2 5s, the stack now holds 10.
28: LIT 0x0 0x0	Load up 0.
29: SLD 0x0 0x1	Load up the which expression from 1 down in the stack (10).
2a: LIT 0x0 0x37	Load up 55.
2b: OPR 0x0 0x1120	Do a 10 > 55 comparison. This is 0.

2c: OPR 0x0 0x1160	Do a bitwise OR with the answer from 10 > 55 with the previously loaded 0.
2d: SLD 0x0 0x1	Load up the which expression from 1 down in the stack (10).
2e: LIT 0x0 0x8	Load up 8.
2f: OPR 0x0 0x1100	Do a 10 < 8 comparison. This is 0.
30: OPR 0x0 0x1160	Do a bitwise OR with the answer from 10 < 8 with the answer from 10 > 55. This is 0 | 0, e.g. 0.
31: JPC 0x0 0x38	Jump to the next case if this case was 0, which it is.
32: LIT 0x0 0x0	Load up 0 for the return value of write.
33: LIT 0x0 0x1	Load up 1.
34: STP 0x0 0x3	Store 1 as a parameter for write.
35: CAL 0x0 0x17	Call write.
36: DEC 0x0 0x1	Get rid of the return value from write.
37: JMP 0x0 0x44	Jump to the end of the which statement.
38: LIT 0x0 0x0	Load up 0.
39: SLD 0x0 0x1	Load up the which expression from 1 down in the stack (10).
3a: LIT 0x0 0xa	Load up 10.
3b: OPR 0x0 0x1080	Do a 10 = 10 comparison. This is 1.
3c: OPR 0x0 0x1160	Do a bitwise OR with the answer from 10 > 55 with the previously loaded 0.
3d: JPC 0x0 0x44	Jump to the end of the which statement if the case was 0, which it isn't.
3e: LIT 0x0 0x0	Load up 0 for the return value of write.
3f: LIT 0x0 0x2	Load up 2.
40: STP 0x0 0x3	Store 2 as a parameter for write.
41: CAL 0x0 0x17	Call write.
42: DEC 0x0 0x1	Get rid of the return value from write.
43: JMP 0x0 0x44	Jump to the end of the which statement.
44: DEC 0x0 0x1	This is the end of the which statement, which decrements the which expression, which was previously loaded for all the cases to compare to.
45: OPR 0x0 0x0	End the data segment. This is also the end of the program.

15.2.3 Commented stack trace for the which sample

This is a stack trace of the program. All the numbers in the trace are in hexadecimal.

```
0x0 0x0 0x0
0x0 0x0 0x0 0x5
0x0 0x0 0x0 0x5 0x5           5 + 5 is added to get 10
0x0 0x0 0x0 0xa               the first case statement starts
0x0 0x0 0x0 0xa 0x0
0x0 0x0 0x0 0xa 0x0 0xa
0x0 0x0 0x0 0xa 0x0 0xa 0x37  10 < 55 is tested here
0x0 0x0 0x0 0xa 0x0 0x0
0x0 0x0 0x0 0xa 0x0
0x0 0x0 0x0 0xa 0x0 0xa
0x0 0x0 0x0 0xa 0x0 0xa 0x8   10 < 8 is tested here
0x0 0x0 0x0 0xa 0x0 0x0
0x0 0x0 0x0 0xa 0x0
0x0 0x0 0x0 0xa               the second case statement starts
0x0 0x0 0x0 0xa 0x0
0x0 0x0 0x0 0xa 0x0 0xa
0x0 0x0 0x0 0xa 0x0 0xa 0xa   10 = 10 is tested here
0x0 0x0 0x0 0xa 0x0 0x1       the result was 1 (true)
0x0 0x0 0x0 0xa 0x1
0x0 0x0 0x0 0xa
0x0 0x0 0x0 0xa 0x0           0 is loaded for the return value
0x0 0x0 0x0 0xa 0x0 0x2       2 is loaded and stored as a param
0x0 0x0 0x0 0xa 0x0
0x0 0x0 0x0 0xa 0x0
0x0 0x0 0x0 0xa 0x0 0x1 0x1 0x40 0x2
0x0 0x0 0x0 0xa 0x0 0x1 0x1 0x40 0x2 0x2
0x0 0x0 0x0 0xa 0x0 0x1 0x1 0x40 0x2 0x2 0x2
0x0 0x0 0x0 0xa 0x0 0x1 0x1 0x40 0x2 0x2
0x0 0x0 0x0 0xa 0x2 0x1 0x1 0x40 0x2
0x0 0x0 0x0 0xa 0x2 write has returned 2
0x0 0x0 0x0 0xa the return is disgarded
0x0 0x0 0x0 0xa get rid of the which expression
0x0 0x0 0x0
```

0x1 static link, 0x1 dynamic link, 0x40 return address.

two copies of 0x2 from the parameter are loaded and the first 2 is written, the second is returned.

Appendex A1 – Language Functions

These language functions define the basic core functionality of P++ that relate deeply to the language. They are defined in the file *lang.p++*.

i2f

```
function i2f(x : integer) : float
```

Casts an integer into a float and returns it.

f2i

```
function f2i(x : float) : integer
```

Casts a float into an integer and returns it.

nocast_i2f

```
function nocast_i2f(x : integer) : float
```

Returns an integer in a float bit-for-bit, without casting.

nocast_f2i

```
function nocast_f2i(x : float) : integer
```

Returns a float in an integer bit-for-bit, without casting.

beep

```
function beep() : integer
```

Generates a beep through the system speaker.

terminate

```
function terminate() : integer
```

Terminates the program immediately.

addressof

```
function addressof(ref h : handle) : integer
```

Returns the address of any variable passed.

isnull

```
function isnull(h : handle) : boolean
```

Returns true if an array reference, integer or pointer is null (0).

ismissing

```
function ismissing(ref h : handle) : boolean
```

Returns true if an optional *reference* parameter is missing.

Appendex A2 – Extra Language Functions

These language functions are defined in the file *langs.p++* and they require both array and string libraries, so are separated from *lang.p++*.

command_line

```
function command_line() : string
```

Returns a string containing the command line arguments for the process.

Appendix A3 – Standard IO Functions

These standard IO functions are for working with the P++'s machines simple IO functions for writing primitive types. More advanced ouput functions (written in P++) can be found amongst the string functions (See Libs F). The functions are defined in the file *array.p++*.

write

```
function write(x : integer) : integer
```

Writes an integer to standard output and returns the integer.

write_byte

```
function write(x : integer) : integer
```

Writes a literal byte (stored in an integer) to standard output and returns the byte.

write_integer

```
function write(x : integer) : integer
```

Writes an integer to standard output and returns the integer.

write_float

```
function write(x : integer) : integer
```

Writes an float to standard output and returns the float.

read_byte

```
function read_byte() : integer
```

Reads a literal byte from standard input and returns it in an integer. Use the *readln* function in *string.p++* if you want to read more advanced types like integers and strings.

Appendix A4 – Math Functions

These math functions can do advanced mathematical functions (such as sin/cos). The functions are defined in the file *array.p++*.

round

```
function round(x : float) : integer
```

Converts a floating-point number into an integer with rounding.

factorial

```
function factorial(x : integer) : integer
```

Calculates the factorial of an integer and returns it.

power

```
function power(x : float, y : integer) : float
```

Calculates x raised to the power of y and returns it.

randomize

```
function randomize(seed : integer) : integer
```

Seeds the random number generator with the specified seed and returns the seed.

random

```
function random(max : integer) : integer
```

Generates a pseudo random number between 0 and max (inclusive) and returns it.

cos

```
function cos(x : float) : float
```

Calculates the cosine of x using the taylor/malcorian series.

sin

```
function sin(x : float) : float
```

Calculates the sine of x using the taylor/malcorian series.

tan

```
function tan(x : float) : float
```

Calculates the tangent of x using the taylor/malcorian series.

Appendex A5 – Array Functions

These array functions are for working with arrays. The functions are defined in the file *array.p++*.

array_length

```
function array_length(array_handle : handle) : integer
```

Returns the length of an array.

array_snip

```
function array_snip(array_handle : handle, length) :
integer
```

Reallocates an array to the specified length. Snipping off the end of the array if the lenth is smaller than the current length, or increasing the size of the array if the new length is longer.

new_array

```
function new_array(length = 0, element_size = 1) : array
```

Dynamically creates and returns a array of any type with the specified length. The default length is 0. You can also specify the size of each element, which the array holds, this size is in integers (32bits), the default is 1 integer.

new_integer_array

```
function new_integer_array(length = 0) : integer[]
```

Dynamically creates and returns a new integer array with the specified length. The default length is 0.

new_float_array

```
function new_float_array(length = 0) : float[]
```

Dynamically creates and returns a new float array with the specified length. The default length is 0.

new_array_copy

```
function new_array_copy(source : array) : array
```

Creates a new copy of the given array. The given array can be any type of array with an element size of 1 integer.

new_integer_array_copy

```
function new_integer_array_copy(source : integer[]) :
integer[]
```

Dynamically creates a new copy of the given integer array.

new_float_array_copy

```
function new_float_array_copy(source : float[]) : float[]
```

Dynamically creates a new copy of the given float array.

array_append

```
function array_append(destination : array, source :
array) : array
```

Appends one array to another array and returns the destination array.

integer_array_append

```
function integer_array_append(destination : integer[],
source : integer[]) : integer[]
```

Appends one integer array to another array and returns the destination array.

float_array_append

```
function float_array_append(destination : float[], source
: integer[]) : float[]
```

Appends one float array to another array and returns the destination array.

array_compare

```
function array_compare(s1 : array, s2 : array) : integer
```

Returns a negative number is s1 is less than s2, a positive number if s1 is more than s2 and 0 if s1 and s2 are identical element-for-element.

integer_array_compare

```
function integer_array_compare(s1 : integer[], s2 :
integer[]) : integer
```

Compares two integer arrays. See array_compare.

float_array_compare

```
function float_array_compare(s1 : float[], s2 : float[])
: integer
```

Compares two float arrays. See array_compare.

Appendix A6 – String Functions

These string functions are for working with strings. The functions are defined in the file *string.p++*.

new_string

```
function new_string(length = 0, c : character = ' ') :
string
```

Returns a new string with the given length. The string will be filled up to the length, with the given character. The default length is 0, the default initial character to fill with is a space.

new_string_copy

```
function new_string_copy(s : string) : string
```

Returns a copy of the supplied string *s*.

string_length

```
function string_length(s : string) : integer
```

Returns the length of the supplied string *s*.

string_append

```
function string_append(destination : string, source :
string) : string
```

Appends the source string onto the destination string and then returns the destination string.

string_find_char

```
function string_find_char(s : string, c : character) :
integer
```

Finds the first occurance of a character and returns the zero based index of the character in the string.

string_reverse_find_char

```
function string_reverse_find_char(s : string, c :
character) : integer
```

Finds the last occurance of a character and returns the zero based index of the character in the string.

print

```
function print(s : string) : string
```

Prints a string to standard output.

You can use the print statement in place of the traditional write statements from *io.p++*. P++’s implicit casting system will cast floats, integers etc into strings for you if you pass them to print.

println

```
function println(s : string = "") : string
```

Prints a string and a newline to standard output.

read

```
function read() : string
```

Reads one character from standard input and returns a string containing that character.

readln

```
function readln() : string
```

Reads a line from standard input and returns a string containing that line.

You can use this to read integers from standard input by just assining the result of *readln* into an integer. P++’s implicit casting system will convert the string returned into an integer for you!

reverse

```
function reverse(s : string) : string
```

Reverses a string, and returns the same string reversed.

hex

```
function hex(number : integer) : string
```

Converts an integer into a hexadecimal string and returns it.

oct

```
function oct(number : integer) : string
```

Converts an integer into an octal string and returns it.

bin

```
function bin(char : character) : string
```

Converts an integer into a binary string and returns it.

c2s

```
function c2s(char : character) : string
```

Converts a character into a string (returns a one length string with the character).

f2s

```
function f2s(f : float, dp = 3) : string
```

Converts a floating point number into a string and returns it. You can specify how many decimal places to print.

i2s

```
function i2s(number, radix = 10) : string
```

Converts an integer into a string. You can specify a radix (the base of the number). The default radix is 10.

```
Example:

s := i2s(100); // s will become "100"
```

f2s

```
function f2s(number, dp = 3) : string
```

Converts an integer into a string. You can specify the number of decimal points to convert. The default radix is 3.

```
Example:

s := f2s(100.5); // s will become "100.500"
```

c2s

```
function c2s(char : character) : string
```

Converts a character into a string (returns a one length string with the character).

a2A

```
function a2A(char : character) : character
```

Converts a lower case character into an uppercase character.

A2a

```
function A2a(char : character) : string
```

Converts an uppercase character into a lowercase character.

snip

```
function snip(s : string, length) : string
```

Reduces or grows a string to a specified size preserving the data.

string_transformer

```
function string_transformer(s : string, cc :
char_char_fn) : string
```

Takes a string, and changes every single character in the string using the suppled function, then returns the same string transformed.
The function that is passed needs to take one character, and return a character. The *string_transformer* function will call this function and pass it every character in the string, the function should do some sort of transformation on that character and then return the new character.

string_filter

```
function string_filter(s : string, cb : char_bool_fn) :
string
```

Filters characters out of a string and then returns the same string, but filtered. You need to pass this function a function that takes a character, and returns a boolean. The *string_transformer* function will call this function passing it every character in the string, and the function should return *true* if the character should be kept in the string, or return *false* if the character should be filtered out.

The following statement will set s to be the string "ello".

```
s := string_transformer("hello", lambda(x) => x = 'h');
```

string_filter_char

```
function filter_char(s : string, c : character) : string
```

Filters out the given character *c* from the string *s* and returns the string *s*.

string_limited_filter

```
function string_limited_filter(s : string, start_index,
end_index, cb : char_bool_fn, cb_stop : char_bool_fn) :
string
```

Like string_filter, except you can choose to stop filtering at anytime. To stop filtering, the second function passed, *cb_stop* should return *false* when it is given a character it identifies as the stopping char.
See the source code for *trim* to see an example.

to_upper

```
function to_upper(s : string) : string
```

Converts a string to uppercase and returns the same string in uppercase.

to_lower

```
function to_lower(s : string) : string
```

Converts a string to lowercase and returns the same string in lowercase.

substring

```
function substring(s : string, start_index, length = -1)
: string
```

Returns a new string that is the part of the string *s* starting at the *start_index* and carrying on the the specified *length*. If you don't specify the length of the substring to return, then a substring from the *start_index* to the end of the string is returned.

The following statement will set s to be the string "llo".

```
s := substring("hello", 2);
```

```
```

left

```
function left(s : string, length) : string
```

Returns a new string that returns the left of a string for a specified length.

The following statement will set s to be the string “he”.

```
s := left(“hello”, 2);
```

right

```
function right(s : string, length) : string
```

Returns a new string that returns the left of a string for a specified length.

The following statement will set s to be the string “lo”.

```
s := right(“hello”, 2);
```

string_compare

```
function string_compare(s1 : string, s2 : string) :
integer
```

Compares two strings. Returns a negative number is s1 < s2, a positive number if s2 > s2 and 0 if s1 = s2.

ltrim

```
function ltrim(s : string, c : character = ' ') : string
```

Trims a specified character from the left of a string and returns the same string, but trimmed. If you don’t specify the character to trim, the default is a space. Trimming stops when a different character is encountered.

rtrim

```
function rtrim(s : string, length) : string
```

Same as ltrim, but trims all characters from the right.

trim

```
function trim(s : string, c : character = ' ') : string
```

Same as ltrim and rtrim together. Trims a specified character from the left and right of a string and returns the same string back, but trimmed.

isnumber

```
function isnumber(c : character) : Boolean
```

Returns true if the character is a number (in ASCII).

isletter

```
function isletter(c : character) : boolean
```

Returns true if the character is an alphabetical letter.

isalphanumeric

```
function isalphanumeric(c : character) : Boolean
```

Returns true if the character is a alphanumeric.

s2i

```
function s2i(s : string)
```

Converts a string into an integer and returns it.

s2f

```
function s2f(s : string) : float
```

Converts a string into a float and returns it.

rot

```
function rot(s : string, rotations := 13) : string
```

Rotates a string using a specified number to rotate by. The default rotation is 13, which means all the letters ‘a’ will be turned into ‘n’ and ‘z’ will be turned into ‘m’.

Appendix A7 - Time Functions

These time functions are for working with the system time. The functions are defined in the file *time.p++*.

system_time

```
function system_time() : integer
```

Returns an integer containing the system time in Unix time format. This is the number of seconds since 01-01-70.

time_components

```
function time_components(time : integer = -1, ref year =
null, ref year_day = null, ref month = null, ref
month_day = null, ref week_day = null, ref hours = null,
ref minutes = null, ref seconds = null) : integer
```

Calculates the components of the supplied unix time (year, day of the year, month, day of the month, day of the week, hours, minutes and seconds) by using the *system_time* function. All the parameters in this function are optional and if you miss out the time parameter, the current system time is used. Supply any arguments (they are all outgoing reference parameters) to the parameters for the components of time you need, and *time_components* function will fill that argument with that time component.

```
var year;

time_components( , year);

/* year now contains the current year */
```

year

```
function year() : integer
```

Returns the year (e.g. 2000)

year_day

```
function year_day() : integer
```

Returns the day in the year (e.g. 100).

month

```
function month() : integer
```

Returns the month.

month_day

```
function month_day() : integer
```

Returns the day of the month.

week_day

```
function week_day() : integer
```

Returns the day of the week. (Sunday = 1, Month = 2….)

hours

```
function hours() : integer
```

Returns the hours of the day.

minutes

```
function minutes() : integer
```

Returns the minutes of the hour.

seconds

```
function seconds() : integer
```

Returns the seconds of the minute.

hms

```
function hms(time : integer = -1) : string
```

Returns a string containing the date in the following format:
“HH:mm:SS” where HH is hours, mm is minutes and SS is seconds.

dmy

```
function dmy(time : integer = -1) : string
```

Returns a string containing the date in the following format:
“DD:MM:YY” where DD is the day of the month, MM is the month and YY is the year.

Appendix A8 – Stream Functions

These stream functions are for working with P++ streams, these functions can work with files, sockets and pipes – all of which are streams. The functions are defined in the file *stream.p++*.

z_write_byte

```
function z_write_byte(z : stream, x : integer)
```

Writes a byte to the stream *z*

z_read_byte

```
function z_read_byte(z : stream)
```

Reads a byte from a stream *z*.

z_print

```
function z_print(z, s : string) : string
```

Prints a string to the stream *z*.

z_println

```
function z_println(z : stream, s : string) : string
```

Prints a string to the stream *z* and a new line character.

z_write_integer

```
function z_write_integer(z : stream, x : integer) :
integer
```

Writes an integer to the stream *z*.

z_write_float

```
function z_write_float(z : stream, x : float) : float
```

Writes a float to the stream *z*.

z_write

```
function z_write(z : stream, x : integer) : integer
```

Writes an integer to the stream *z*..

z_readln

```
function z_readln(z) : string
```

Reads a line from the stream *z*.

I – File functions

These file functions are for working with the filesystem. The functions are defined in the file *file.p++*.

f_open

```
function f_open(name : string, flags) : integer
```

Opens a file specified by the *name*. You must specify the flags which will determine how the file is opened.

File open flags.

Flag	Value	Description
`F_READ`	`0x1`	Open the file for reading.
`F_WRITE`	`0x2`	Open the file for writing.
`F_APPEND`	`0x4`	Open the file for appending.
`F_CREATE`	`0x80`	Create the file if it doesn't exist.
`F_TEXT`	`0x100`	Open the file with text translation.
`F_BINARY`	`0x200`	Open the file in raw binary mode.

f_close

```
function f_close(z : stream)
```

Closes a the file specified by the stream *z*.

f_eof

```
function f_eof(z : stream) : Boolean
```

Returns true if the end of the file specified by the stream *z* has been reached.

Appendex A9 : Socket functions

Socket functions are for TCP/IP networking. These are not yet completed.

Appendix Z1 : P++ Machine Op-codes

Table of P++ machine opcodes

Each machine op-code has two parameters, parameter l and parameter a.

`LIT`	Load up a literal number. *a* = number.
`OPR`	Does an integer operation. *a* = operation.
`FPO`	Does a float operation. *a* = operation.
`LOD`	Loads a variable. *l* = level difference. *a* = address.
`STO`	Stores a variable. l = level difference. *a* = address. The stack is popped.
`CAL`	Creates a new data segment and calls a function. *l* = level difference. *a* = address.
`INC`	Increments the stack pointer. *a* = amount in integers.
`JMP`	Jumps to an instruction. *a* = instructional address.
`JPC`	Jumps to an instruction if the stack top is 0. a = instructional address. The stack is popped.
`STP`	Stores the stack top as a parameter. *a* = number of integers till the next data segment.
`LID`	Loads a variable by an address. *l* = level difference, *a* = address of address.
`LDA`	Loads the address of a variable. *l* = level difference, *a* = address.
`SID`	Stores a variable by an address. *l* = level difference, *a* = address of address. The stack is popped.
`DEC`	Decrements the stack pointer. *a* = amount in integers. The stack is popped.
`CAI`	Creates a new data segment and calls a function. *l* = level difference, *a* = address of address.
`CII`	Creates a new data segment and calls a function. *l* = level difference, *a* = address of address of address.
`CAS`	Creates a new data segment and calls a function. *l* = level difference. The address for the function should be the stack top. The stack is popped.
`SLD`	Loads up a number from somewhere on the stack. *a* = how many integers down the stack the number is.
`TRM`	Terminate the program immediately.
`SWS`	Swaps the top 2 on the stack.

The following are inline unary operations and work on one variable. l = level difference, a = address. The stack top should be the second operand, this is popped after the operation.	
`ISL`	Shift bits left.
`ISR`	Shift bits right.
`IAD`	Add.
`ISU`	Subtract.
`IMU`	Multiply.
`IDI`	Divide.
`IXO`	Bit wise XOR.
`IOR`	Bit wise OR.
`IAN`	Bit wise AND.
The following are inline indirect unary operations and work on one variable. l = level difference, a = address of address. The stack top should be the second operand. The stack top should be the second operand; this is popped after the operation.	
`IZL`	Shift bits left.
`IZR`	Shift bits right.
`IIA`	Add.
`IIS`	Subtract.
`IIM`	Multiply.
`IID`	Divide.
`IIX`	Bit wise XOR.
`IIO`	Bit wise OR.
`IIB`	Bit wise AND.
Function op-codes.	
`SRV`	Set the return value. The top stack is the return value. *a* = number of integers underneath the current base of the data segment the value should go. The stack is popped.
`LRV`	Loads the return value onto the stack. *a* = number of integers underneath the current base of the data segment the value should go.
Memory op-codes.	
`MAL`	Allocates some memory. The top stack should be the size in integers. *l* = level difference, *a* = address. A handle to the memory is put in the variable located by l and a.
`MLS`	Allocates some memory. The top stack should be the size in

	integers. *l* = level difference, *a* = address. A handle to the memory is put on the stack.
`FRE`	Frees some memory given a handle. *l* = level difference, *a* = address.
`RAL`	Reallocates some memory given a handle. The top stack should be the new size in integers. *l* = level difference, a = address.
`MSZ`	Loads the size of some memory given a handle. *l* = level difference, *a* = address.
`MLD`	Loads some value off memory, the CX register specifies the memory offset. *a* = the offset down the stack where the handle to the memory is.
`MSS`	Sores some value onto memory, the CX register specifies the memory offset. *a* = the offset down the stack where the handle to the memory is.
`POP`	Puts the stack top and puts it into a register and pops the stack. *a* = the register.
`PUS`	Pushes a register onto the stack. *a* = the register.
`MOV`	Moves a value from one register to another. *l* = source register, *a* = destination register.
`RIO`	Does static bit wise OR with a register. *l* = number to or with, *a* = register.
`PAS`	Puts the stack top into a register without popping the stack. *a* = the register.
`SRG`	Sets a register. *l* = value, *a* = the register.
Utility op-codes	
`TME`	Loads the time in seconds since 1-1-1970 onto the stack.
`WRT`	Writes the stack top as an integer to standard output.
`WRF`	Writes the stack top as a float to standard output.
`WRB`	Writes the stack top as a byte to standard output.
`DEB`	Writes the l and a param to standard output for debugging.
`CLK`	Loads the CPU clock time elapsed for the process.

`CLI`	Allocates enough memory to contain a string for the command line arguments. The CX register specifies additional memory length to allocate, and it is also the offset where the command line arguments will start to be written. The length of the string written will be loaded onto the stack.
`REB`	Reads a byte from standard input and loads it onto the stack as an integer.
File op-codes.	
`FOF`	Opens a file, *l* and *a* specify the variable that holds the memory handle that holds the name of the file. The CX register holds the offset of where in the memory the name of the file is contained. Loads a handle to the file onto the stack.
`FCF`	Closes a file, *l* and *a* specify the variable that holds the memory handle that holds the name of the file. The CX register holds the offset of where in the memory the name of the file is contained. Loads a handle to the file onto the stack.
`EOF`	Determines if a file handle given represents a file which has been completely read (end of file). *l* and *a* specify the variable that holds the file handle.

Table of P++ integer operators.

This table documents operators to be used with the OPR P++ opcode. Unless specified, these operators work on the top two of the stack.

`OPR_NOT`	Bitwise NOT the top.
`OPR_ADD`	Add.
`OPR_SUB`	Subtract.
`OPR_MUL`	Multiply.
`OPR_DIV`	Divide.
`OPR_ODD`	Odd the top.
`OPR_MOD`	Modulus.
`OPR_EQL`	Equals.
`OPR_NEQ`	Not equals.
`OPR_LES`	Less than.
`OPR_LEQ`	Less than or equal to.
`OPR_GRE`	Greater than.
`OPR_GRQ`	Greater than or equal to.
`OPR_SHL`	Shift left.
`OPR_SHR`	Shift right.
`OPR_BOR`	Bitwise OR.
`OPR_AND`	Bitwise AND.
`OPR_XOR`	Bitwise XOR.
`OPR_NEG`	Negates the top.

Table of P++ floating point operators.

This table documents operators to be used with the FOP P++ opcode. Unless specified, these operators work on the top two of the stack.

FOP_ADD	Add.
FOP_SUB	Substract.
FOP_MUL	Multiple.
FOP_DIV	Divide.
FOP_EQL	Equals.
FOP_NEQ	Not equals.
FOP_LES	Less than.
FOP_LEQ	Less than or equal to.
FOP_GRE	Greater than.
FOP_GRQ	Greater than or equal to.
FOP_INT	Converts the top to a float.
FOP_FLO	Converts the top to an integer.
FOP_NEG	Negates the top.