This is the traditional “hello world” program.


```
import "io"

let start() be
  out("Greetings, Human.\n")
```

- `import` is a lot like `import` in java, and a bit like `#include` in C++.
- “`io`” is the standard library with input/output and other very basic functions.
- `let` introduces most simple declarations, it is not a type. There are no types.
- `start` serves the same purpose as `main` in java and C++.
- `be` is required when defining a function like this.
- The body of a function is a single statement, but see the next example.
- `out` is the ancestor of C’s `printf` and java’s `System.out.printf`.

Here is a bigger version.

```
import "io"

let start() be
{ out("Greetings, Human.\n");
  out("Now go away and leave me alone.\n") }
```

- Curly brackets combine multiple statements into one statement, called a block.
- Semicolons are only required between statements, as a separator.
  (the original BCPL didn’t require semicolons at all, but that leads to too many preventable mistakes, so I made a change there)
- But if you forget and put an extra one before the }, the compiler won’t complain.

Now for some local variables. (from now on, I won’t show the `import` statement)


```
let start() be
{ let x = 5, y = 10, z;
  x := x + 1;
  z := x * (y + 1);
  y +:= 2;
  out("x=%d, y=%d, z=%d\n", x, y, z) }
```

- `let` introduces the declaration again. You know that `x`, `y`, and `z` are variables and not functions because they are not followed by `()`.
- `let` is followed by any number of variable names all separated by commas. each variable may be given an initial value (like `x` and `y`), or left uninitialised (like `z`).
- There can be any number of `let`s, you don’t have to declare all your variables on one line.
- Unlike in the original BCPL, `let`s and other declarations can appear anywhere in a block. Originally, all declarations had to come before any executable statements.

- := is the symbol for an assignment statement. Unlike in C++, you can't give yourself a hard-to-debug problem by accidentally typing = instead of ==.
- Assignments are statements, not expressions. := can not be used as an operator in an expression.
- +:= is an update assignment just like += in C and java. You can use it with all the operators you'd reasonably expect: *:=, -:=, /:=, /\:=, !:=, etc.
- The full list of arithmetic operators are:
  Dyadic signed integer operations: + - * / rem **
  Dyadic floating point operations: #+ #- #* #/ #**
  Dyadic unsigned integer operations: ##* ##/ ##rem
  Monadic signed integer operations: + - abs
  Monadic floating point operations: #- #abs

Names of variables and functions and other things

- Must begin with a letter
- May also include any number of letters, digits, underlines, and dots.
  A B C D E F G H I J K L M N O P Q R S T U V W X Y Z
  a b c d e f g h i j k l m n o p q r s t u v w x y z
  0 1 2 3 4 5 6 7 8 9 _ .
- Capital letters and little letters are not distinguished.
- cat, CAT, Cat, and cAt are just four ways of typing the same variable.
- That applies to the rest of the language too: let, LET, and Let are the same thing.


```
let start() be
{ let x;
  x := 123;
  out("x=%d\n", x);
  x := "hello";
  out("x=%s\n", x);
  x := 12.34;
  out("x=%f\n", x) }
```

- Variables do not have types, you can store anything you like in any variable.
- It is up to the programmer to remember what kind of thing has been put in which variables.
- Every variable is just a 32 bit value. How those bits are interpreted or used is determined by what you do with that variable.
- The 32-bit values and memory locations are called "Words", regardless of their use. All of memory is a giant array of words.


```
let start() be
{ let x = 84;
  out("%d in decimal is:\n", x);
  out("  %x in hexadecimal and %b in binary\n", x, x);
  out("  and is the ascii code for the letter %c\n", x) }
```

- out uses %d for integers to be printed in decimal,
- %x or %X for integers to be printed in hexadecimal,
- %h also prints in hexadecimal, except that zeros appear as lower case 'o'. This sometimes makes numbers easier to read.
- %b for integers to be printed in binary,
- %c for integers to be printed as characters,
- %C for integers to be printed as characters, but every character, even a control code, is made explicitly visible. ASCII code 0 appears as \0, \ as \\, quotes as \' and \", newlines as \n, returns as \r, tabs as \t, backspaces as \b, ordinary spaces as \s, and any other invisible character as *abc* where *abc* is the three digit octal rendering of the character's ASCII code.
- %f prints floating point values, always according to this format: +3.141593e+00
- %s prints strings,
- %v prints strings using the %C format for every character, even the terminating 0 is displayed as \0.
- With the formats d, x, h, b, s, and v, a width may appear between the % and the format letter, as in %7d. In the case of the numeric formats, this is a minimum width, if the number requires fewer digits than this, extra spaces are added. For %v, the width is the number of characters from the string that will be printed, not the number of characters that will be needed to display them.
- A width may also be specified for the formats c and C, it must be between 0 and 4 inclusive. Normally the c and C formats ignore all but the 8 least significant of the value being printed, but 32 bit word can hold up to four bytes or character codes. When a width is provided, exactly that number of bytes will be printed, even if they are equal to zero.
- If the width is preceded by a '-', as in %-7d, spaces will be added to the right. Otherwise they will be added to the left.
- With the formats d, x, h, and b, if the size is preceded by a zero, as in %08x and %-07d, zeros will be used for the padding instead of spaces.
- With the format s, if the size is preceded by a zero, as in %08s and %-07s, the size is exact. Short strings are padded as usual, but strings longer than the given size are cut short.
- With the d and b formats only, the % sign may be followed by a comma. If a comma appears then the digits printed will be separated into groups of three (for %,d) or four (for %,b), separated by commas in the traditional way.

Input from the user

```
let start() be
{ let x, y;
  out("type a number. ");
  x := inno();
  out("and another one: ");
  y := inno();
  out("%d times %d is %d\n", x, y, x*y) }
```

- inno waits for the user to type an integer in decimal, and returns its value.

- `inch` reads a single character and returns its ascii value.
- If you want to read anything more complicated than a decimal integer, you'll have to write a function for it.

```
let inbin() be
{ let value = 0;
  while true do
  { let char = inch();
    if char < '0' \/ char > '1' then
      resultis value;
    value := value * 2 + char - '0' } }

let start() be
{ let x;
  out("type a number in binary. ");
  x := inbin();
  out("that is %d in decimal\n", x) }
```

With that definition, `inbin` reads an integer from the user in binary.

- `while` means the same as it does in C++ and java, but
  - you don't need to put parentheses around the condition, and
  - you *do* need to put the word `do` after the condition.
- `true` is exactly equivalent to `-1`, `false` is exactly equivalent to `0`.
- `while`, and all other conditionals, accepts any non-zero value as being true.

- `if` means the same as it does in C++ and java, but
  - you don't need to put parentheses around the condition, and
  - you *do* need to put the word `then` or `do` after the condition.
  - `if` statements never have an `else`.

- The logical operators are `/\` for *and*, `\/` for *or*, and `not` or `~` for *not*.
  - `/\` and `\/` use *short-circuit* evaluation:
    in `A /\ B, if A` is false, `B` will not even be evaluated.
    in `A \/ B, if A` is true, `B` will not even be evaluated.
  - `not` replaces 0 (which is false) to -1 (which is true),
    it replaces everything other than 0 with 0.
  - `~` is exactly the same thing as `not`.
  - The *bit-wise* logical operators are `bitnot`, `bitand`, `bitor`, `eqv`, and `neqv`.

- The relational operators for integer values are
  - `<` for less than,
  - `>` for greater than,
  - `<=` for less than or equal to,
  - `>=` for greater than or equal to,

- `=` for equal to, don't use ==.
- `<>` for not equal to (it is saying less than or greater than)
- `/=` also means not equal to, and so does `\=`, the three are identical.
- For floating point comparisons use
  `#<  #>  #<=  #>=  #=  #<>  #/=  #\=`
- For unsigned integer comparisons use
  `##<  ##>  ##<=  ##>=  ##=  ##<>  ##/=  ##\=`
- Relational operators may be strung together: `a<b<c` means `a<b /\ b<c`.

- `resultis X` means the same as `return X;` does in C++ and java: the function exits immediately, and returns `X` as its value, but.
  - `resultis` must always be given a value to return.
  - `return` is used to exit from a function that does not produce a value.

- Single quotes mean the same as in C++ and java: a character enclosed in single quotes is the integer value of its Ascii code, but up to four characters may be enclosed in the single quotes, because 4 character codes can fit in 32 bits:
  `'' = 0`,
  `'a' = 97`, that is the ASCII code for the letter a.
  `'ab' = 'a'×256 + 'b'`,
  `'abc' = 'a'×256×256 + 'b'×256 + 'c'`,
  `'abcd' = 'a'×256×256×256 + 'b'×256×256 + 'c'×256 + 'd'`.
- Recall that the out functions `%c` and `%C` formats can take a width between 0 and 4. The hexadecimal values 0x45, 0x46, 0x47, and 0x20 are the Ascii codes for E, F, G, and space respectively, so `out("'%4c'", 0x20454647)` would print `' EFG'`.

Conditional Statements

```
if x < 0 then count := count + 1;
if y >= 0 do count := count - 1;
if a + b > 100 then
{ out("Too big!\n");
  finish }
```

- In an `if` statement, `then` and `do` mean exactly the same thing.
- `finish` is the same as `exit()` in C++ and java, except that it is a statement, not a function, so there is no `()` pair following it. It just stops the whole program.
- But `finish` may be followed by an expression that should evaluate to a small integer. This is the exit code that will be delivered to the host system. If there is no expression, the exit code is zero.

```
unless x >= 0 do count := count + 1;
unless y < 0 do count := count - 1;
```

```
unless a + b <= 100 then
{ out("Too big!\n");
  finish }
```

- `unless X` means the same as `if not(X)`.

```
test 1 <= x <= 10 then
  out("Everything is fine\n")
else
  out("Panic! x = %d, out of range\n", x);
```

- Neither `if` nor `unless` may take an `else`.
- Allowing that in C++ and java makes the meanings some programs unclear in a way that most programmers are unaware of.
- `test` is the thing to use. `test` is the same as `if` in C++ and java, but it must *always* have an `else`, `else` is not optional with `test`.

```
test x < 1 then
  out("Too small\n")
else test x > 10 then
  out("Too big\n")
else
  out("OK\n")
```

- Of course `tests` may be strung together like that.
- The word `or` may be used instead of `else`, they mean exactly the same thing.

◈

Loops

```
x := 1;
while x < 10 do
{ out("%d ", x);
  x +:= 1 }
```

- That prints 1 2 3 4 5 6 7 8 9

```
x := 1;
until x > 10 do
{ out("%d ", x);
  x +:= 1 }
```

- That prints 1 2 3 4 5 6 7 8 9 10
- `until` is just a reader-friendly way of saying `while not`.
- `until X` means exactly the same as `while not(X)`.

```
x := 1;
{ out("%d ", x);
  x +:= 1 } repeat
```

- That prints 1 2 3 4 5 6 7 8 9 10 11 12 13 14 ... and never stops

- `repeat` is the same as `do ... while (true)` in C++ and java.

```
x := 1;
{ out("%d ", x);
  x +:= 1 } repeatwhile x < 10
```

- That prints 1 2 3 4 5 6 7 8 9
- `repeatwhile` is the same as `do ... while` in C++ and java, the body of the loop gets executed once even if the condition is initially false.

```
x := 1;
{ out("%d ", x);
  x +:= 1 } repeatuntil x > 10
```

- That prints 1 2 3 4 5 6 7 8 9 10
- `repeatuntil X` is the same as `repeatwhile not(X)`.


```
{ let x = 0;
  while true do
  { x +:= 1;
    if x rem 3 = 0 then loop;
    if x > 16 then break;
    out("%d ", x) }
  out("end\n") }
```

- That prints `1 2 4 5 7 8 10 11 13 14 16 end`
- `rem` is the remainder or modulo operator, like `%` in C++ and java.
- Try to remember that `%` means something else and will cause very odd errors.
- `break` means exactly the same as in C++ and java, it immediately terminates the loop that it is inside. It can only be used in a loop.
- `loop` means exactly the same as in `continue` does in C++ and java, it abandons the current iteration loop that it is inside, and starts on the next. It can only be used in a loop.


```
{ let i = 1234, sum = 0;
  for i = 3 to 25 by 3 do
  { sum +:= i;
    out("%d ", i) }
  out("i=%d\n", i) }
```

- That prints `3 6 9 12 15 18 21 24 i=1234`
- A `for` loop always creates a new temporary control variable, that variable does not exist any more once the loop ends.
- The initial value and the final value may given by any expression.
- The by value must be a compile time constant, meaning that the compiler must be able to work out its value before the program runs, so it can't involve any variables.

```
max := 9;
for i = 1 to max+1 do
{ if i = 5 then max := 20;
  out("%d ", i) }
out("max=%d\n", max) }
```

- That prints `1 2 3 4 5 6 7 8 9 10 max=20`
- The terminating value for a `for` loop is calculated just as the loop starts, and is stored until the loop ends. It is not recalculated at each iteration. In the example, changing the value of max had no effect on how many times the loop ran.
- If you don't provide a `by` value, the compiler assumes `1`.

```
for i = 10 to 1 do
  out("%d ", i)
```

- That prints nothing at all.
- If you want the loop to count backwards you must explicitly say `by -1`.
- If the `by` value is positive, the loop runs while the variable is `<=` to the `to` value.
- If the `by` value is negative, the loop runs while the variable is `>=` to the `to` value.


```
while true do
{ let c = inch();
  switchon c into
  { case ' ':
          out("a space\n");
          endcase;
    case '.':
          out("a dot\n");
          endcase;
    case '+':
          out("a plus sign, ");
    case '-': case '*': case '/':
          out("an operator\n");
          endcase;
    case '0' ... '9':
          out("a digit\n");
          endcase;
    case 'A' ... 'Z': case 'a' ... 'z':
          out("a letter\n");
          endcase;
    default:
          out("something else\n") } }
```

- `switchon X into` is just the same as `switch (X)` in C++ and java.
- Execution jumps immediately and efficiently to the `case` label matching the value of the expression, which should be an integer.
- `switchon` does not use a series of comparisons, so it is faster than a sequence of `test else test else` ...

- `endcase` causes a jump to the end of the `switchon` statement. If no `endcase` is met, execution runs into the next case. In the example, a `'+'` gets the response "a plus sign, an operator".
- The `case` labels must be constants, and there may be no repetition (or in the case of ranges given with `...`, there must be no overlap). Each possible value may only appear once.
- The `default` label catches all values that are not covered by a `case` label.
- `default` is not required. Without it, unmatched values do nothing.
- The overall range of all the `case` values must not be very large, or the resulting executable code will be too big to run.


```
if count = 0 then
  $debug(12)
```

- `$debug` causes program execution to be suspended, and control is delivered to the assembly language debugger. The parameter given to `$debug` is made visible when the debugger takes over, so you can tell which is which when you have multiple debugging points. For details on how to debug programs, see the commands described in the separate documentation for the emulator.

Disapproved-of Statements

```
let start() be
{ let a = 0;
  start: a +:= 1;
  if a rem 10 = 4 then goto start;
  if a > 100 then goto elephant;
  out("%d ", a);
  goto start;
  elephant: }
```

- That program will count from 1 to 100, skipping numbers whose last digit is 4, then stop.
- Any statement or `}` may be given a label. Labels are names with the same rules as variables, and are attached to a statement with a colon.
- Reaching a label has no effect on execution.
- A `goto` statement causes an immediate jump to the statement with the matching label.
- It is not possible to jump into a block from outside it, and it is not possible to jump to a label in a different function. Labels are in fact local variables. A new value may be assigned to a label at any time (e.g. `elephant := start`).
- `goto` may be followed by an expression. If the value of the expression does not turn out to be a label, the results are unpredictable.

- Anything that happens in a program that uses a `goto` is the programmer's own fault, and no sympathy will be received.

```
let start() be
{ /* this function is to demonstrate
     the use of comments in a program.
     comments may be very big or very
     small or anything in between */
  let a = 0;
  let b = 0; // b is the hypotenuse
  let c = 5, d = 99;
  a := (c + 1) * (d - 1);
  if a < 20 /* not good */ then out("%d ", a);
  d := c - 4;
  a := 0 // the curly bracket on this line is ignored }
} // so we need an extra one here
```

- Comments can go anywhere, they have the same effect as a space.

Functions

```
import "io"

let factorial(n) be
{ let f = 1;
  for i = 1 to n do
    f *:= i;
  resultis f; }

let display(a, b) be
{ out(" N N!\n");
  out("---------\n");
  for i = a to b do
    out(" %d %d\n", i, factorial(i));
  out("----------\n") }

let average(x, y) = (x+y)/2

let start() be
  display(3, average(7, 11))
```

- That program prints a table of factorials from 3 to 9.
- If a function has parameters, there names are listed between the parentheses in its declaration, separated by commas. Nothing else can go in there, there is nothing to say about a parameter except for its name.
- Parameters behave just like local variables.
- If a function's result can be expressed as a single expression, the simplified form as used for `average` may be used. Any expression may follow the `=`.
- If a function only consists of a single statement, the `{` and `}` are not required.
- `resultis` is used to exit a function and return a value,
- `return` is used to exit a function without returning a value.

- When a function is called, it is not necessary to provide the correct number of parameter values. If too few values are provided, the last parameters will simply be uninitialised variables.
- BUT: any attempt to assign to or modify an un-passed parameter will have disastrous and hard-to-trace consequences.


```
let f(x, y) be { ... }
and g(a) be { ... }
and h(p, q, r) be { ... }
```

- Every function must be declared before it is used. There are no prototypes, *simultaneous declaration* is used instead.
- When function definitions are linked together using `and` instead of repeated `let`s, all of those functions are declared before any of their defining statements are processed.
- In the example above, each of the three functions `f`, `g`, and `h` may call any or all of those same three functions.


```
let process(a, b) be
{ let f(z) = (z + 10) * (z - 10);
  let modify(x) be
  { let z = f(x + 1);
    if z < 0 then resultis 1;
    resultis x + 3 }
  let sum = 0;
  for i = a to b do
    sum +:= modify(i);
  resultis sum }
```

- Functions may have their own local function definitions.
- In the example, the function `modify` is only available within the function `process`. Elsewhere the name `modify` has no meaning, just as with local variables.
- This feature is of limited usefulness; `modify` is not permitted to access any of `process`'s parameters or local variables, although it can access other local functions.


```
let increment(x) be
{ let total;
  if numbargs() = 0 then total := 0;
  total +:= x;
  out("  the total is now %d\n", total) }

let start() be
{ out("reset\n");  increment();
  out("add 1\n");  increment(1);
  out("add 2\n");  increment(2);
  out("add 1\n");  increment(1);
  out("add 1\n");  increment(1);
```

```
  out("reset\n");  increment();
  out("add 2\n");  increment(2);
  out("add 1\n");  increment(1);
  out("add 3\n");  increment(3) }
```

- The library function `numbargs()` returns the number of parameters (arguments) that the current function was called with.
- The idea of the example is that the variable `total` is used to accumulate all the values `increment` has been given. Calling `increment` with no parameters is a signal to reset the total back to zero.
- Naively we might expect it to report totals of 0, 1, 3, 4, 5, 0, 2, 3, and 6.
- Of course it doesn't work. Local variables are created anew each time a function is called, and lost when it exits.

```
let total = 0;

let increment(x) be
{ if numbargs() = 0 then total := 0;
  total +:= x;
  out("  the total is now %d\n", total) }
```

- This alternative definition does work. `let` creates local or global variables depending upon whether it is used inside or outside of a function.
- A global variable isn't the ideal solution here. `total` is supposed to be private to `increment`. Now there is a risk that other functions could change it.


```
let increment(x) be
{ static { total = 0 }
  if numbargs() = 0 then total := 0;
  total +:= x;
  out("  the total is now %d\n", total) }
```

- This version solves both problems. It works.
- A static variable is a private global variable. It is created and initialised only once, when the program is started, and it exists until the program ends, but it is only visible inside its enclosing function. Elsewhere the name is unknown.
- A number of static variables may be declared at once, inside the same `static { }`, as in `static { total=0, min=99, max=-1 }`.
- If `increment` had any local functions, they *would* be permitted to access `increment`'s static variables.


```
let array(a, b) be
{ test lhs() then
    out("you said array(%d) := %d\n", a, b)
  else test numbargs() = 1 then
  { out("you said array(%d)\n", a);
    resultis 555 }
  else
```

```
    out("you said array(%d, %d)\n", a, b) }

let start() be
{ let v, w;
  array(2) := 345;
  array(3) := 9876;
  v := array(2);
  w := array(3);
  out("v + w = %d\n", v + w) }
```

- There are no arrays is this example, array is just a normal function.
- A function call may appear on the left-hand-side of an assignment statement, as in
  `array(2) := 345` or
  `storage(34, x + 9) := y * z`
- When that happens, it is just treated as a normal function call, but the expression to the right of the `:=` becomes an extra parameter,
- and inside the function, the library function `lhs()` returns `true` instead of its normal value of `false`. `lhs()` is true if the function call is the left-hand-side of an assignment.
- This allows an approximation of the get and set methods of object oriented programming to be implemented with a reader-friendly syntax, as the example is hinting.
- The example prints

```
you said array(2) := 345
you said array(3) := 9876
you said array(2)
you said array(3)
v+w = 1110
```

Very Local Variables

```
let start() be
{ let a = 3, b = 4, c, d;
  c := t * (t + 1) where t = a + 2 * b - 1;
  d := x * x + y * y where x = a + b + 1, y = a - b - 2;
  out("c = %d, d = %d\n", c, d) }
```

- The `where` clause introduces one or more temporary local variables.
- `where` attaches to a whole single statement. Of course, that statement could be a block of many statements inside { }.
- Those variables exist only for the execution of that one statement, then they are destroyed leaving no trace.
- The example prints `c=110, d=73`.


```
manifest
{ pi = 3.14159,
  size = 1000,
  maximum = 9999,
```

```
    half = maximum/2 }
```

- A `manifest` declaration introduces one or more named constants.
- The value of a manifest constant can not be changed deliberately or accidentally by a running program.
- `manifest` declarations may be global or local inside a function.
- The values given to manifest constants must be compile time constants, values that the compiler can easily work out before the program runs. They may not make use of any variables or functions, nor may they be strings.
- `manifest` is the ancestor of `const` in C++ and `final` in java.


```
let addup(a) be
{ let sum = 0, ptr = @ a;
  for i = 0 to numbargs() - 1 do
  { sum +:= ! ptr;
    ptr +:= 1 }
  resultis sum }

let start() be
{ out("1 to 5: %d\n", addup(1, 2, 3, 4, 5));
  out("3 + 12 + 7: %d\n", addup(3, 12 ,7));
  out("nothing: %d\n", addup()) }
```

- `@` is the address-of operator. It provides the numeric address of the memory location that a variable is stored in.
- `!` is the follow-the-pointer operator. It assumes that its operand is a memory address and provides the contents of that location.
- Every variable and every value is 32 bits long, and memory locations are 32 bits long.
- Parameters to a function are always stored in neighbouring memory locations, in ascending order, so `addup` successfully adds up all its parameters regardless of how many there are. This is also how `out` works.


```
let glo = 7

let start() be
{ let var = 10101;
  let ptr = @ glo;
  ! ptr := 111;
  ! ptr *:= 2;
  ptr := @ var;
  ! ptr +:= 2020;
  out("glo = %d, var = %d\n", glo, var) }
```

- An `!` expression can be the destination in an assignment.
- The sample program prints `glo = 222, var = 12121`.

```
let start() be
{ let fib = vec 20;
  fib ! 0 := 1;
  fib ! 1 := 1;
  for i = 2 to 19 do
    fib ! i := fib ! (i - 1) + fib ! (i - 2);
  for i = 0 to 19 do
    out("%d\n", fib ! i) }
```

- `vec` is a special form that can be used as the initial value for any variable. It is *not* an expression that can be used in assignments or anywhere else.
- Its argument must be a compile-time constant.
- When `name = vec size` appears, `name` is declared as an ordinary variable, and immediately next to it in memory a space of exactly `size` words is left uninitialised. The value of `name` is set to the address of the first of these locations.
- So `fib` is a pointer to an array of twenty 32-bit values.
- If `vec` appears as the value of a local variable, it is a local temporary array with the same lifetime as the variable. If `vec` appears as the value of a static or global variable, it is a permanent array.

- The infix form of the `!` operator is very simple. Its exact definition is

  `A ! B ≡ ! (A + B)`
- It is like using `[ ]` to access an array in C++ and java, except that it is symmetric, `A ! B` is always the same thing as `B ! A`.
- `fib` was initialised to `vec 20`, which means `fib` is a variable that contains the address of the first of an array of 20 memory locations. Thus `fib+1` is the address of the second in that array, and `fib!1` accesses the value stored there.
- The sample prints the first 20 fibonacci numbers.


```
let total(v, n) be
{ let sum = 0;
  for i = 0 to n - 1 do
    sum +:= v ! i;
  resultis sum }

let start() be
{ let items = table 23, 1, 2 * 3, 9, 10;
  let twice = vec(5);
  for i = 0 to 4 do
    twice ! i := 2 * items ! i;
  out("the total of items is %d\n", total(items, 5));
  out("the total of twice is %d\n", total(twice, 5)) }
```

- A table is a pre-initialised array. The value of `items` is the address of the first of a sequence of five memory locations. When the program first starts, the values 23, 1, 6, 9, and 10 are stored in those locations, and they are never re-initialised. The elements of a table behave like static variables.
- The values in a table must be compile-time constants, strings, or other tables.

- The variables `items` and `twice` both contain pointers to arrays, when they are used as parameters in a function call, it is those pointers that are copied, the `@` operator is not used.
- Inside the function, the parameter `v` has exactly the same value as `items` or `twice`, so it is used in exactly the same way.


```
let makearray() be
{ let a = vec(10);
  for i = 0 to 9 do
    a ! i := 2 ** i;
  resultis a }

let start() be
{ let powers = makearray();
  out("The answer is\n");
  for i = 0 to 9 do
    out("  %d\n", powers ! i) }
```

- `**` is the to-the-power-of operator for integers.
- The `makearray` function is wrong. The memory occupied by `a`, `i`, and the array itself is temporary and local to the function. As soon as the function exits, it will be re-used for something else.
- `powers` does receive the address of the memory locations that the array used to occupy, but it has been re-used, so the expected values are no longer there.
- To do this correctly, the `newvec` library function is used. `newvec` is a bit like `new` in C++ and java, but much more basic. It is closer to `malloc` in C.

This is selection sort.

```
let sort(array, size) be
{ for i = 0 to size-1 do
  { let minpos = i, minval = array ! i;
    for j = i + 1 to size - 1 do
      if array ! j < minval then
      { minpos := j;
        minval := array ! j }
    array ! minpos := array ! i;
    array ! i := minval } }

let start() be
{ manifest { N = 20 }
  let data = vec(N);
  random(-1);
  for i = 0 to n - 1 do
    data ! i := random(99);
  sort(data, N);
  for i = 0 to n - 1 do
    out("%2d\n", data ! i);
  out("\n") }
```

- Normally `random(N)` will produce a random number between `0` and `N` inclusive.
- It is of course a *pseudo*-random number generator: every time you run the program it will produce the same sequence of numbers.
- `random(-1)` changes that. It should be used just once in a program, it randomises the pseudo-random number sequence so that it will not be predictable.


```
import "io"
import "heap0"

let makearray(n) be
{ let a = newvec(n+1);
  for i = 0 to n do
    a ! i := 2 ** i;
  resultis a }

let start() be
{ let powers1, powers2;
  init();
  powers1 := makearray(10);
  powers2 := makearray(20);
  out("The answers are\n");
  for i = 0 to 10 do
    out("  %d\n", powers1 ! i);
  for i = 0 to 20 do
    out("  %d\n", powers2 ! i);
  freevec(powers1);
  freevec(powers2) }
```

- This is from an earlier example of something that didn't work, but now corrected.
- Note that it is importing a second library `heap0` as well as the usual `io`.
- `newvec` is like `vec`, it gives a pointer to an array. But it doesn't use temporary memory that is local to the function, it uses permanent heap memory that will never be recycled, so the pointer remains valid for ever. `newvec` is similar to `new` in C++ and java, and more similar to `malloc` in C.
- Unlike `vec`, `newvec(X)` is a normal function call, it can be used anywhere in a program, and its parameter can be any expression.
- In every other way, an array created by `newvec` is indistinguishable from an array created by `vec`.
- `freevec` is the function used to release `newvec` allocations for recycling.
- The version of `newvec` provided by `heap0` is extremely primitive. It can not request additional memory allocations from the operating system because there is no operating system. It doesn't even do its advertised job: it makes no attempt to recycle memory. In fact it does nothing at all, it just serves as a place-folder for when you create your own heap system.
- Before first using `newvec` in a program, the programmer must call the `init` function. In its simplest parameterless form, `init` takes over all of the computer's memory that is not in use, and uses it as the heap. This is nearly always the best approach.

- If the programmer wishes to control the memory that is used for the heap, they can use any memory that is available. This would normally be done by creating a large local array in `start`, and giving it to `init`:

```
let start() be
{ let heap = vec(10000);
  init(heap, 10000);
  ...
```

The parameters to `init`, if supplied, are the address of the beginning of the area of memory to be used as the heap, and the number of words that are available.

These are the definitions of `newvec` and `freevec` from the `heap0` library:

```
export { newvec, freevec, init }
static { vecsize = 0, vecused = 0, vecspace }

let newvec(n) be
{ let r = vecspace + vecused;
  if vecused + n > vecsize then
  { print("\nnewvec: insufficient free memory\n");
    resultis nil }
  vecused +:= n;
  resultis r }

let freevec(v) be
{ }

let init(address, size) be ...
```

I have left out the definition of print, it does the same thing as outs.

- These default versions of `newvec` and `freevec` don't do much at all. `newvec` constantly takes away from the heap space and will eventually run out. `freevec` does nothing at all, so there is no recycling. That is why this library is called `heap0` rather than just `heap`.
- The point is that programmers can still do effective programming before eventually creating their own heap library defining new versions of `init`, `newvec`, and `freevec` that will do the job effectively.
- But if the programmer is not interested in the educational value of creating their own heap, there is also a library just called `heap` which does it properly, so just import `heap` instead of `heap0`.


```
manifest
{ node_data = 0,
  node_left = 1,
  node_right = 2,
  sizeof_node = 3 }

let new_node(x) be
```

```
{ let p = newvec(sizeof_node);
  p ! node_data := x;
  p ! node_left := nil;
  p ! node_right := nil;
  resultis p }
```

- When implementing a binary tree of integers, each node is a very small object. It just contains three 32-bit values: the data item, the address of the node to the left, and the address of the node to the right.
- It might as well just be a three element array. The programmer decides which array positions the three items will occupy and defines well-named constants to make the program comprehensible.
- `new_node` is effectively a constructor for such a node.
- `nil` is a constant equal to 0. It does the same job as `NULL` in C++ and `null` in java. Its only purpose is to indicate "no pointer here".

```
let add_to_tree(ptr, value) be
{ if ptr = nil then
    resultis new_node(value);
  test value < ptr ! node_data then
    ptr ! node_left := add_to_tree(ptr ! node_left, value)
  else
    ptr ! node_right := add_to_tree(ptr ! node_right, value);
  resultis ptr }

let inorder_print(ptr) be
{ if ptr = nil then return;
  inorder_print(ptr ! node_left);
  out("%d ", ptr ! node_data);
  inorder_print(ptr ! node_right) }

let start() be
{ let heap = vec(10000);
  let tree = nil;
  init(heap, 10000);
  for i = 1 to 20 do
  { let v = random(99);
    tree := add_to_tree(tree, v);
    out("%d ", v) }
  out("\n");
  inorder_print(tree);
  out("\n") }
```

- Those three functions are nothing special. They just create a tree of random numbers, then print them in order.
- When dealing with a linked list, you will often find loops containing statements like `ptr := ptr ! link_next`. That could be abbreviated to `ptr !:= link_next`, but that's pretty ugly.

- Builtins are things that look like functions but really aren't. Their names always begin with a `$` sign, and can only be used with the syntax of a function call, e.g. `n := $find_first_non_zero(ptr1, ptr2);`. Names that begin with a `$` sign can not be used anywhere else, they do not denote values.
- Builtins give direct access to hardware features that would otherwise require the programmer to insert portions of assembly language. They often provide operations that need to be performed quickly, but would otherwise require a slow BCPL loop.
- The builtins are all described in section 57.

Strings are like strings in other languages, except for one small complication. Because our memory has a 32 bit word as its basic unit, rather than the 8 bit byte that is currently popular, accessing the individual characters of a string takes a little more effort than usual. For efficiency, the characters of a string are packed together four per word so that there is no wasted space.

```
let start() be
{ let s = "ABCDEFGHIJKLMN";
  for i = 0 to 3 do
    out("%08x\n", s ! i) }
```

```
44434241
48474645
4C4B4A49
00004E4D
```

- Every memory location is 32 bits wide, ASCII characters only require 8 bits. A string is just an array in which character codes are packed 4 per entry.
- The `%08x` format prints a number in hexadecimal, stretched out to the full 8 digits, with leading zeros added if necessary.
- In hexadecimal, the ASCII codes for the letters A, B, C, D, E are 41, 42, 43, 44, 45 and so on.
- The first character is stored in the least significant 8 bits of a string. That makes the output look backwards, but it makes more sense for programming.
- A string is always 8 bits longer than would be required for the characters alone. The end of a string is marked by 8 bits of zero.

```
let start() be
{ let a = vec(6);
  a ! 0 := 0x44434241;
  a ! 1 := 0x48474645;
  a ! 2 := 0x4C4B4A49;
  a ! 3 := 0x4E4D;
  out("%s\n", a) }
```

```
ABCDEFGHIJKLMN
```

- A constant string in "double quotes" is just an array (in static memory) initialised with the character codes when the program starts, just like a `table`.
- But naturally, any array that is big enough can be used as a string.

- `0x` prefixes a numeric constant written in hexadecimal. `0o` may be used for octal, and `0b` for binary.

```
let strlen(s) be
{ let i = 0;
  until byte i of s = 0 do
    i +:= 1;
  resultis i }
```

- That is the `strlen` function from the standard library `io`, it returns the length (number of characters not including the terminating zero) of a string. It is the only library function that does anything with strings other than printing them. You will need to make your own string functions for comparing, copying, and so on.

The escape codes that may appear inside string and character constants are:

`\\`	which represents	`\`
`\"`		`"`
`\'`		`'`
`\n`		newline, ascii 10
`\r`		return, ascii 13
`\t`		tab, ascii 9
`\b`		backspace, ascii 8
`\s`		ordinary space, just so it can be made explicit
`\nnn`		nnn is three octal digits: char with that ascii code

- `of` is an ordinary two-operand operator. Its left operand describes a sequence of bits within a larger object. Its right operand should be a pointer (memory address).
- `byte` is an ordinary one-operand operator. Its result is a perfectly ordinary number that is interpreted by `of` to describe one single character of a string.
- Due to hardware limits, `byte` and `of` can only be used to access the first 8,388,604 characters of a string.

```
let start() be
{ let alpha = "ABCDEFGHIJKLMNOPQRSTUVWXYZ";
  let p;
  out("byte 23 of alpha = '%c'\n", byte 23 of alpha);
  p := byte 23;
  out("byte 23 = %d\n", p);
  out("5896 of alpha = '%c'\n", 5896 of alpha) }
```

```
byte 23 of alpha = 'X'
byte 23 = 5896
5896 of alpha = 'X'
```

- `byte` is a perfectly ordinary operator whose result is a perfectly ordinary number.

```
let start() be
{ let s = vec(8);
  let letter = 'z';
  for i = 0 to 25 do
  { byte i of s := letter;
    letter -:= 1 }
  byte 13 of s -:= 32;
  out("%s\n", s) }

zyxwvutsrqponMlkjihgfedcba
```

- `byte ... of` can be used as the destination in an assignment or an update.
- The letter M became capital because the difference between the codes for capital letters and little letters in ASCII is 32.
- Writing `byte 13 of s -:= 'a' - 'A';` would have made that clear.

◈


```
let start() be
{ let bits = 0b10001000100010001101101101100010;
  let sel = selector 11 : 5;
  let part = sel from bits;
  out("%b\n", bits);
  out("                %b\n", part);
  sel from bits := 0b01010101010;
  out("%b\n", bits) }

10001000100010001101101101100010
                11011011011
10001000100010000101010101000010
```

- `selector` is like `byte`, but it is not limited to describing 8-bit items.
- `selector B : R` describes the `B`-bit region of a word that has `R` bits to the right of it.
- `B` may not be more than 32 nor equal to 0.
- `B + R` may not be more than 32.
- `B` and `R` do not have to be constants, any expression will do.
- `from` is like `of`, but it is not given a pointer, it is given the actual 32-bit value to select bits from.
- `from` may be used as the destination in an assignment or an update.
- All of this is mainly for packing a number of small values efficiently in a single word. If you were working with 5-bit values, all in the range 0 to 31, you could fit six of them into a single word, wasting the two most significant bits. If the value of `v` is `0b00001000100010001101101101100010` the six 5-bit values would be `0b00100`, `0b01000`, `0b10001`, `0b10110`, `0b11011`, and `0b00010`, or 4, 8, 17, 22, 27, and 2. `selector 5 : 5 from v` would deliver the 27, `of` and `from` assume everything is unsigned.
- If you are dealing with signed 5-bit values, they would be in the range −16 to +15, their most significant bits determining the sign. That means that the six values shown in the previous paragraph should really be interpreted as 4, 8, −15, −10, −5,

and 2. To achieve this, just use the `$sign_extend(val, bit)` builtin function. `val` is the number that needs to be reinterpreted and `bit` says which bit should be taken as the sign bit (count bits from the left starting with 0). In the example we are using 5-bit values, the bit numbers will be 0, 1, 2, 3, and 4. Bit 4, being the most significant, would be the sign bit. Taking the specific example of `0b10110`, which gave `val = 22`, the assignment `val := $sign_extend(val, 4)` would set `val` to the intended value of -10.

```
let start() be
{ manifest { those = selector 16 : 8 : 2 }
  let them = table 0x13578642, 0xBEEFFACE, 0x1A2B3C4D, 0xE8500C2A;
  out("%x\n", them ! 2);
  out("  %x\n", those of them);
  those of them := 0x9988;
  out("%x\n", them ! 2);
  selector 1 : 31 : 2 of them := 1;
  out("%x\n", them ! 2) }
```

```
1A2B3C4D
  2B3C
1A99884D
9A99884D
```

- `selector B : R : N` describes the `B`-bit region of a word that has `R` bits to the right of it, in word `N` of an array.
- `selector B : R` is an abbreviation for `selector B : R : 0`.
- When a selector value is used with the `from` operator, the `N` portion is ignored because `from` works on a single word, not an array.
- `byte X` is exactly equivalent to `selector 8 : (X rem 4) * 8 : X / 4`.
- There is no form of selector that can describe a sequence of bits that are not all contained within the same word.
- The result of `selector` is a single value that contains `B`, `R`, and `N`. `B` occupies the least significant 5 bits (with 32 being represented by 00000), `R` occupies the next 5 bits, and `N` the remaining 22 bits. `N` can be negative. Thus `N` can not exceed 2,097,151.
- `x := selector B : R : N` is equivalent to the three assignment sequence

```
selector 5 : 0 from x := B;
selector 5 : 5 from x := R = 32 -> 0, R;
selector 22 : 10 from x := N
```

◈

- `bit` is very similar to `byte`, except of course that it selects a single bit. Examples:

```
{ let x = 8362 * 94721, v = vec(10), y;
  bit 13 from x := 1;
  y := bit 0 from x;
  bit 200 of v := 1;
  y := bit 99 of v;
  bit 123 of v +:= 1;
 ...
```

◈

- `A -> B, C` is the conditional operator that is spelled `A ? B : C` in C++ and java.
- If `A` is `false` (i.e. zero), its value is the value of `C`, and `B` is never evaluated.
- If `A` is not zero, its value is the value of `B`, and `C` is never evaluated.


```
let x = 0x98765432;
out("%08x\n%08x\n%08x\n", x, x << 12, x >> 12)
```

```
98765432
65432000
00098765
```

- `<<` is the left shift operator.
- `A << B` is the value of `A` shifted `B` bits to the left. The leftmost `B` bits are lost, the rightmost `B` bits become zero.
- One hexadecimal digit equals four binary digits, so `<< 12` is a 3 digit hexadecimal shift.
- `>>` is the right shift operator.

```
out("%08x\n%08x\n%08x\n", x, x alshift 12, x arshift 12)
```

```
98765432
65432000
FFF98765
```

- `alshift` and `arshift` are the arithmetic left and right shift operators.
- `alshift` is exactly the same as `<<`, it is only included for completeness.
- `arshift` preserves the sign (most significant) bit, so for negative numbers the new bits appearing from the left are ones.
- `A alshift B` computes `A * 2`^{B}.
- `A arshift B` computes `A / 2`^{B}.

```
out("%08x\n%08x\n%08x\n", x, x rotl 12, x rotr 12)
```

```
98765432
65432987
43298765
```

- `rotl` and `rotr` are the bitwise left and right rotate operators.
- They perform normal shifts, but the bits that fall off at one end reappear at the other instead of zeros, so nothing is lost.


```
let a = 0b10011001110101100100111001100101,
    b = 0b11001010101110001010010011111100,
    s =  "--------------------------------";
out("%032b\n%032b\n%s\n%032b\n", A, B, S, A bitand B)
```

```
10011001110101100100111001100101
```

```
11001010101110001010010011111100
--------------------------------
10001000100100000000010001100100
```

- bitand is the bit-by-bit and operator, the same as & in C++ and java.
- Each bit in the result is 1 only when both corresponding bits in the operands are also 1.

```
    out("%032b\n%032b\n%s\n%032b\n", A, B, S, A bitor B)
```

```
10011001110101100100111001100101
11001010101110001010010011111100
--------------------------------
11011011111111101110111011111101
```

- bitor is the bit-by-bit or operator, the same as | in C++ and java.
- Each bit in the result is 1 when any of the corresponding bits in the operands is also 1.

```
    out("%032b\n%s\n%032b\n", A, S, bitnot A)
```

```
10011001110101100100111001100101
--------------------------------
01100110001010011011000110011010
```

- bitnot is the bit-by-bit not operator, the same as ~ in C++ and java.
- Each bit in the result is the opposite of the corresponding bit in the operand.

```
    out("%032b\n%032b\n%s\n%032b\n%032b\n",
                                    A, B, S, A eqv B, A neqv B);
```

```
10011001110101100100111001100101
11001010101110001010010011111100
--------------------------------
10101100100100010001010101100110
01010011011011101110101010011001
```

- eqv is the bit-by-bit logical equivalence operator.
- Each bit in the result is 1 when only when the corresponding bits in the operands are equal, either both 0 or both 1.
- neqv is the opposite of eqv, usually called “exclusive or”, and the same as ^ in C++ and java.
- Each bit in the result is 1 when only when the corresponding bits in the operands are different.


```
let count = 0;
for i = 1 to 32 do
{ if n bitand 1 then count +:= 1;
  n rotl:= 1 }
```

- That code fragment counts the number of ones in the binary representation of `N`, without changing the value of `N`.
- BEWARE! The bit-by-bit operators have the same priority as the corresponding logical operators, which are lower than the relational operators. That is not the same as in C++ and java.
- `if n bitand 1 = 1 then` ... would be interpreted as
  `if n bitand (1 = 1) then` ..., which is the same as
  `if n bitand true then` ..., which is the same as
  `if n bitand 0b11111111111111111111111111111111 then` ..., which is the same as
  `if n then` ...


```
manifest { pi = 3.1415927 }

let start() be
{ let width = 2.75, height = 6.125;
  let area = width #* height;
  let perimeter = (width #+ height) #* 2.0;
  let circarea = pi #* width #** 2;
  out("area = %f\n", area);
  out("perimeter = %f\n", perimeter);
  out("circle area = %f\n", circarea) }
```

```
area = +1.684375e+01
perimeter = +1.775000e+01
circumference = +2.375829e+01
```

- The floating point operators are the same as the integer operators, but with a `#` prefixed to their names.
- `#+`, `#-`, `#*`, and `#/` assume their operands are both floating point values, and produce a floating point result. If the operands are of the wrong type the results are meaningless.
- `#**` raises a floating point number to an integer power.
- `#<`, `#>`, `#<=`, `#>=`, `#=`, `#<>`, (and `#/=` and `#\=`) assume their operands are both floating point values, and produce result of either `true` or `false`. If the operands are of the wrong type the results are meaningless.
- Exception: `0` and `0.0` have the same representation, so are interchangeable.
- There are no special variations of the `%f` output format.


```
manifest { pi = 3.1415927 }

let start() be
{ let radius = 10;
  let circumf1 = 2.0 #* pi #* radius;
  let circumf2 = 2.0 #* pi #* float radius;
  let millpi = (fix (1000.0 #* pi)) * 1000;
  out("circumf1 = %f\n", circumf1);
  out("circumf2 = %f\n", circumf2);
```

```
    out("million pi about %d\n", millpi) }
```

```
circumf1 = +8.828181e-44
circumf2 = +6.283185e+01
million pi about 3141000
```

- See what happens when integers and floating point values are carelessly mixed?
- `float` takes an integer operand and converts it to floating point format.
- `fix` takes an floating point operand and converts it to integer format. It uses truncation towards zero, not rounding.
- `float` and `fix` are not functions, they are operators with high priority.


```
    let ia = 123, ib = -456;
    let fa = 3.2714e9, fb = -1.044e-11;
    let fc = #- fa;
    out("%d -> %d\n", ia, abs ia);
    out("%d -> %d\n", ib, abs ib);
    out("%f -> %f\n", fa, #abs fa);
    out("%f -> %f\n", fb, #abs fb);
    out("%f -> %f\n", fc, #abs fc) }
```

```
123 -> 123
-456 -> 456
+3.271399e+09 -> +3.271399e+09
-1.044000e-11 -> +1.044000e-11
-3.271399e+09 -> +3.271399e+09
```

- `abs` and `#abs` are also high priority operators. They find absolute values.
- The `e` notation for times-ten-to-the-power-of may be used in floating point constants. It is not an operator, it is part of the denotation of the number.
- `+`, `-`, and `#-` have unary versions too.
- A leading `-` attached to a numeric constant is part of the number, not an operator, so negative floating point numbers may be written in the normal way, without a #.

- Special operators are defined for unsigned integer arithmetic. They treat their operands as 32-bit magnitudes without a sign bit. They are:

  `##*  ##/  ##rem`
  `##<  ##>  ##<=  ##>=  ##=  ##<>  ##/=`

  There are no special operators for unsigned + or - because those operations work the same way as the signed versions. Just use + and - as usual.


```
    a := 7;
    b := 10;
    c := 1;
    d := b * valof { let f = 1;
                     for i = 1 to a do
                       f *:= i;
                     resultis f } + c;
```

- `valof` is a unary operator whose operand is a block - even if it is just one statement, it still needs the enclosing {}.
- The sample code sets `d` to ten times the factorial of 7 plus 1.
- `valof`s are of marginal usefulness.


```
let max(a, b) be test a>b then resultis a else resultis b;
let min(a, b) be test a<b then resultis a else resultis b;

let start() be
{ let x = 37, y = 12;
  let range = x %max y - x %min y;
  out("the range is %d\n", range) }
```

- The `%` sign allows a function to be used as an infix operator.
- `%` must be prefixed directly to a function name, it is not itself an operator, and can not be applied to an expression whose value is a function.
- `x %name y` means exactly the same as `name(x, y)`.
- `%name` has a higher priority than any other operator except the unary ones.

This was not part of traditional BCPL.

The `start` function may be given a parameter. It is similar to the `char * argv[]` parameter in C and C++ and to the `string [] args` parameter in Java. The value of the parameter will be a `nil`-terminated vector of strings supplied on the command line. To provide strings on the command line, use the `-c` flag when running the program. Follow the `-c` flag with a single string. Spaces will be taken as separators. e.g.

```
run prog -c inputs.txt
```

An example use:

```
$ cat cline.b
let start(argv) be
{ let i = 0;
  while argv ! i <> nil do
  { out("%d: \"%s\"\n", i, argv ! i);
    i +:= 1 } }

$ prep cline
ok
$run cline -c "one two three"
0: "one"
1: "two"
2: "three"
```

The escape sequences `\` (a `\` followed by a space), `\n`, `\t`, `\\`, `\'`, and `\"` may appear in the command line string, but remember that your unix shell also processes those characters.

This was not part of traditional BCPL.

If any program file contains a function called `pre_start`, it will be executed before the normal `start` function executes. If different `.b` files are compiled then linked together, then all or their `pre_start`s will execute (in an undertermined order) before the one `start`. `pre_start` will not receive any parameters. Beware: each file's `pre_start` is called after that file's globals have been initialised, but the order in which linked-together files are processed is undefined. This means that any file's `pre_start` and initialisations could override those of other files.

Assembly language may be incorporated directly into programs

```
let f(x, y) = x * 1000 + y

manifest { number = 123 }

let hippo = 0

let start() be
{ let cat = 7, goldfish = 3;
  assembly
  { load  r1, [<goldfish>]
    add   r1, <number>
    mul   r1, 10
    store r1, [<hippo>]
    push  77
    load  r1, [<cat>]
    mul   r1, [<goldfish>]
    push  r1
    push  4
    call  <f>
    add   sp, 3
    store r1, [<goldfish>] }
  out("hippo=%d, goldfish=%d\n", hippo, goldfish) }
```

- After the word `assembly`, the assembly language must be enclosed in { } even if it consists of only one statement.
- Inside the assembly language, names surrounded by < > will be replaced by their correct form in assembly language, but they must be the names of variables, functions, or manifest constants.
- Everything else is passed directly to the assembler without any further processing. Any errors will be reported by the assembler in a possibly unhelpful way after compilation is complete.
- The assembly language in the example is equivalent to
  ```
  hippo := (goldfish + number) * 10;
  goldfish := f(cat * goldfish, 77)
  ```
  the program prints `hippo=1260, goldfish=21077`
- The assembly language and machine architecture are documented separately.

- The calling convention is that parameter values are pushed in reverse order, then a value equal to `numbargs() * 2 + (lhs() -> 1, 0)` for the called function is pushed, then the function is called, then the stack is adjusted to remove the pushed values.

When programming for an operating system, you sometimes have to write interrupt handling functions and system calls. Both are handled in exactly the same way. These are just like ordinary functions except that they do everything they can to avoid changing anything that a user's program might be able to detect. For this to happen, such functions need to exit in a very particular way, not using the ordinary return statement. When you write an interrupt handler in BCPL, just insert the key word `handler` between the `let` and the function's name. When this is done, every return statement and the automatic return at the end of the function are automatically changed to the special required form. Interrupt handlers also do not return values, so inside a handler `resultis` statements are forbidden. This is a basic example of a handler for the timer interrupt:

```
let handler tim() be
{ outs("Ding!\n");
  $set_special_register(sr$timer, 3000) }
```

For full access to everything, an interrupt handler would ideally be declared like this:

```
let handler whatever(oldflags, interruptcode, info1, info2,
                     fp, sp, r12, r11, r10, r9, r8, r7, r6,
                     r5, r4, r3 ,r2, r1, r0) be
{ let pcaddr = @ oldflags - 2;
  ... }
```

With that setup, the parameters `r0` to `fp` will be equal to the values those registers had when the interrupt occurred. If you assign a new value to one of those parameters, the corresponding register will be set to that new value when control returns to the interrupted program. That would be a very bad thing for an interrupt handler to do, they are not supposed to make detectable changes. But it is a very useful thing for a system call. That is how they return results. Every function that returns a result just stores it in register R1 before exitting, so the assignment `R1 := 99` will make your syscall return 99 as its result. The variable `pcaddr` holds the address of the stack memory location where the interrupted code's program counter was stored. If an interrupt is unable to correct the problem that caused it (this is most likely with a division-by-zero interrupt) but you want the program to continue anyway, it should add 1 to that value.

```
! pcaddr +:= 1;
```

Normally, when an interrupt handler finishes, the instruction that was interrupted is given another chance and re-executed. If the problem hasn't been fixed, that will cause the same interrupt again immediately. Adding 1 to the program counter means that the offending instruction will be skipped.

Extremely long programs can cause trouble. Everything except for constants is accessed through its position relative to the value of some CPU register. Machine code instructions only have 16 bits to represent the numeric part of an operand, so things can only be accessed if they are between 32768 words before and 32767 words after their reference point. The reference point for local variables and parameters is the frame pointer register `FP`. The reference point for everything else is the program counter `PC`.

- When a local variable or parameter is too far away, the compiler is able to handle it, it produces extra instructions to make the access work correctly. This is automatic, the programmer does not have to do anything at all.
- When something else, like a function or label or global or static variable or string or table, the compiler has no way of knowing. Sometimes the assembler will be the component that detects the problem. In this case the assembler's error message will include the line number for the place in the original BCPL program where the access occurred. Assembler-detected errors also include the line number within the assembly language .ass file, which can be helpful.
- In other cases, such as when something imported from a library file is too far away, the problem may only be detected when the linker tries to make the final executable file. The linker's error message will contain the one thing it knows, the name of the symbol (variable name, function name, etc) the caused the problem. It is not able to determine where, in either the BCPL file or the assembly language file, the problem happened.
- If you are able to determine where the too-distant access occurred, just add the keyword `distant` immediately before the reference. So `outs("hello")` could become `distant outs("hello")`, `outs(distant "hello")`, or `distant outs(distant "hello")`.
- In cases where an assembler-generated error mentions a local label (they always consist of a capital `L` followed by a number, e.g. `L314`), that means you have put far too much code inside a loop or conditional, or between a `goto` and its target. That is just bad programming.
- If you do have to use `distant` a lot, you would probably be better off using indirect addressing and dynamic loading with jump tables for your libraries. That is quite complex.

The solution to the problem of a file getting too long is the creation of dynamically loaded libraries (`.dll` files). These are almost the same as executable files, except that they include a table of the addresses of those functions that the programmer decided to make public. If you know where the table is, and you know what the index of a function in that table is, you can call that function indirectly through the table without any need for `distant` declarations.

- The programmer creating a `dll` only needs to do three things: declare the functions that are to be made available using the keyword `dynamic`; define manifest constants for the functions' positions in the table; export those constants. Assume this example is in a file called `lib.b`:

```
export { idx_one, idx_two }
manifest { idx_one = 3, idx_two = 1 }
```

```
dynamic { one: idx_one, two: idx_two }
let one(x) be
  resultis x * x;
let two(a, b) be
  resultis (a - 1) * (b + 1);
```

- This file would be compiled in exactly the normal way. Whenever a file contains any `dynamic` declarations, a `.dll` file is created instead of the usual `.exe`. If a `.dll` file has a `start` function, it can be run by itself in the normal way in order to make debugging and testing convenient. On loading a `.dll`, the emulator adjusts its settings so that the user will not have to do anything unusual.
- The `dll` file is exactly the same as an `exe` file except that the first word is a `JUMP` instruction to make the program skip to the beginning of the executable code. The least significant 16 bits of this instruction are the number of entries in the function table. Following this first word is a list of the addresses of each of the dynamic functions. The addresses are relative to the beginning of the file.
- A program wishing to use this library only needs to import the indexes, read the file into memory, and calculate the address to call as a function. Imagining that somehow we know there is a lot of free memory starting at address 0x2000:

```
import "lib"
...
r := devctl(dc$tapeloadfile, "lib.dll", 0x2000);
max := (! 0x2000) bitand 0xFFFF;
functions := 0x2001;
x := (0x2000 + functions ! idx_one)(12);
```

  - `devctl` is a function from the `"io"` library that handles hardware operations. `dc$tapeloadfile` is the operation code for reading a file from the unix host system. The `dc$...` codes are listed in section 59. `r` will be negative if the operation failed, otherwise the number of words that were read.
  - `max` extracts the least significant 16 bits from the `JUMP` instruction at the beginning of the loaded file. For foolproof operations you would check that any function index you use is `>= 0` and `< max`. `lib` had two dynamic functions, but specified that their indexes would be 3 and 1, so `max` will be 4 because there is space for indexes 0, 1, 2, and 3.
  - `functions` is the address of the beginning of the function table.
  - `functions ! idx_one` is entry number 3 in the function table. That is the address of the function `one` relative to the beginning of its file, so the load address `0x2000` has to be added to it to find the true address of the function.
  - The last line of the example is just a normal function call with a parameter equal to `12`. The address of the function is calculated rather than just being a name as usual. `x` will be 144.
  - Normally you will not know where a large amount of free memory can be found. The `$memory_available` builtin function, described in section 55, will let you find where free memory is.

- Many builtins refer to areas of memory, they are called `src`, `dst`, `first`, `last`, `addr`, and `array` in the following. The values provided should be simple numeric addresses. The name of a `vec`, `table` or function, the result of a `newvec`, or one of those plus or minus some offset are all suitable values. `num`, `val`, `s`, and `size` should evaluate to ordinary integers. `var` should be a variable or something else that can be assigned to.
- The builtin functions are:
  - `$memory_move(dst, src, num)` - copy `num` consecutive words from `src` to `dst`.
  - `$memory_zero(dst, num)` - overwrite `num` consecutive words from `dst` with 0.
  - `r := $find_first_non_zero(first, last)` - scan all memory starting from `first` and running up to `last` (inclusive). As soon as a location is found that contains a non-zero value, the search stops and that location's address is returned as the value of `r`. If no such location is found, `r` is set to -1.
  - `r := $find_first_one_bit(val)` - The single word `val` is inspected starting with the leftmost (most significant) bit until a 1 is found, then `r` is set to the number of bits remaining to the right of that first 1. If there are no 1's, `r` is set to -1.
  - `r := $find_last_zero_bit(val)` - The single word `val` is inspected starting with the rightmost (least significant) bit until a 0 is found, then `r` is set to the number of bits remaining to the right of that first 0. If there are no 0's, `r` is set to -1.
  - `s := $adjust_selector(s, num)` - `s` must be a selector as created by the `selector`, `byte`, and `bit` operators. The selector is adjusted by moving it on by `num` units of its own size, taking into account the fact that selectors can not spread across two words. For example, if `s` is originally equal to `byte 17` and `num` is 5, then `s` will become equal to `byte 22`. `num` may also be negative. In `selector B : R : N`, `B` is the size. In `byte N` the size is 8, in `bit N` the size is 1.
  - `r := $memory_available(array, size)` - `array` must be an array or vector of at least `size` words. `size` should be an even number. The array is filled with pairs of numbers where the first is the beginning address of an area of memory, and the second is its end address. `array ! 0` is set to the first location occupied by anything loaded from the `.exe` file being executed, `array ! 1` is set to the last such address. After that, `array ! e` and `array ! (e + 1)`, where `e` is an even number, are set to the beginning and end addresses of all areas of memory that actually exist, in order of increasing address. The range between `array ! 0` and `array ! 1` will almost always be inside the range between `array ! 2` and `array ! 3`. All other ranges will be non-overlapping. Nothing is written beyond `array ! (size - 1)`. `r` is set to the value `size` should have had to cover all ranges of existing memory exactly.
    Example:
    ```
    manifest { size = 10 }
    let v = vec(size);
    let r = $memory_available(v, size);
    out("The program uses addresses %08X to %08X\n", v ! 0, v ! 1);
    if r > size then
    { out("for complete information, size should be %d\n", r);
      r := size }
    for i = 2 to r - 1 by 2 do
    ```

```
        out("Memory exists from %08X to %08X\n", v ! i, v ! (i + 1));
```

- `$clear_physical_page(addr)` - the 2048 word page of memory that includes physical address `addr` is set to all zeros.
- `r := $atomic_test_and_set(var)` - the variable `var` is read and set to 1 instantaneously. `r` is set to the original value that was read. `var` should be the variable itself, not a pointer to it.
- `$set_flags_and_jump(val, dst)` - all of the CPU flags are set to the value of the bit in their position within `val`, so `val` might be `(1 << flag$vm) + (1 << flag$r)` for example, and the program jumps to `dest` in the manner of a `goto`, simultaneously. `dest` must be a pointer to some executable code, possibly a function name, possibly a vector or other address containing data read from an .exe file. This is the last step in turning virtual memory on, `dest` should be a physical address.
- `r := $get_special_register(num)` - `r` is set to the value of the special CPU register indicated by `num`, typically a builtin name beginning with `sr$`, see the next section.
- `$set_special_register(num, val)` - The special CPU register indicated by `num`, typically a builtin name beginning with `sr$`, is set to `val`.
- `r := $get_flag(num)` - `r` is set to the one-bit value of the CPU flag indicated by `num`, typically a builtin name beginning with `flag$`, see the next section.
- `$set_flag(num, val)` - The CPU flag indicated by `num`, typically a builtin name beginning with `flag$`, is set to `val`.
- `$enable_interrupts()` - With no parameters, interrupt processing is enabled: the CPU's IP flag is set to zero. `$enable_interrupts(addr)` - With one parameter, first the special register INTVEC is set to the address given, then interrupt processing is enabled
- `$pause(num)` - The CPU pauses for `num` milliseconds without consuming any resources.
- `$debug(num)` - The program stops and goes into single-step mode, the instruction about to be executed will be displayed as `" BREAK num"`. Press enter to execute the instruction or 'r' to continue running normally. See the documentation on the emulator for more options.
- `$file_name()` - Evaluates to a string giving the name of the BCPL file that this call appeared in.
- `$line_number()` - Returns an integer being the line number that this call appeared on in the BCPL file.
- `$get_sp(), $get_fp(), $get_pc()` - Return the value of the stack pointer, frame pointer, or program counter.
- `$set_sp(num), $set_fp(num), $set_pc(num)` - Set the stack pointer, frame pointer, or program counter to the value of `num`.
- `$sign_extend(val, num)` - `val` is treated as a signed value `num` bits long and is stretched to the full 32 bits of a word. In a `num`-bit value, bit (`num` - 1) is the sign bit. This bit's value is copied over all the bits to the left of it. The result is returned.
- `$bits_in(val)` - Returns the number of bits required to hold `val`. If `val` is 5, that is 101 in binary so the result would be three. If `val` is 618, that is 1001101010 in binary so the result would be ten. It is an approximation of the base-two logarithm. Zero is considered to be zero bits long, and negative numbers will be 32 bits long.

- The compiler has a number of built-in constants that are used to identify some hardware-specific things. The all have a $ sign in their names, not at the beginning. Programmer-defined identifiers may not contain $ signs.
  - The following refer to individual one bit CPU flags by number: `flag$r`, `flag$z`, `flag$n`, `flag$sys`, `flag$ip`, `flag$vm`, `flag$int`, and as bit masks: `flag$maskr`, `flag$maskz`, `flag$maskn`, `flag$masksys`, `flag$maskip`, `flag$maskvm`, `flag$maskint`.
  - The following refer to special CPU registers: `sr$flags`, `sr$pdbr`, `sr$intvec`, `sr$cgbr`, `sr$cglen`, `sr$debug`, `sr$timer`, `sr$syssp`, `sr$sysfp`, `sr$usrsp`, `sr$usrfp`, `sr$watch`, `sr$exitcode`, `sr$ipl`, `sr$emgret`.
  - The following are interrupt numbers: `int$none`, `int$memory`, `int$pagefault`, `int$pagefault2`, `int$unimpop`, `int$halt`, `int$divzero`, `int$unwrop`, `int$timer`, `int$privop`, `int$keybd`, `int$badcall`, `int$pagepriv`, `int$debug`, `int$intrfault`, `int$sysstkfl`, `int$badop`, `int$watch`.
  - The following are device control operations: `dc$disccheck`, `dc$discread`, `dc$discwrite`, `dc$discclear`, `dc$tapecheck`, `dc$taperead`, `dc$tapewrite`, `dc$taperewind`, `dc$tapeload`, `dc$tapeloadfile`, `dc$tapeunload`, `dc$termin`, `dc$termout`, `dc$seconds`, `dc$useconds`, `dc$datetime`, `dc$netss`, `dc$netsend`, `dc$netrecv`, `dc$tapelength`.
  - The following apply to the virtual memory system, six numbers: `page$validbitnum`, `page$systembitnum`, `page$size`, `page$tablenumbits`, `page$pageintablebits`, `page$pagenumbits`, `page$offsetbits`, six bit masks: `page$validmask`, `page$systemmask`, `page$tablenummask`, `page$pageintablemask`, `page$pagenummask`, `page$offsetmask`, and four selectors: `page$tablenumsel`, `page$pageintablesel`, `page$pagenumsel`, `page$offsetsel`.
  - The following two numbers are for use with the disc system: `disc$blockwords`, `disc$blockbytes`.
  - The following are error codes returned by the `devctl` library function: `err$badcode`, `err$readparams`, `err$devnumber`, `err$position`, `err$memory`, `err$devfailed`, `err$notfound`, `err$badparam`, `err$inuse`, `err$cantcreate`, `err$nodata`.

Summary of Priorities within Expressions

priority			section
17, highest	`distant` constants, identifiers, parenthesised subexpressions, `valof` block		55 4 46
16	`F(A, B, C, ...)`	function calls	20, 56
15	`+`, `-`, `#-`, `not`, `~`,	unary operators	44 8

	bitnot, !, @, abs, #abs, float, fix		40 30 44 43
14	%name	infix function call	46
13	!	array access	30
12	**, #**	to the power of	32, 42
11	*, #*, /, #/, rem ##*, ##/, ##rem	multiplicative	14, 42, 45
10	+, #+, -, #-	additive	42
9	selector, byte, from, of, bit	bit ranges	36 - 38
8	<<, >>, alshift, arshift, rotl, rotr	shifts and rotations	39
7	<, >, <=, >=, <>, /=, \=, #<, #>, #<=, #>=, #<>, #/=, #\= ##<, ##>, ##<=, ##>=, ##=, ##<>, ##/=	relational	9 42 45 48
6	/\, bitand	conjunctions	8, 40
5	\/, bitor	disjunctions	8, 40
4	eqv	equivalence	40
3	neqv	exclusive or	40
2	-> ,	conditional	37
1, lowest	table	tables	38

Functions in the library “io”

`out(format, ...)`
See sections 1, 2, 5, 6, and 32.

`outch(c)`	print a single character
`outno(n)`	print a single integer in decimal
`outhex(n)`	print a single integer in hexadecimal
`outbin(n)`	print a single integer in binary
`outf(f)`	print a single floating point number
`outs(s)`	print a single string
`outsv(s)`	print a single string, with every character explicitly visible

These functions add nothing. The are the helper functions used by `out()`.

`floatformat(val, fmt, str)`
`val` should be a floating point number, `fmt` should be a string containing exactly a format that C's `printf` function could use to format a floating point number, and `str` should be a vector or other memory address where there is enough space to store the entire string that `printf` would have printed. The address of the string is also returned so that this can be conveniently used for printing, e.g.

```
outs(floatformat(pi, "%12.6f\n", str))
```

`inch()`

Read a single character from the input stream, return its ASCII code. `inch()` does not return a value until a whole line has been typed and terminated with `ENTER`, apart from this the only characters given special treatment are `control-C` which stops a program, and backspace. The default buffer used to store the input until `ENTER` is pressed has space for only 100 characters.

`set_kb_buffer(V, size)`

`V` should be a vector `size` words long. `V` replaces the default buffer used by `inch()`, so that up to `size*4-5` characters may be typed before pressing `ENTER`.

`inno()`

Read an integer in decimal, return its value. Uses `inch()`.

`numbargs()`

Returns the number of parameters the current function was called with, see section 23.

`lhs()`

Returns true if the current function call was the left hand side of an assignment. See section 25.

`thiscall()`

Returns a reference to the current stack frame.

`returnto(sf, value)`

Returns from all active functions until the one represented by `sf` (previously obtained from `thiscall()`) is reached. `value` is used as the `resultis` value from the last exitted function. `value` is optional.

`seconds()`

Returns the number of seconds since midnight 1st January 2000, local time.

`useconds(v)`

The parameter must be a vector of at least two words. `v ! 0` is set to the number of seconds exactly as described for `seconds`. `v ! 1` is set to the current number of microseconds into that second.

`datetime(t, v)`

`t` is a time as obtained from `seconds()`, `v` must be a vector of at least 7 words. The time in `t` is decoded as follows:

`v ! 0` := year
`v ! 1` := month, 1-12
`v ! 2` := day, 1-31
`v ! 3` := day of week, 0 = Sunday
`v ! 4` := hour, 0-23
`v ! 5` := minute, 0-59
`v ! 6` := second, 0-59

`datetime2(tsec, tusec, v)`

The date and time represented by `tsec` and `tusec` (which should be values as produced by `useconds`) sre stored in compressed form in v. v must be a vector of at least 2 words. If `tsec` is -1, the current date and time are used.

```
v ! 0 :  13 most significant bits = year
         4 next bits = month
         5 next bits = day
         3 next bits = day of week, 0 = Sunday
         7 least significant bits not used
v ! 1 :  5 most significant bits = hour
         6 next bits = minute
         6 next bits = second
         10 next bits = milliseconds
         5 least significant bits not used
```

The library also defines these selectors `dt2_year`, `dt2_month`, `dt2_day`, `dt2_dow`, `dt2_hour`, `dt_minute`, `dt2_second`, and `dt2_millisec`, which may be used to extract these values, e.g. `dt2_minute of v` would produce the minutes value.

`strlen(s)`

Returns the length in characters of the string `s`, not including the zero terminator.

`random(max)`

Returns a random integer between 0 and `max`, inclusive.

`devctl(op, arg1, arg2, arg3,...)`

Input/output device control functions, see section 50.

`devctlv(v)`

Has the same functionality as `devctl()`, but the parameters `op`, `arg1`, `arg2`, etc are provided as elements 0, 1, 2, etc of the vector `v`.

`netaddresstostring(addr, str)`

`addr` should be the two vector containing an IP address as returned by the `dc$netss` command to `devctl()`. `str` should be a vector of at least 6 words. It is filled with a human readable rendition of the given IP address, e.g. 129.171.33.6.188.233.

`stringtonetaddress(str, addr)`

`str` Should be a string containing the human readable version of an IP address, e.g. "129.171.33.6.188.233". `addr` should be a vector of two words, it will be filled with the properly encoded version of an IP address for use with the network-related `devctl` operations.

Unit numbers identify individual discs or magnetic tape drives, numbered from 1 up. Tapes and Discs are numbered independently, there can be a tape number 1 and a disc number 1 at the same time. All of these operations may return a negative value to indicate an error.

`op = dc$termin`
`arg1` = maximum number of characters to read
`arg2` = memory address
As many characters as are inside the hardware keyboard buffer, up to the given maximum, are copied into memory packed four per word and zero terminated to form a proper string. Space for the zero terminator must be included in the value of `arg1`. The number of characters actually read, not including the terminator, is returned. There must be at least `arg1 / 4 + 1` words available staring from `arg2`.

`op = dc$termout`
`arg1` = maximum number of characters to write
`arg2` = memory address
`arg1` characters are taken from memory at the given address, which must contain characters packed four to a word as in a normal string, and displayed on the monitor. If a zero character is encountered, the process terminates immediately. The number of characters written is returned.

`op = dc$disccheck`
`arg1` = unit number
If the disc unit is available returns the total number of blocks it contains
Otherwise returns 0

`op = dc$discread`
`arg1` = unit number
`arg2` = first block number
`arg3` = memory address
The indicated block (512 bytes or 128 words) are is read directly into memory starting at the address given. On success returns 1. On failure returns a negative error code.

`op = dc$discwrite`
`arg1` = unit number
`arg2` = first block number
`arg3` = memory address
128 words of memory starting from the address given are written directly into the indicated block. On success returns 1. On failure returns a negative error code.

`op = dc$discclear`
`arg1` = unit number
Completely erases the entire contents of the disc. On failure returns a negative error code. On success returns number of blocks in the disc.

`op = dc$tapecheck`

arg1 = unit number

If the disc unit is available returns 'R' or 'W' indicating whether the tape was mounted for reading or for writing. Returns 0 if not available.

op = dc$tapelength

arg1 = unit number

The length (in bytes) of the real file currently loaded on the given magnetic tape unit is returned.

op = dc$taperead

arg1 = unit number

arg2 = memory address

One block is read from the tape unit directly into memory at the address given, returns the number of bytes in the block. All blocks are 512 bytes except that the last block of a tape may be shorter.

op = dc$tapewrite

arg1 = unit number

arg2 = memory address

arg3 = size of block in bytes

The indicated number of bytes, which must not exceed 512, of memory starting from the address given are written directly as a single block to the tape. Returns the number of bytes written.

op = dc$tapeloadfile

arg1 = file name

arg2 = memory address

The entire named file is copied from the host unix system into memory starting at the given address. No tape unit number is needed, the file does not need to be loaded first or unloaded afterwards. On success returns the number of words read.

op = dc$taperewind

arg1 = unit number

Rewinds the tape to the beginning.

op = dc$tapeload

arg1 = unit number

arg2 = string, the filename of the tape

arg3 = the letter 'R' or 'W'.

The named file is made available as a magnetic tape unit. The letter R indicates that it is read only, and W that it is write only. Returns 1 on success, or a negative error code.

op = dc$tapeunload

arg1 = unit number

The tape is removed from the drive, returns 1 for success or a negative error code.

op = dc$netss

arg1 = unit number
arg2: 1 = turn on, 0 = turn off.
arg3: = address, a vector of two words.
Addresses are 6 byte values, based on IP addresses, e.g. 129.171.33.6.210.4
On calling netss, the first word should be zero, and the second word can be zero to request an ephemeral port, or N to request specific port N.
On return from netss, the two words will be filled with the actual local 'IP' address (6 bytes).

op = dc$netsend
arg1 = unit number
arg2 = to-address, a vector of two words as returned by NETSS
arg3 = pointer to buffer containing the bytes to be sent.
arg4 = number of bytes to send. Up to 1024 bytes may be sent.
The bytes are sent to the destination address.

op = dc$netrecv
arg1 = unit number
arg2 = from-address, a vector of two words
arg3 = pointer to buffer to contain the bytes received
arg4 = the length of the vector in words
If no bytes have been received, err$nodata (minus eleven) is returned.
If any bytes are received, they (up to 1024 of them) are stored in arg3, and their number is returned by devctl. The from-address vector is filled with the 'IP' address of the sender.

op = dc$seconds
The number of seconds since 00:00 on 1st January 2000 is returned.

op = dc$useconds
arg1 = pointer to two words of memory
The number of seconds (as described for dc$seconds) is stored in the first word, and the number of microseconds into the current second is stored in the second word.

op = dc$datetime
arg1 = a number of seconds as returned by dc$seconds.
arg2 = pointer to seven words of memory
If the number of seconds given is -1, the current date and time are used.
The given time is decoded into human-understandable components, which are written into the seven words as follows. 0: year, 1: month, 2: day of month, 3: day of week (0 means Sunday), 4: hour, 5: minute, 6: second.

op = dc$floatformat
arg1 = any floating point number.
arg2 = a string containing a printf format suitable for floating point numbers.
arg3 = a pointer to memory where the resultant string will be stored.
This provides the functionality of the floatformat library function described in section 58. Slightly more detail is provided there.

Compiling and running on rabbit.

The program should be in a file whose name ends with `.b`. Here it is

```
$ ls hello.*
hello.b
$ cat hello.b

import "io"

let start() be
{ out("Greetings, Human.\n");
  out("Now go away and leave me alone.\n") }
```

First run the compiler (you don't need to type the .b in the file name). It creates an assembly language file whose name ends with `.ass`. The `.ass` file is human readable, you can look in it if you want.

```
$ bcpl hello
$ ls hello.*
hello.ass   hello.b
```

Then run the assembler. It produces an object file which is not human readable.

```
$ assemble hello
$ ls hello.*
hello.ass   hello.b   hello.obj
```

Then use the linker to combine your object file with the object files for the libraries it uses. The result is an executable image file whose name ends in `.exe`

```
$ linker hello
$ ls hello.*
hello.ass   hello.b   hello.exe   hello.obj
```

Fortunately there is a single command that does all three of those steps for you. It is called `prep`, short for prepare.

Finally tell the emulator to start up and run that executable file

```
$ run hello
Greetings, Human.
Now go away and leave me alone.
```

So all that could have been done with just

```
$ prep hello
$ run hello
```

If your program goes wrong while it is running, control-C will stop it, but you'll need to type a few more control-Cs to stop the emulator too.

Additional files to help with debugging are also produced. `hello.das` contains debugging information provided by the assembler and `hello.dli` contains such information from the linker. The emulator uses them automatically when the occasion arises.

Other files or features may be enabled by providing flags to the commands:

- `-la` creates an assembly listing file whose name ends in `.las`. It shows every instruction that went into the program, along with its hexadecimal encoding and information on where it will be when the program is running.
- `-ll` creates an linker listing file whose name ends in `.lli`. It shows all of the things that were taken from different .obj files and gives information on where they will be in the running program.
- `-d` suppresses the creation of the debugging information files.
- `-exe` only works for very simple programs that import no libraries. The assembler produces an executable file directly and the linking phase is skipped.