Contents

1 Introduction

Angort[1] is a stack-based concatenative programming language, based on the venerable Forth, with some extra features. These include:

- variables and function parameters;
- higher order functions with (nearly) full lexical closure;
- lists and hashes (dictionaries) with garbage collection;
- native C++ plugin architecture.

1.1 Rationale and history

Angort was initially written out of curiosity, driven by my need to understand how high-level constructions such as garbage-collected values and closures worked down at the machine code level. As a veteran assembler and C/C++ programmer, I found languages like Python and even Java difficult to “trust”, in a sense, without this understanding.

My first attempt was a language typical of the Algol lineage, Lana. While successful, Lana wasn’t particularly interesting and so I abandoned it. Later, I found I needed a control language for a robotics project: the ExoMars rover locomotion prototype, Blodwen. The Blodwen control system used a fairly complex C++ API, which was appropriate for many tasks but not for ad-hoc control or experiment scripting. My initial instinct was (as always) to quickly write an interpreter for a Forth-like language.

As the project progressed, I found the language an interesting platform for the high-level features mentioned above. Notably, my final year project required a subsumption system *à la* Brooks, and I found this much easier to write in the new language than in C++. Prompted by this discovery I continued to work with and add features to the language over the next year, and was

[1]The name is an entirely random pair of syllables, it has no significance.

surprised by how powerful a stack-based language with anonymous functions and collections could be. As I started my Ph.D. I found working with Angort a natural way to script experiments, particularly once I found a good set of paradigms for interfacing with C++ code, and so continued working with it.

1.2 Brief examples

Here are some examples. First, the familiar recursive quicksort algorithm:

```
# Colon at the start of a line introduces a named function definition.
# This function has one parameter "lst" and one local variable "piv".
:qs |lst:piv|
    # if length of list is <=1...
    ?lst len 1 <= if
        # return the list itself
        ?lst
    else
        # otherwise remove the first item as the pivot
        ?lst pop !piv
        # quicksort the list of items less than the pivot
        ?lst (?piv <) filter qs
        # add the pivot to that list
        [?piv] +
        # quicksort the list of items greater than the pivot
        # and add it.
        ?lst (?piv >=) filter qs +
        # resulting list is left on the stack, and so returned.
    then;
```

This could be used thus:

```
[0,6,2,3,7,8] qs each{i.}
```

to sort and print the given list of integers.

Next, a program to read a CSV file and print the sums of all the columns:

```
# load the CSV plugin but do not import the symbols into the
# default namespace, so we have to access them with csv$...
`csv library drop

# load the CSV file - creates a CSV reader object which will read
# into a list of hashes where all columns are floats, and uses it
# to read the file into the global "CSV".

[% `types "f" ] csv$make "magic.log" csv$read !CSV

# make a list of keys from the first item in CSV.
[] ?CSV fst each {i,} !Keys

# create a "slug" (an anonymous function which runs immediately,
# to provide local variables and multiline flow control)
# with a local variable i.
(|:i|
    # for each key, store the iterator value in "i" so it's
    # accessible within a closure
    ?Keys each { i!i
        # create a pair consisting of the iterator (i.e. key name)
        [i,
         # and the sum of the key's values, done by using map/reduce:
         # the map extracts the key's values, the reduce performs the sum.
         0 ?CSV (?i swap get) map (+) reduce
         ]
        # format the pair and print it.
        "%s %f" format.
    }
)@ quit # run the slug and quit
```

Angort combines the power and ease of a modern dynamic language with the convenience of a Forth-like language, and has been used in various applications including:

- building command/control/monitoring environments for robotic systems;
- scripting experiment runs for neural networks;
- writing fairly complex data analysis tools;
- generating visualisations of neural networks and other data;
- algorithmically generated music;
- simple 2D games.

1.3 Creating control languages

Because the interpreter is interactive and drops back to a command prompt upon completion of a script, it is very useful for building domain-specific control languages. For example, our ExoMars locomotion prototype is controlled with Angort, and its script includes the following definitions:

```
# define a constant "wheels" holding a range from 1-6 inclusive

range 1 7 const wheels

# define a new function "d" with a single parameter "speed"

:d |speed:|
    # set a help text for this function
    :"(speed --) set speed of all drive motors"

    # for each wheel, set the required speed to the value
    # of the parameter
    wheels each {
        ?speed i!drive
    }
;

# slightly more complex function for steering

:t |angle:|
    :"(angle --) turn front wheels one way, back wheels opposite way"

    ?angle dup 1!steer 2!steer
    0 0 3!steer 4!steer
    ?angle neg dup 5!steer 6!steer
;

# define a function to stop the rover by setting all speeds to zero
:s 0 d;
```

Once these words are defined we can steer the robot in real time with commands like:

```
2500 d
30 t
s
```

These will set the rover speed to 2500, turn it to 30 degrees, and stop it respectively. We can also directly type things like:

```
wheels each { i dactual .}
```

which will print the actual speeds of all the wheels. In the examples given so far, functions such as `dactual`, `!drive` and `?drive` are links to native C++ code[2]: it is very easy to interface Angort with C++.

1.4 Functional programming

It's also possible to perform some functional programming with anonymous functions:

```
:sum |list:| 0 ?list (+) reduce;
```

will allow us to sum a list and print the results:

[2] The latter two functions use *properties*: syntactically they look like variable sets and gets but actually cause C++ code to run, setting and getting the motor drive speed in the robot.

```
[1,2,3,4,5] sum .
```

or even:

```
1 1001 range (dup*) map sum .
```

to print the sum of the squares of the first 1000 integers. As can be seen, Angort is a very terse language.

1.5 Downloading and building Angort

The requirements to build Angort are

- (probably) Linux
- the BSD `libedit` library[3] – this can be found in the Ubuntu package `libedit-dev` or source at `http://thrysoee.dk/editline/`
- CMake
- Perl (to parse the function definition files)

Angort can be downloaded from `https://github.com/jimfinnis/angort` . Once downloaded, it can be built and installed with the following commands (from inside the top-level Angort directory):

```
mkdir build
cd build
cmake .. -DCMAKE_BUILD_TYPE=Release
make
sudo make install
```

The interpreter will then be installed, typically as `/usr/local/bin/angort`. A small set of Angort libraries (i.e. libraries written in Angort) will also be installed into `/usr/local/share/angort`.

Note that you can normally parallelise the build with `make -j`, but if you wish to rebuild this manual, you should run make without the `-j` from an clean build directory. The manual build process uses files generated from the C++ which the parallelisation makes rather a mess of.

A set of native C++ plugin libraries can also be downloaded from `https://github.com/jimfinnis/angortplugins`. Once Angort has been installed, these can be built and installed with

```
./buildall
sudo ./install
```

They will also be installed into `/usr/local/share/angort`. Using `reallybuildall` will attempt to build extra libraries which require additional packages, such a a CURL interface, SDL graphics support and JACK MIDI support. Many of these have been written for somewhat esoteric purposes as I have needed them.

[3] I'm not using the GNU `readline` library for two reasons: firstly, it doesn't permit multiple instances, which are needed for the integral debugger; and secondly it infects software with the GPL and has no linking exception.

2 Keywords and tokens

The Angort language contains the following tokens – strings and characters which are interpreted by the parser directly. Most of these are converted one-to-one into bytecode instructions, although a few are syntactic markers and affixes (such as the single quote for symbols and the vertical bar "|" for delimiting the function local variable definitions).

!=	<=	>=	?`	!`
!+	!-	<<	>>	??
{	}	(	)	[
]	,	<	>	=
+	/	-	*	;
.	!	?	@	:
#	&	|	%	`
if	then	else	leave	dup
call	global	swap	drop	not
and	or	ifleave	const	over
each	include	stop	cmp	package
require	private	public	def	defconst
recurse	self	cases	case	otherwise
sourceline	yield	try	catch	catchall
throw	inc	dec		

Everything else is either a literal (such as `35.0` or `"wibble"`), a local variable name, or the name of a function or global variable[4]. Functions and globals are identifiers bound to a value, stored in a namespace (see Sec. 17).

3 Getting started: immediate mode

This section will describe the basic concepts behind the language, such as reverse Polish notation and the stack, and introduce using the language in immediate mode.

3.1 Immediate mode

In immediate mode, each line of Angort is compiled and run straight away. It is the default mode of the Angort interpreter, even with script files as we will see below. Angort will be in this mode when a script completes without `quit` being called. In immediate mode, when a script has not been provided on the command line or a script has completed without quitting, Angort shows a prompt like this:

```
1|0 >
```

The two numbers are the number of garbage-collectable objects in the system and the number of items on the stack, respectively. The interpreter is in "immediate mode", as opposed to "compilation mode" — any text entered will be compiled to bytecode and run when enter is pressed, rather than being added to a function definition.

3.2 Reverse Polish notation and the stack

Angort is a stack-based language: there is a single stack containing values, and most functions change the contents of the stack in some way. For example,

```
3
```

[4]Functions are actually global variables, see Sec. 18.

by itself will just put the value 3 on the stack. Then

```
.
```

will pop the value from the top of the stack and print it.

```
3 4 + .
```

will push 3 and 4 onto the stack, then add them together replacing them with 7, and then print the 7. This kind of notation is often referred to as **reverse Polish notation** (RPN) as opposed to the more common **infix** notation. More complex expressions are built out of sequences of operations on the stack. For example, the expression

$$\sin(5 + 32 + \sqrt{43 \times 12})$$

would be written as

```
43 12 * sqrt 32 + 5 + sin
```

which can be read as "multiply 43 and 12, find the square root, add 32, add 5, find the sine." This is a little difficult at first, but rapidly becomes second nature[5]

3.3 Running scripts

Scripts are traditionally[6] given the extension `.ang`, but this appears nowhere in the codebase. To run a script, create a file of Angort function definitions and commands and pass it as an argument to the `angort` command. Angort will still run the script in immediate mode: each line will be parsed and run in order. If a function definition is encounted, Angort will switch to compilation mode for the duration of the function.

Again, in immediate mode (i.e. outside a function) each line of Angort is parsed and run immediately. This means that multiline flow control constructs (such as loops and `if..then`) will not work unless they are inside a function. This is easily solved by using anonymous functions, as described in Section 9.2.

At the end of a script, Angort will be in immediate mode and will ask for more lines, with the prompt described above. This is often useful when Angort is used to provide a domain-specific control language: the script sets up a large number of commands and perhaps initialises native C++ libraries, and then waits for the user's commands. If this is not required, terminate the script with the `quit` command. This will cause Angort to exit with the return code 0.

3.4 Words and functions

Most of the Angort language consists of words which each perform a single, isolated task – there is very little syntax. Even common "syntactic" keywords like `if` (for conditions), `[]` (for creating lists) and `+` (for addition) operate primarily through stack operations, although `if` does some more complex things at compile time (such as calculating jump offsets). Code in Angort largely consists of many of these words chained together to perform a task. These words can be combined into user-defined functions.

In general, I will use "word" to refer to a single Angort built-in token, particularily if it compiles to a single opcode; while "function" will refer to user-defined functions. However, I may vary – there is no real distinction, since an Angort function call is itself a single opcode.

[5]Most old calculators work using RPN, and quite a few of the more powerful programmables still do.

[6]A somewhat unusual word to use when there is currently only one user of the language.

3.5 Types and literals

Angort is a dynamically typed language, with type coercion (weak typing). The following types are available:

Name[1]	**Definition**	**Hashable**[2]	**Example of literal**
`none`	The nil object, specifying "no value"	N	`none`
`integer`	32 or 64-bit integers depending on architecture	Y	`5045`
`long`	depends on architecture	Y	`5045l,32L,0ffx, 45ao, 10111b, 3200ffffhl`
`float`	32-bit floats	Y	`54.0`
`double`	64-bit floats	Y	`54.0l,32.2L`
`string`	Strings of characters	Y	`"Hello there"`[3]
`symbol`	Single-word strings, internally stored as integers and generally used as hash keys	Y	`` `foo ``
Callable[4]	Angort and C++ functions	N	`( dup * 1 + ), ?myfunction`
`range`	A range of integers between two values with optional step	Y	`0 4 range`[5]
`frange`	A range of floats between two values with step	Y	`0 4 0.1 frange`
`list`	A array/list of values[6]	N	`[1,2,"foo"]`
`hash`	A map of values to values implemented as a hash table, where keys can strings, symbols, integers or floats	N	``[% `foo 1, "fish" "fish"]``

There are some other types used internally, such as the types for iterators and deleted objects in a hash.

Integers can also have a base character, 'b', 'x'/'h', 'o' or 'd', which goes before the optional 'l' for 'long.' **Note that** all numbers must start with a digit, so a leading zero will be required for some hex numbers (e.g. '0ffh' not 'ffh').

3.6 Booleans

Booleans in Angort are represented by integers – zero is false, non-zero is true. However, functions and words which nominally take a boolean value can accept any type, which is converted to a boolean thus:

- integers are false if zero, true otherwise
- "none" values are false
- values of any other type are true

This avoids the need for such constructions as

```
isnone not if "it exists!" then
```

Note: This feature arrived in version 3.1.0, and may not be implemented fully in some plugin libraries. Use integers if in doubt.

[1]This is the name of the type as returned by the `type` word (see Sec. 3.8).
[2]Indicates that values of this type may be used as keys in a hash (see Sec. 12).
[3]Multi-line strings can be defined using a "heredoc" similar to those in languages like bash, PHP and Python. See Sec. 3.7.
[4]Actually conflates three types internally: `native` for C++ functions; `codeblock` for simple Angort functions; and *closure*, whose values consist of a codeblock and some stored variables. Check if a value is callable with `iscallable`.
[5]This is actually a call to the `range` function which takes two integers to create a range.
[6]Stored internally as an automatically resized array rather than a linked list.

3.7 Heredoc strings

Multi-line strings can be stacked using “heredoc” syntax. A heredoc block starts and ends with a marker, which is a double hyphen followed by an identifier. Both markers must be at the start of a line. Everything between the markers, excluding the trailing newline character, will be put into a single string on the stack. Here is an example:

```
# the heredoc start marker. Markers must be on a line by themselves
# with no comment or trailing whitespace.
--FOO
This is a heredoc, which can contain quotes both 'single' and "double".
It can even contain apparent heredoc markers, which will be ignored
if they don't have the same identifier.
--FOO

# The string is on the stack, so print it.
.
```

3.8 Retrieving value types

It is possible to retrieve the type of a value, or check that it is one of a particular set of types, using the following words:

name	**description**	**stack picture**
type	get name of value’s type as a symbol	(val – symbol)
isnumber	1 if value’s type is numeric, else 0	(val – int)
isnone	1 if value’s type is none, else 0	(val – int)
iscallable	1 if value’s type is codeblock, closure or native function, else 0	(val – int)

4 Command line options

Angort has the following command line options, each of which must be a separate argument: one cannot put multiple boolean switches in a single argument. I don't use `getopt`, and this helps with Angort programs parsing their own arguments. The arguments handled by Angort (and stripped out by the option handler to produce `args` as described below) are:

- `-d` : debugging type 1 (show opcodes and stack on each instruction);
- `-D` : debugging type 2 (show tokeniser trace);
- `-lNAME` : load plugin library name – this is a file; in the shared plugin path (see Sec. 19) or on a relative path, ending in `.angso`. The `.angso` suffix which should not be supplied;
- `-if` : import `future` (see Sec. 17.6);
- `-id` : import `deprecated`;
- `-e` : execute the first non-option argument as an Angort string, rather than loading and running a script;
- `-n` : the first two non-option arguments are two Angort; command strings. The first is run once, the second is run repeatedly for each line in `stdin`, with the each line being stacked prior to each run;
- `-` : all remaining arguments are skipped, and copied into the stripped arguments list (see below).

All arguments described above (with the natural exception of `-`) are copied into the "stripped argument list." This is the list which is returned by the `args` word. To retrieve the raw argument list (i.e. `argv` as passed to `main()`,) use the `env$rawargs` word.

5 Defining new functions

New functions are defined with code of the form

```
:functionname ... ;
```

A simple example is:

```
:square dup *;
```

This will define a function `square` which will duplicate the value on top of the stack, multiply the top two values (effectively squaring the number[7]) and exit, leaving the squared number on the stack. This can then be used thus:

```
3 square 4 square + .
```

which will print 25, i.e. $3^2 + 4^2$. **Note that leaving values behind on the stack on exiting a function is how we return values in Angort.** While defining a function, Angort is is compilation mode — functions will be converted to bytecode but added to the definition of the new function rather than executed immediately. Function definitions can therefore span more than one line:

```
:factorial |x:|
    ?x 1 = if
        1
    else
        ?x ?x 1 - factorial *
    then
;
```

This example uses variables, recursion and the Angort `if..else..then` construction. This latter in particular looks a little odd to programmers not familiar with Forth – please see Section 9.1 for more details.

5.1 Local variables and parameters

Until now, most of the code has been perfectly valid Forth[8]. However, the manipulation of stack required in Forth is a challenge (and modern computers have a little more memory), so Angort has a system of named parameters and local variables. These can be defined by putting a special block of the form

```
|param1,param2,param3... : local1,local2,local3|
```

after the function name in the definition. Locals and parameters are exactly the same internally, but the values of parameters are popped off the stack when the new function is called.

Once defined, locals (and parameters) can be read (i.e. pushed onto the stack) using a question mark followed by the variable name. Similarly, a local is written (i.e. a value popped off the stack and stored in the local) by using an exclamation mark. For example,

```
:pointless |x,y:z|
    ?x ?y + !z
;
```

will read the two arguments into the locals x and y, add them, store the result in the local z, and then exit, throwing everything away. Note that a function with parameters but no locals is defined by leaving the part after the colon empty:

```
:magnitude |x,y:|
    ?x dup *
    ?y dup * +
    sqrt
;
```

[7]This will produce an error if the value is not a number.

[8]With the exception of the factorial function, which uses local variables and recursion.

while leaving the part before the colon empty will define a function with locals but no parameters:

```
:countToTen |:count|
    0!count
    {
        ?count 1+ dup !count
        dup .
        10 = ifleave
    }
;
```

Here, the curly brackets enclose a loop, which is exited by the `ifleave` instruction when the stack top value is 10. See Section 9.4 for more details.

5.2 Incrementing and decrementing

It is possible to increment and decrement the value on top of the stack using the `inc` and `dec` words rather than using the less efficient `1+` and `1-` sequences. Additionally, rather than writing

```
?var inc !var
```

we can use a special shorthand for incrementing a variable:

```
!+var
```

As well as being more concise, it again produces more efficient code (a single opcode rather than three). We can also decrement:

```
!-var
```

This works with both locals and globals.

5.3 Parameter type checking

It is possible to specify the type required for each parameter by putting the type name after the parameter name, separated by a slash '/' thus:

```
:magnitude |x/float,y/float:|
    ?x dup *
    ?y dup * +
    sqrt
;
```

If the type of a parameter does not match, Angort will attempt to convert the value to the given type. If this fails, a type mismatch exception will be thrown. Note that the attempted conversion can lead to some unexpected results. For example,

```
"foo" 3 magnitude
```

will work, giving the result 3.0 . This is because the string "foo" will be converted to a float value of zero. However,

```
none 3 magnitude
```

will fail because `none` cannot be converted to any other type.

5.3.1 Stricter type checking (numbers and strings)

If the type specified is `numeric`, the parameter must be one of the numeric types – no conversion from string types is permitted. Specifying `stringstrict` will specify that the parameter must be a string or a symbol; again no conversion will be performed. For even stricter type checking, use the facilities described in Sec. 3.8.

5.4 Word documentation strings and stack pictures

Following the locals and parameters (or just the function name if there are none) there may be a function documentation string. It has the form

```
:"(before -- after) what the function does"
```

The section in brackets is known as a *stack picture*,[9] and describes the action of the function on the stack. The part before the hyphen shows the part of the stack removed by the function, with the stack top to the right, and the part after shows its replacement. For example:

```
:dist |x1,y1,x2,y2:|
    :"(x1 y1 x2 y2 -- distance) calculate the distance between two points"
    ?x1 ?x2 - dup *      # this is (x1-x2)^2
    ?y1 ?y2 - dup *      # this is (y1-y2)^2
    + sqrt               # sum them, and find the root
;
```

Note that the names in the picture are not necessarily those of variables: they are single word descriptions of values. Typically these will be variable names for the "before" part, but not for the "after" part (since the return values are simply what's left on the stack).

To print the documentation for a word, enter `??wordname` at the prompt. This will also work for most built-in Angort functions, but not all[10]. It will also work for any functions inside C++ plugin libraries (see Section 19). For example:

```
Angort version 2.9.2 (c) Jim Finnis 2012-2017
Use '??word' to get help on a word.
1|0 > ??redir
redir: (filename --) open a new file and redirect output to it.
 For more complex file output, use the IO library. On fail, throws
 ex$failed.
1|0 >
```

6 Basic built-in words

6.1 Stack manipulation

There exist a number of words whose sole purpose is to manipulate the stack. These words, and their stack pictures, are given in the table below.

name	stack picture
dup	(a – a a)
drop	(a –)
swap	(a b – b a)
over	(a b – a b a)

Note that this is a much smaller set than the set of Forth stack words, which includes `rot`, `roll`, `nip` and `tuck`. The local variable system makes such complex operations unnecessary.

6.2 Binary operators

In the following operations

- Booleans are represented by integers: nonzero is true, zero is false.

[9] Or sometimes a *stackflow symbol* – the concept is borrowed from Forth and the literature varies.

[10] This is because most Angort functions compile to calls to C++ functions stored as "native" values, which can have descriptions. Simpler operations, such as `if..then..else`, loop markers, arithmetic and the "." print operator, compile directly to bytecode opcodes. These cannot have descriptions.

- If one of the operands is a string, the other will be converted to a string. Only "+", "*" and comparison operators will be valid.
- If one of the operands is a long and the other is an integer, the integer will be converted to a long and the result will be a long.
- If one of the operands is a float and the other is an integer or long, that will be converted to a float and the result will be a float.
- If one of the operands is a double and the other is an integer, long, or float, that will will be converted to a double and the result will be a double.
- The comparison operators will do identity checks on objects (lists, ranges etc.), not deep comparisons.
- The comparison functions return booleans-as-integers, as described above.

name	**stack picture**	**side-effects and notes**
+	(a b – a+b)	
-	(a b – a-b)	
$*$	(a b – a*b)	ints are coerced to floats if one operand is an int, but not otherwise. "string int *" produces a repeated string, but "int string *" is invalid.
/	(a b – a/b)	
%	(a b – a%b)	integer remainder ("mod") operator
$>$	(a b – a$>$b)	string comparison works as expected, collections (lists and hashes) are invalid.
$<$	(a b – a$<$b)	string comparison works as expected, collections are invalid.
=	(a b – a=b)	string comparison checks for equality, not identity. Collection comparison (lists and hashes) test for identity only (i.e. true if the two operands are references to the same collection).
!=	(a b – a!=b)	see "=" above.
and	(a b – a$\wedge$b)	binary and – inputs are coerced to integers, nonzero is true
or	(a b – a$\vee$b)	binary or – inputs are coerced to integers, nonzero is true
band	(a b – a & b)	bitwise AND – inputs coerced to long, output is long
bor	(a b – a $\mid$ b)	bitwise OR – inputs coerced to long, output is long
bxor	(a b – a $\sim$ b)	bitwise XOR – inputs coerced to long, output is long

6.3 Unary operators

name	**stack picture**	**side-effects and notes**
not	(a – !a)	logical negation of a boolean (i.e. an integer)
neg	(a – -a)	arithmetic negation of float or integer
abs	(a – $\|a\|$)	absolute value
isnone	(a – bool)	true if value is `none`
bnot	(a – !b)	bitwise not – input coerced to long, output is long

6.4 Basic input and output

This section covers simple input and output over the standard IO streams. for file IO, the `io` library is required. Output can be redirected to a file with the `redir` word, however.

6.4.1 Simple input

To read a line from standard input (i.e. the keyboard unless Angort has been redirected) use `read`:

```
(|:s|
    {
        read !s
        ?s "quit" = ifleave
        ?s toint dup* .
    }
)@ quit
```

This will convert each line to an integer and print its square, unless "quit" is entered, which will terminate the program. Note that `read` will not add the trailing newline character.

The construction used here – putting the code inside `( ... )@` – allows local variables and control structures which extend over more than one line to be used outside a named function. It simply creates an anonymous function (with the brackets) and runs it immediately (with the `@` symbol). This is dealt with in more detail in Section 9.2.

6.4.2 Simple output

Three words cover most output situations:

- the "dot" word, written as ".", will pop the stack and print the value followed by a newline. This is shown in many of the examples above. The value is converted to a string for printing using a method internal to the type.
- The `p` word will do the same, but without the newline.
- The `nl` word will just print a newline.

6.4.3 Output to stderr

To output to the standard error stream, use the `errp` word to print a value without the newline, and `errnl` to print just a newline. There is no direct equivalent to ".", but this is achieved with the sequence `errp errnl`.

6.4.4 Output redirection

The `redir` word takes the name of a file. All subsequent output using ".", `p` and `nl` will be redirected to this file. The `endredir` word will end the redirection. Performing another `redir` while a redirect is active will have the same result as `endredir "filename" redir`. There is no notion of a stack of file redirection.

6.4.5 Special output

Several words exist for special output:

- `x` pops and prints a value as a hex integer.
- `rawp` pops and prints a value as a raw hex value. This will be the underlying value stored in the value union. It is occasionally useful for debugging.

Neither of these words print a trailing newline.

6.4.6 Formatted output

A version of the C `printf` formatted print exists, under the name `format`. This takes a list of values (see Section 10) and a format string:

```
[1,"hello"] "%20d: %s" format.
```

While it is not a full implementation, all but the more usual formatting options are supported.

7 Global variables

Global variables are defined in two ways. The "polite" way is to use the `global` keyword, which creates a new global of the name following it, initially holding the nil value `none`:

```
1|0> global foo
1|0> ?foo.
NONE
1|0> 5!foo
1|0> ?foo.
5
```

The other way to define globals is simply to access a variable whose name begins with a capital letter. If no global or local exists with that name, a global is created with the initial `none` value:

```
1|0> 5 !Foo
1|0> ?Foo.
5
```

Globals, unlike locals, do not require a `?` sigil to be read: if the name is used "bare", their value will be stacked just as if the name was preceded with `?`, unless they contain an anonymous function. In this case, the function is run (see Section 18 below). It is, however, good practice to use the sigil. Writing to a global is done using the familiar `!` sigil.

Finally, accessing a global using its name without a sigil will do nothing if it contains None, while accessing it with the `?` will stack the None:

```
1|0 > 10!A
1|0 > ?A.
10
1|0 > A.
10
1|0 > none!A
1|0 > A.
  from [dot] <stdin>:5/2
Error: stack underflow in stack 'main'
Last line input: A.
1|0 > ?A.
NONE
1|0 >
```

8 Constants

Constants are similar to globals, but with the following differences:

- they are defined and set using the `const` keyword — this will pop a value off the stack and set the new constant to that value;
- they can never be redefined or written to.

Here are some examples which might be found at the start of a maths package:

```
3.1415927 const pi
2.7182818 const e

180 pi/   const radsToDegsRatio
pi 180/   const degsToRadsRatio

:degs2rads
    :"(degs -- rads) convert degrees to radians"
    degsToRadsRatio*
;

:rads2degs
    :"(rads -- degs) convert radians to degrees"
    radsToDegsRatio*
;
```

9 Flow control

This section of the manual will describe how Angort handles conditional code and loops. There are also several idioms using anonymous functions which can be used for flow control, which will be described briefly.

9.1 Conditions

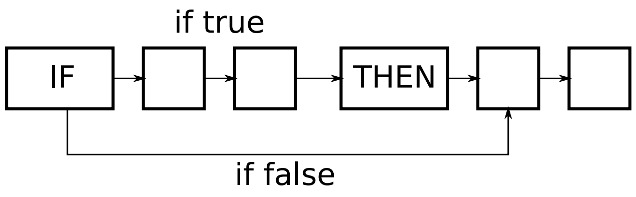
These have the form:

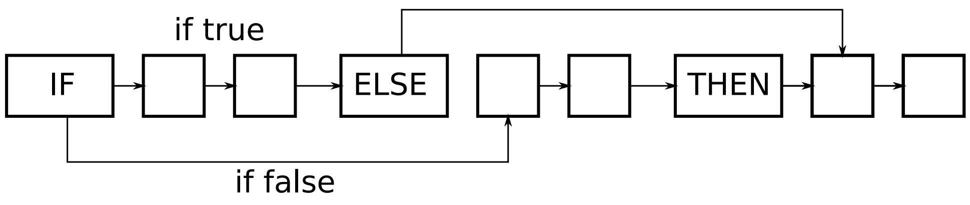
```
<condition> if <runs if true> then
```

The word `if` pops the item off the top of the stack, and jumps forward to just after the corresponding `then` if it is false (zero). There is also a two way branch:

```
<condition> if <runs if true> else <runs if false> then
```

This works the same way, except that the `if` jumps to after the `else` if the top value is false, and `else` jumps forward to just after `then`. This is shown in the figure below:





Here is a code example, printing whether a number is even or odd:

```
:evenodd
    # Get the number on the stack modulo 2, and use
    # whether it is zero or not as the condition.
    2 % if
```

```
        "number is odd"      # stack this string if there is a remainder
    else
        "number is even"     # stack this string if remainder is zero
    then                     # end the conditional
    .                        # print the top of the stack
;
```

We can run this with various values:

```
1|0> 4 evenodd
number is even
1|0> 3 evenodd
number is odd
```

Conditions can be nested.

9.2 Multi-line flow control and "slugs"

It is important to note that Angort processes input on a line-by-line basis when not defining a function, both within a script and in immediate mode. Therefore, code like the following will not work:

```
read len 0 = if
    "zero length string"
then
```

while the following will:

```
read len 0 = if "zero length string" then
```

With the former code, Angort will attempt to compile and run the first line, and will find that the line has no matching `then`. This will not occur on the latter example because the `then` is provided on the same line. This will not happen while a function is being defined, because Angort suspends compilation until the function is completed with a semicolon.

While this is a limitation when writing scripts it is easily surmounted by either putting code into a named function, or defining and immediately running an anonymous function. Anonymous functions are enclosed with parentheses, leaving the function on the stack. They can then be run with `call` or `@` (the two are equivalent). Thus the above can be written

```
(
    read len 0 = if
        "zero length string"
    then
)@
```

In informal conversations I have taken to referring to anonymous functions which are called immediately as "slugs," both from the typesetting terminology and from the visual appearance of `(....)@`.

9.3 Case blocks

In order to avoid testing if..else..then constructions where a number of different, mutually exclusive cases need to be tested, the "case block" structure is provided, which emulates the "elsif" construction in other languages.

Case blocks begin with `cases`, and each case consists of

```
    <condition> if <action> case
```

The case block must end, after the final case, with

```
    <default action> otherwise
```

An example:

```
:test |a:|
    cases
        ?a 0 = if "It's Zero". case
        ?a 1 = if "It's One". case
        ?a 2 = if "It's Two". case
        ?a 3 = if "It's Three". case
        ?a 4 = if "It's Four". case
        ?a 5 = if "It's Five". case
        ?a 6 = if "It's Six". case
        ?a 7 = if "It's Seven". case
        "It's something else".   otherwise

;
```

This function will print an appropriate message for integers from 0 to 7, and “It’s something else” for any other value.

9.4 Loops

There are two kinds of loops in Angort — infinite loops, which must be broken out of using `leave` or `ifleave`; and iterator loops, which loop over a collection (i.e. a hash or list) or a range. Both are delimited with curly brackets `{}`, but iterator loops use the `each` word before the opening brace.

9.4.1 Infinite loops

Any `leave` word will break out of an infinite loop:

```
( |:counter|          # a local variable "counter", no parameters
    0 !counter           # set counter to zero
    {                    # start loop
        ?counter.        # print the counter
        !+counter        # increment the counter
        ?counter 100=    # is it equal to 100?
        if leave then    # if so, leave the loop
    }
)@
```

This will count from 0 to 99. Note the use of an anonymous function as described in Section 9.2, which contains a local variable `counter`. The sequence

```
if leave then
```

is so common that it is coded to a special word of its own, and can be written as

```
ifleave
```

The function above could be written tersely as

```
(|:c| 0!c {?c. !+c ?c 100= ifleave)@;
```

This is still using a slug (see Sec. 9.2) because we require the local variable provided by the anonymous function. Loops can be nested, and the `leave` words will jump out of the innermost loop in which they appear.

9.4.2 Iterator loops

Iterator loops loop over an *iterable* value. These are currently ranges, hashes and lists. They are delimited by curly braces like infinite loops, but are preceded by the `each` word. This pops the iterable off the stack and makes an iterator out of it, over which the loop runs. The iterator exists for as long as the loop does, and is destroyed when the loop completes. Again, iterator loops can be nested (and can be nested with infinite loops). If we use an iterator loop to iterate over a range object, the counting example can be rewritten as:

```
(
    0 100 range          # stack a range from 0-99 (see below)
    each {               # start the iterator loop
        i                # push the current item in the iterator
        .                # print it
    }                    # end of loop
)@
```

Here, the `range` word takes two integers a, b from the stack which define an integer range $\{x \in \mathbb{Z} \mid a \leq x < b\}$. Note that range end value is exclusive — that is, it is the first value which is *not* in the range — so the above example, specified as `0 100 range`, actually counts from 0 to 99. This looks somewhat odd with integer ranges, but serves to prevent float ranges relying on equality tests:

```
(
    0 10 0.09 frange     # float range from 0-10 in steps of 0.09
    each {i.}            # print them
)@
```

will print about 9.990008 as its final value. Because this doesn't require any local variables, it works as an immediate mode one-liner:

```
0 10 0.09 frange {each i.}
```

Although we've not covered lists and hashes yet, it's useful to know that we can iterate over the values in a list. For example

```
[1,2,3,4,5] each {i.}
```

will print the numbers from 1 to 5. Similarly, we can iterate over the keys and values of a hash:

```
[% "foo" "bar", "baz" "qux"] each {i.}
```

will print the hash's keys: "foo" and "bar", while

```
[% "foo" "bar", "baz" "qux"] each {i p "=" p ival.}
```

will print "key=value" for each entry. Here, `ival` gets the value (as opposed to the key) of the current pair in the iterator, and `p` prints without a newline.

9.4.3 Nested iterator loops

Each loop has its own iterator, even if they come from the same iterable:

```
( |:r|
    0 10 range !r           # store a range
    ?r each {               # outer loop
        i.                  # print value of outer loop iterator
        ?r each {           # inner loop
            i.              # print value of inner loop iterator
        }
    }
)@
```

The range here is used twice, generating two separate iterators with their own current value.

While the word `i` retrieves the iterator value inside the current loop, inside nested iterator loops it is often useful to get the value in outer loops. This can be done with the `j` and `k` words, to get the outer and next outer iterator values:

```
( |:r|
    0 10 range !r           # store a range
    ?r each {               # outer loop
        ?r each {           # inner loop
            i j +           # print sum of inner and outer iterators
        }
    }
)@
```

The `jval` and `kval` words also exist to get a hash iterator value for two outer iterator loops.

9.5 The state of the stack inside a loop

Note that the iterator is not kept on the normal stack — it is popped off when the loop starts and kept inside a special stack. The same applies to infinite loops: no loop state is kept on the stack. This leaves the stack free for manipulation:

```
# sum all the integers from 0 to x inclusive
:sumto |x:|
    0               # stack an accumulator
    0 ?x 1+ range   # stack a range from 0 to x inclusive
    each {          # stack an iterator, then pop it onto the iterator stack
        i           # get current iterator value
        +           # add it to the accumulator
    }               # end loop
;                   # finish and return the accumulator
```

Note that this isn't actually how you'd do it: `reduce` is more efficient:

```
:sumto |x:| 0 0 ?x 1+ range (+) reduce;
```

This more functional style of programming is dealt with in Section 14.

9.6 Words to create ranges

There are three words to create a range on the stack:

name	stack picture	side-effects and notes
range	(x y – range)	create an integer range over the interval $[x, y)$ with a step of 1; i.e. a range from x to $y - 1$ inclusive.
srange	(x y s – range)	create an integer range over the interval $[x, y)$ with a step of s
frange	(x y s – range)	create an float range over the interval $[x, y)$ with a step of s

See Section 5.4 for how stack pictures define the parameters and return values for a function.

9.7 Explicit iterators

It's sometimes necessary to use iterators manually instead of creating them automatically with "each" and getting the values using "i" in an iterator loop. To do this, we have the following words:

name	stack picture	side-effects and notes
mkiter	(iterable – iterator)	create an iterator
icur	(iterable – item)	get the current item
inext	(iterable –)	advance the iterator
idone	(iterable – bool)	return 1 if the iterator is at the last item
ifirst	(iterable –)	reset the iterator to the start
iter	(– iterable)	return the iterator which is being iterated over in the current loop

You might need this when you want to iterate over two collections in parallel, for example in implementing the `zipWith` function[11]. This function takes two collections, and runs a binary function on pairs of items, each from one of the collections, returning a list:

```
["foo","bar"] ["fish","zing"] (+) zipWith each{i.}
```

would print

```
foofish
barzing
```

[11] `zipWith` is already built into Angort, this is just an illustrative example.

This could be implemented by using an index to go over both lists:

```
    :zipWith |a,b,f:|
        [] # stack an empty list
        0 ?a len ?b len min range each {
            i ?a get # get from first list
            i ?b get # get from second list
            ?f call # call the function
            , # and append to the list
        }
;
```

but a more elegant solution might be:

```
    :zipWith |a,b,f:it|
        ?b mkiter !it
        []
        ?a each {
            ?it idone ifleave # exit if explicit iterator done
            i # get loop iterator value
            ?it icur # get explicit iterator value
            ?f@, # call the function and append to the list
            ?it inext # advance the explicit iterator
        }
    ;
```

Here, we use an each loop to iterate over list a, and an explicit iterator to iterate over list b.

9.8 Conditional compilation

The normal **if..else..then** construction doesn't help if you want a piece of code to be completely ignored by Angort, for example when it refers to a library that may not be loaded. In that case, you don't want Angort to try to compile the code at all because it won't be able to resolve the identifiers to the unloaded library. Consider the code

```
0!Threads
(
    ?Threads if
        `thread library drop
        0 mythreadfunction thread$create !Thread
    then
)@
```

This will fail because the thread library won't load, but Angort still needs to compile the `thread$create` call.

To fix problems like this, the **compileif** directive can skip entire lines. Because it reads the value on the stack top it is compiled as an opcode, so don't put it inside a function and make sure it ends the line it's on. The skipping stops when the **endcompileif** directive is found on a line by itself. The above code could then be rewritten

```
0!Threads
?Threads compileif
    `thread library drop
    0 mythreadfunction thread$create !Thread
endcompileif
```

There is also the **elsecompileif** directive, which works as you would expect:

```
0!Threads
?Threads compileif
    `thread library drop
    thread$mutex !MyMutex
    0 mythreadfunction thread$create !Thread
    :lock ?MyMutex thread$lock
    :unlock ?MyMutex thread$unlock
elsecompileif
    :lock ;
    :unlock ;
endcompileif
```

Note that **compileif clauses cannot be nested.** Those caveats in full:

- **compileif** structures cannot be nested;
- **ifcompile** should end a line;
- **elsecompileif** and **endcompileif** should be on a line by themselves;
- **compileif**, **elsecompileif** and **endcompileif** cannot be inside a function definition (anonymous or otherwise).

10 Lists

Angort lists are actually array based, with the array resizing automatically once certain thresholds are passed (similar to a C++ vector or Java ArrayList). A list is created by enclosing Angort expressions in square brackets and separating them by commas:

```
[]               # the empty list
[1,2,3]          # a list of three integers
["foo","bar",1] # two strings and an integer
```

As noted above, lists can be iterated over. Lists can also contain lists, and can be stored in variables — more precisely, references to lists can be stored in variables:

```
1|0> [1,2] !A       # create a list and store it in global A
2|0> ?A!B           # copy A to B
2|0> 3 ?A push      # append an item to A
2|0> ?A each {i.}   # print A
1
2
3
1|0> ?B each {i.}   # print B - it also has the extra item!
1
2
3
```

Note that the list in B has also changed — it is the same list, just a different reference. The following are some words which act on lists, with their stack pictures:

name	**stack picture**	**side-effects and notes**
[	(– list)	creates a new list
,	(list item – list)	appends an item to the list
]	(list item – list)	appends an item to the list
get	(n list – item)	get the nth item from the list
set	(item n list –)	set the nth item in the list
remove	(n list – item)	remove and return the nth item
shift	(list – item)	remove and return the first item
unshift	(item list –)	prepend an item
pop	(list – item)	remove and return the last item
push	(item list –)	append an item
in	(item iter –)	return true if an item is in a list, integer range or hash keys

Note that the literal notation for lists — the square brackets and the comma — fall naturally out of the definition of the words[12]. The comma is a useful word, acting as a way to add items to a list on top of the stack without popping that list. This means we can write code to do a list copy:

```
:copylist |list:|
    []               # stack an empty list
    ?list each {     # iterate over the list passed in
        i ,          # for each item, add it to the list on the stack
    }
;                    # finish and return the list on the stack
```

This syntax allows us to create a form of list comprehension:

```
[] [1,2,3,4] each {i dup * dup 2 % if , else drop then}
```

will create the list $[1, 9]$: the squares of the numbers but only if they are odd. It does this by squaring the number, duplicating and testing to see if it is nonzero modulo 2, and if so appending it to the list, otherwise dropping it. In reality, we would probably use the `map` and `filter` words with anonymous functions:

```
[1,2,3,4] (dup *) map (2 %) filter
```

Exercise for the reader: what does the idiom

```
[swap]
```

do?

10.1 slice – getting parts of lists or strings

The `slice` word can be used to extract part of a list or string, returning a smaller list or string. The semantics are identical for both strings and lists.

At version 3, Angort is in a transition phase with two implementations of `slice`, each with entirely different semantics! This is due to the older code being both conceptually wrong and buggy, but still in use.

In the `future` namespace (see , `slice` has similar semantics to the Python array slice operator. In the `deprecated` namespace, a crude slice operator is supported which is briefly documented below (and has bugs with negative indices). The default implementation of `slice` will throw an exception – either `future` or `deprecated` must be imported, for example with

```
`future nspace [`slice] import
```

10.1.1 The `future` implementation

This will become the default implementation in Angort 4.x.x, when existing code has been corrected. This implementation uses similar semantics to Python:

- The stack picture is `(list/string start end - out)` where both `start` and `end` are zero-based indices. The slice includes the start and excludes the end, so

  ```
  "foo" 0 2 slice.
  ```

  will print

  ```
  fo
  ```

  A zero end index maps to the length of the sequence. This means that

  ```
  "fooble" 1 0 slice
  ```

[12]There is an exception: if the tokeniser finds the sequence `[]` it discards the second bracket, allowing us to notate the empty list in a natural way.

will print from character 1 to the end of the string:

```
ooble
```

- Negative indices (or a zero end index as mentioned above) count from the end, so -1 is the last element, -2 is the penultimate element etc. Thus,

```
"fooble" -4 -1 slice.
```

will print from the fourth character from the end to just before the last character:

```
obl
```

10.1.2 The `deprecated` implementation

This is the default implementation for version 2.x.x and lower. Here, the stack picture is `(list/string start len - out)`: the arguments are the zero-based start index and the number of elements in the slice. If the length is negative the slice is to the end of the string or list.

The start index may be also be negative, indicating distance from the end, but this is bugged for lists.

11 Symbols

Symbols are single-word strings which are stored in an optimised form for easy and quick comparison (they are turned into unique integers internally). They are specified by using a backtick (`) in front of the word. They're most useful as keys in hashes, covered in the next section. Examples are `` `foo ``, `` `bar ``. Symbols can be used most places where strings are used, but cannot be iterated or sliced.

12 Hashes

Hashes are like Python dictionaries: they allow data be stored using keys of any hashable type (see table in Sec. 3.5). Hashes are created using a similar syntax to the list initialiser, but with a `%` after the closing brace and data in key,value pairs[13]:

```
[%]
```

creates an empty hash, and

```
[%
    `foo "the foo string",
    `bar "the bar string"
]
```

creates a hash with two entries, both of which are keyed by symbols (although the keys in a hash can be of different types). We can add values to the hash using `set` which has the picture `(val key hash -)`:

```
[%]!H                           # create the empty hash
"the foo string" `foo ?H set    # add a string with the key `foo
"the bar string" `bar ?H set    # add a string with the key `bar
```

and read values using `get` which has the picture `(key hash - val)`:

```
2|0> `foo ?H get
2|1> .
the foo string
```

[13]This is a somewhat awkward syntax, but all the other bracket types were used elsewhere.

We can also iterate over the hash's keys and values:

```
:dumphash |h:|
    ?h each {                   # iterate over the hash's keys
        i p                     # print key without trailing new line
        ":   " p                # print colon and spaces
        ival                    # get the value of the key in the hash
        .                       # and print it
    }
;
```

If we run this on the hash H defined above, we get

```
2|0> ?H dumphash
foo:   the foo string
bar:   the bar string
```

We can nest iterator loops for hashes too, as we did with lists. Just as `i`, `j` and `k` get the inner, next outer and next outer loop keys (see Sec. 9.4.3), `ival`,`jval` and `kval` get the inner, next outer and next outer loop values with hashes.

12.1 Shortcut symbol get and set

There is "syntactic sugar" for retrieving the value of a key in a hash where the key is a literal symbol. The syntax is `?`key` , meaning "get the value for *key* in the hash on the stack," and so has the stack picture `(hash - value)`. Therefore, instead of using

```
`foo ?H get
```

we can use

```
?H?`foo
```

This has been added because this is by far the most common use-case. We also have the same ability to set a value in a hash with a literal symbol, using the syntax `!`key`. This has the stack picture `(value hash -)`. Thus we can do

```
96 ?H!`temperature
```

instead of

```
96 `temperature ?H set
```

12.2 Words for hashes

Hashes can also use many of the same words as lists:

name	stack picture	side-effects and notes
[%	(– hash)	creates a new hash
,	(hash key value – hash)	adds a value to the hash
]	(hash key value – hash)	adds a value to the hash
get	(key hash – value)	get a value from the hash, or `none` if it is not present
set	(value key hash –)	set a value in the hash
remove	(key hash – value)	remove and return a value by key
in	(key hash –)	return true if a key is in the hash

Note that the comma and close bracket words examine the stack to determine if they are working on a list or a hash.

12.3 Hash to string function

DEPRECATED – do not use. It requires passing far too many things down the call chain, is prone to bugs particularly in debugging, and I never use it. By default, printing a hash — or converting it to a string any other way — will just print the default string. This tells you it's a hash and gives its address in memory:

```
1|0 > [%] .
<TYPE hash:0x16a4840>
2|0 >
```

However, if we define a hash member called `toString` (where this key is a symbol) which is a function, then that function will be called to generate a string. This is useful in many cases where hashes are used as data structures[14].

13 Garbage collection

Garbage collection is done automatically — up to a point. Specifically, the system does reference-counted garbage collection continuously, which works well in most cases. However, it is possible to create cycles:

```
[] !A               # make a list called A
[?A] !B             # make a list called B, referencing A
?B ?A push          # add a reference to B in A
```

Now there are two objects referencing each other — a cycle. This can happen in lists, hashes and closures. Reference-counted garbage collection will never delete these. Therefore it may be necessary in programs with a complex structure to call the full garbage collector occasionally.

This is done periodically, by default every 100000 instructions or so. This interval can be changed by writing to the `autogc` property with a new interval:

```
1000 !autogc
```

It can also be disabled entirely by setting `autogc` to a negative value. A full garbage collect can be done manually by the word

```
gc
```

Incidentally, this is the same style of garbage collection used by Python.

14 Functional programming

Anonymous functions are defined with brackets, which will push an object representing that function (and any closure created) onto the stack. This can then be called with `call` (which can be abbreviated to "`@`") Such functions may have parameters and local variables. For example, this is a function to run a function over a range of numbers, printing the result:

```
:over1to10 |func:|
    1 10 range each { i ?func@ . } ;
```

With this defined, we can now use it to show the squares of those numbers:

```
(|x:| ?x dup *) over1to10
```

or more simply

```
(dup *) over1to10
```

[14]However, this can sometimes go horribly wrong, particularly where debugging is involved. Use with care.

14.1 Recursion

This doesn't really belong here; should it go into a putative "debugging and optimisation" section? Recursion is normally achieved by calling the function by name within its own definition:

```
:factorial |x:|
    ?x 1 = if 1 else ?x ?x 1 - factorial * then;
```

This is not possible in an anonymous function. To achieve recursion in anonymous functions, use the `recurse` keyword to make the recursive call:

```
1 10 range (|x:| ?x 1 = if 1 else ?x ?x 1 - recurse * then) map
"," intercalate.
```

will print the factorials of the first 10 natural numbers. See below for how `map` performs a function on each member of an iterable to produce a list, and for how `intercalate` builds a string out of a list by joining string representations of its members with a separator.

The keyword `self` is occasionally useful: rather than calling the containing function recursively, it stacks a reference to the function itself. Thus:

```
:foo inc self !LastFuncCalled;
4 foo.
4 ?LastFuncCalled@.
```

creates a function `foo` which, when called, increments the value on the stack and also stores a reference to `foo` in the global `LastFuncCalled`. We then call `foo`, and then call whatever is stored in `LastFuncCalled` – which will be `foo` again.

14.1.1 A warning

Angort has a limited return stack size of 256 frames, and because of the nature of the language there is no tail call optimisation. Recursive algorithms may therefore run out of stack. Also, Angort may not be suitable for expressing very complex recursive functions. Consider for example the quicksort algorithm: this can be implemented as

```
:qs |l:p|
    ?l len 1 <= if
        ?l
    else
        ?l pop !p
        ?l (?p <) filter qs
        [?p] +
        ?l (?p >=) filter qs +
    then;
```

but running it on a large number of items will be very slow. Try

```
[] 0 100000 each {rand 200000 %,} qs
```

This generates a list of 100000 integers in the range $[0, 199999]$ and sorts them. On my laptop it takes about 7 seconds – a long time for such a simple task. This is because the algorithm recurses deeply and widely, and each recursion constructs three lists (a new single-item list for the pivot and two lists using a filter) and pastes them together, constructing a temporary list on the way. This is very inefficient. In contrast, using the built in `sort` word takes only 0.25s:

```
[] 0 100000 each {rand 200000 %,} sort
```

This is still slow, because the internal comparison operator needs to perform typechecking on each pair of elements it compares so that integers and floats in the same list will be compared correctly. However, it is a big improvement because the `libc qsort` function is being used.

14.2 Words for dealing with functions

name	**stack picture**	**side-effects and notes**
map	(iter func – list)	apply a function to an iterable, giving a list
reduce	(start iter func – result)	set an internal value (the accumulator) to "start", then iterate, applying the function (which must take two arguments) to the accumulator and the iterator's value, setting the accumulator to this new value before moving on.
filter	(iter func – list)	filter an iterable with a boolean function
filter2	(iter func – falselist truelist)	filter an iterable with a boolean function, placing true elements and false elements in separate lists
zipWith	(iter iter func – list)	iterate over both iterables, combining the elements with a binary function and putting the results into a list
all	(iterable func – bool)	true if the function returns true for all members of the iterable.
any	(iterable func – bool)	true if the function returns true for any members of the iterable.

We can now list all the squares of a range of numbers:

```
0 100 range (dup *) map each {i.}
```

We can also write a function to sum the values in an iterable using `reduce`:

```
:sum |itr:|
    0           # the accumulator value starts at zero
    ?itr        # this is the iterable
    (+)         # and this is the function
    reduce      # repeatedly add each item to the accumulator,
                # setting the accumulator to the result. When
                # finished, return the accumulator.
;
```

14.3 Closures

Anonymous functions can refer to variables in their enclosing function or function, in which case a closure is created to store the value when the enclosing function exits. This closure is mutable - its value can be changed by the anonymous function. For example, consider the following function:

```
:mkcounter |:x|     # declare a local variable x
    0!x             # set it to zero
    (               # create a function
        ?x dup .    # which prints the local
        1+ !x       # and increments it
    )
;
```

This returns an anonymous function which refers to the local variable inside the function which created it. We can run this function and store the returned function in a global:

```
mkcounter !F    # run it and store the returned function+closure
```

If we now run

```
    ?F call
```

a few times, we will see an incrementing count - the value in the closure persists and is being incremented. We can call mkcounter several times and each time we will get a new closure.

14.3.1 Closures are by reference

However, all closures are *by reference* — the child functions get references to variables closed in the parent function, so a function which returns several functions will all share the same closure, and any changes to variables in the closure will be reflected in all the other functions. For example:

```
:mklistoffunctions |:x|
    []
    0 10 range each {
        i !x (?x),
    }
;
```

looks like it should produce a list of functions, each of which returns the numbers from 0 to 9. However, all the functions will return 9 because they all share the same copy of `x`. We can get around this by using a *closure factory* function to hold private copies:

```
:factory |x:| (?x);

:mklistoffunctions |:x|
    []
    0 10 range each {
        i factory,
    }
;
```

14.3.2 Iterators are not stored in closures

Notice that in the `mklistoffunctions` example above we did not write

```
:mklistoffunctions
    []
    0 10 range each {
        (i),
    }
;
```

and instead used a local variable *x* to store the iterator value. This is because the values of loop iterators such as `i` are not true variables, and thus are not stored in the closure. Iterators must be stored in local variables if they are to be used in an anonymous function created in their loop's context.

15 Exceptions

Exception handling is done with a `try/catch/endtry` construction:

```
try
    ...
catch:symbol1,symbol2...
    ...
endtry
```

Upon entry to the catch block, the stack will hold the exception symbol on the top with extra data (typically a string with more information) under that. There are quite a few built-in exceptions: they are listed in `exceptsymbs.h`. To throw your own exception, use
`throw (data except -)`, e.g.

```
"Can't open file: " ?fn + `badfile throw
```

To catch all exceptions, use the `catchall` word:

```
try
    ...
catch:symbol1,symbol2...
    ...
catchall
    ...
endtry
```

16 Getting help

There are many other functions and operations available in Angort. These can be listed with the `list` word, and help can be obtained on all words with `??` (except the very low-level words compiled directly to bytecode, which are all covered above):

```
1|0 > ??filter
filter: (iter func -- list) filter an iterable with a boolean function
1|0 >
```

If you have loaded a library or package and not imported the functions into the main namespace (see section **??**below), you can use the fully qualified name to get help:

```
1|0 > `io library drop
1|0 > ??io$open
io$open: (path mode -- fileobj) open a file, modes same as fopen()
1|0 >
```

17 Namespaces, libraries and packages

All Angort identifiers (apart from the built-in tokens listed in Table **??**) are stored in a *namespace*. An identifier in a program can be either *fully qualified*, in which case the namespace for the identifier is explicitly given before a dollar sign, such as `std$quit`; or *unqualified*, in which case Angort will scan only the imported namespaces to find it. Angort starts with a set of default namespaces, which are imported by default. These include

- `coll` for collection functions,
- `string` for string functions,
- `math` for mathematical functions,
- `env` for system environment functions (command line arguments, environment variables etc.),
- `coll` for collection handling functions,
- `user` for names defined by the user.

Others may be added. Since these namespaces are imported, the user does not need to enter the fully qualified name. As noted above in Sec. 17.6, the `future` and `deprecated` namespaces are not imported. A namespace can be imported by using a command of the form

```
`mynamespace nspace import
```

or to only import some names

```
`mynamespace [`name1,`name2 ..] import
```

See below for more ways to manipulate namespaces.

17.1 Packages

It is possible to define a new namespace using the `package` directive, typically done inside a separate file. Build a package by putting the `package` directive at the start of the file along with the package name, e.g.

```
package wibble
```

and include that file with `require` instead of the usual `include`.

This will cause a new namespace to be created where `package` is invoked, and all subsequent definitions until the end of the file will be put into that namespace. On return from the file, `require` will put a namespace identifer (or NSID) on the stack. All the globals, constants and functions defined in the package are available by prefixing the name with the package name and a dollar sign:

```
package$name
```

Note that the package name is that given to the `package` directive, not the name of the file! Angort returns to defining things in the `user` namespace at the end of the file (but see `endpackage` below in Sec. 17.5 if we want to go back to the `user` namespace before then).

17.2 Importing packages into the default space

With the NSID returned by `require` n the stack, we can import the namespace — either all of it or part of it. The `import` word takes two forms:

```
require "wibble.ang" import
```

will import all the public names defined in the package, while

```
require "wibble.ang" [`foo, `bar] import
```

will only import the given names — in this case, `foo` and `bar`. If we do not wish to import the package at all we can do

```
require "foo.ang" drop
```

to discard the NSID. The namespace identifier can also be used in other ways:

name	stack picture	side-effects and notes
require "filename"	(– nsid)	load a package
library	(libname – nsid)	load a native plugin library (see Sec. 19)
nspace	(name – nsid)	look up a loaded package or library and return the NSID
package packagename		directive to start a new namespace
private		all subsequent names are not exportable from the namespace
public		all subsequent names are exportable from the namespace (default)
import	(nsid –)	import all public definitions from a namespace into the default namespace
import	(nsid symbollist –)	import some public definitions from a namespace into the default namespace
names	(nsid – symbollist)	get a list of names in a namespace
ispriv	(nsid name – bool)	return true if a name is private in the namespace
isconst	(nsid name – bool)	return true if a name is constant in the namespace

17.3 Loading and importing libraries

C++ plugin libraries – which are dealt with more fully in Sec. 19 – are imported in a similar way using the `library` function: `` `io library import `` will load the IO library and import all its functions. Note the difference, however: because `require` is a directive dealt with by the compiler, the name of the package follows the keyword. The word `library` is an actual function, which requires a string on the stack, so the library name (string or symbol) precedes `library`. This loads the library and returns the NSID for importing (or dropping).

17.4 Overriding functions in default namespaces

Although they are defined as constants, it possible to override the definitions of functions defined in default imported namespaces like `std$p` and `std$quit`. For example the standard function `p` can be overridden by forcing a new global in the `user` namespace:

```
global p
:p "wibble" .;
```

Once this has been done, we now have two `p` functions: `std$p`, which is the standard definition; and `user$p`, which is our new function. Because `user` is the "current" namespace, i.e. that into which new names are defined, this will be searched first on compilation, so unqualified `p` will be resolved as `user$p`, our new function. It is still possible to use `std$p` by using the fully qualified name. We could also define a user function with the same name as a standard function, which uses that standard function, by using the fully qualified name:

```
global quit
:quit "Angort is quitting now!". std$quit;
```

17.5 Local packages

It is sometimes necessary to create a package inside a script which is not included from another script. One example is where the script does not terminate, leaving the user at the prompt with some functions defined, but some functions should be private.

To do this, define the package thus:

```
package somename
private
...private functions...
public
...public functions...
endpackage import
```

The `endpackage` word does the same as the return from a `require` normally does — close off the package and stack the package ID.

17.6 The "deprecated" and "future" namespaces

Angort is a changing language, and to permit this some functions are occasionally moved into the "deprecated" namespace when they are obsolete. Other functions are added to the "future" namespace when they are new and break back-compatibility. While both these namespaces are loaded, none of their symbols are imported. Typically, a function might have an old version in the deprecated space and a new version in the future space, while the default namespace contains a placeholder which gives details of the differences between the versions and throws an exception. it is up to the user to explicitly import either the deprecated or future version.

To do this, we need to import symbols from a namespace which is already loaded, which we can do using the `nspace` function; and then use `import`. For example, if we wish to use `foo` and `bar` from the future namespace but `fish` from the deprecated namespace, we could use

```
`future nspace [`foo,`bar] import
`deprecated nspace [`fish] import
```

18 The true nature of function definitions

Words are actually global variables bound to anonymous functions. Given that anonymous functions are written as blocks of Angort in brackets (see below), then

```
:square |x:| ?x ?x *;
```

could also be written as

```
global square
(|x:| ?x ?x *) !square
```

with exactly the same end result. Referring to a global by using the `?` sigil will simply stack its value, whereas referring to it without the sigil will check if it is holds a code block or closure and run it if so, otherwise stack the value. This is useful in functional programming.

18.1 Sealed functions

Combined with constants, this allows for "sealed" function definitions. In general all functions can be redefined, but a function defined thus:

```
(|x:| ?x ?x *) const square
```

cannot be. This can be useful in package development.

This is also useful when you want to import only certain names from a plugin library (Section 19). Consider the `vector2d` library, which contains the functions `vector2d$x` and `vector2d$y`. These are cumbersome, so we could import just these in their short forms as

```
?vector2d$x const x
?vector2d$y const y
```

Using the `?` ensures that we get the function itself, rather than trying to run it[15].

18.2 Forward declarations (deferred declarations)

It's also possible to use this mechanism to define global variables, use them as function names, and change their values later. This lets us defer function definitions:

```
global foo      # define a global called ``foo'' (null-valued)
:bar foo;       # a function which uses it

:foo...;        # actually define foo
```

19 Plugin libraries (currently Linux only)

These are shared libraries which can be loaded into Angort. They communicate with Angort via a simplified interface, and are easy to write in C++ (see the plugins directory for some examples). Once created, they should be named with the `.angso` extension and placed in Angort's library search path (a colon separated string). By default, this is

```
.:~/.angort:/usr/local/share/angort
```

but it can be changed using the `searchpath` property (note that ~ will be expanded to the user's home directory). For example, to append `/home/foo/bin` to the path, you could write

```
?searchpath ":/home/foo/bin" + !searchpath
```

Plugin libraries are loaded using the `library` word, which leaves a namespace ID (an *nsid*) on the stack so that `import` can be used, or `drop` to not import anything. The namespace is the library name, which may not be that of the library file — it's defined in the plugin code. For example, to load the standard file IO library and import it, we would write

```
`io library import
```

Again,**Note that** unlike package loading with `require`, the library name comes first. This is because `library` is an instruction which is compiled and then run, rather than a compiler directive which is acted on at compile time. This means we can do things like

```
[`id3, `mpc, `io] each {i library import}
```

to import lists of libraries.

[15] In reality we would probably use the optional list parameter to `import`, as described in Section 17.1, although this would leave the functions open for redefinition.

20 Useful paradigms

Angort is a fairly novel language, being an update of stack-based languages with collections and functional elements. As such, new paradigms for performing familiar tasks must be found, and often features of the language allow these tasks to be performed in an efficient manner but need to be "discovered" by actually using the language. In this section I hope to show some examples of a few useful paradigms I have discovered during development.

20.1 Use of the comma operator for list building

A trivial example is the list comprehension, a feature whereby a new list can be constructed by performing some operation on elements of another. In Python, for example, one can obtain the squares of a range of elements using a comprehension thus:

```
print [x**2 for x in range(20)]
```

Naturally one can use `map` for this in Angort:

```
0 20 range (dup *) map "," intercalate.
```

but I found I had this facility even before the `map` function had been written, simply by using the comma outside the normal list context:

```
[] 0 20 range each {i dup *,} "," intercalate.
```

Breaking this down:

```
[]          # stack an empty list
0 20 range  # stack a range from 0-19 inclusive
each {      # pop and iterate over that range. Empty list is now stack top.
 i          # push the next element
 dup *      # square it
 ,          # append it to the list
}           # end loop

"," intercalate.      # print the resulting list
```

In a list context, the stack picture for the comma word is `(list item - list)`: it appends the item to the list and leaves the list on the stack. While this permits the intuitive `[1,2,3,4]` syntax for list "literals", it also provides a powerful list-building facility. In a hash context, the stack picture is `(hash value key - hash)`[16]. This provides a way to generate hashes:

```
[%] 0 20 range each {i i dup*,}
```

will generate a hash of the squares of the first 20 non-negative integers, keyed by the integer.

20.2 Objects with methods and private members using hashes and closures

We can emulate objects in Angort by creating objects as hashes, with methods as functions closing over the local variables in the creating function:

```
# create a rectangle object as a hash, with delegates to draw it
:mkrectangle |x,y,w,h:|
    [%
        # a member to draw the rectangle - this
        # will create a closure over x,y,w,h.
        # We assume there is a graphics package with
        # a drawrect function; this is not a part of standard
        # Angort (although see the SDL plugin in angortplugins).
```

[16]Which context is used depends on the type of second item on the stack: if it is a list, the list context is used; otherwise the type of the third item is checked and the hash context is used if it is a hash. If not, an exception is thrown.

```
        `draw (?x ?y ?w ?h graphics$drawrect),

        # another to move it by some amount

        `move (|dx,dy:|
            ?x ?dx + !x
            ?y ?dy + !y
        )
    ]
;

# create it
100 100 10 10 mkrectangle !R
# draw it
?R?`draw@
# move it and redraw
20 20 ?R?`move@
?R?`draw@
```

Values such as `?R?`move` are actually closer to C# "delegates", in that they contain references to both the method to perform and the object which should perform it (i.e. the closure). This also provides a "private member" mechanism: if a value is defined in a closure in the function which creates the hash, rather than in the hash itself, that value will only be visible to functions defined in the hash.

Public member variables can be implemented by storing the hash itself as a local variable called (say) `this`, and accessing it inside the methods. Using this technique, the above could be written as

```
# create a rectangle object as a hash, with delegates to draw it
:mkrectangle |x,y,w,h:this|
    [% dup !this # store the hash itself in the closure
        # set the member values
        `x ?x,
        `y ?y,
        `w ?w,
        `h ?h,

        `draw (?this?`x ?this?`y
            ?this?`w ?this?`h graphics$drawrect),

        `move (|dx,dy:|
            ?this?`x ?dx + !this?`x
            ?this?`y ?dy + !this?`y
        )
    ]
;
```

Look at how to do inheritance, particularly using members in the superclass.

20.3 Local static variables

It is often useful to create "static" variables local to functions – local variables which keep their value outside the function's scope. In C, we could write a simple counting function like this:

```
int counter(){
    static int ct = 0;
    return ct++;
}
```

In Angort, we can easily do the same thing with a "factory" function to create a closure:

```
:mkcounter |:ct|
    0!ct
    (?ct !+ct);
mkcounter const counter
```

This is useful if we want to create lots of counters, which would be hard to do from a single counter in C.

```
mkcounter const counter1
mkcounter const counter2
```

However, if we only need one, creating `mkcounter` seems wasteful. We can do without by using an anonymous factory:

```
(|:ct|
    0!ct
    (?ct !+ct)) @ const counter
```

20.4 Control languages

- private/public sections
- word definition
- prompt changing

21 Optimisation and debugging

Angort does not have an optimising compiler: generally, tokens are converted directly into instructions. It is theoretically possible to write an optimiser but this has not yet been done. However, some tricks are available to assist in writing fast code:

- Use constant expressions rather than "folding" constants by hand (see below).
- Use the stack (or locals) to store common subexpressions.
- Use debug trace mode, print messages or breakpoints to determine which instructions are run most often.
- Above all, write the code which must run the fastest as native C++.

While this last point may seem to defeat the idea of a new language, it should be remembered that Angort is not intended as a high-performance language like C++.

21.1 Constant expressions

Constant expressions allow the compiler to compile a section of code, run it, and insert an instruction which will stack the value it produces. They are delimited by angle brackets, << and >>. They are useful to "fold" long constant expressions into a single value, which could be a list, hash or closure as well as a scalar or string. For example, we could write the following code:

```
:foo
    0 1000 range each {
        ["a","b","c","d","e"] each {
          i j dosomething
        }
    };
```

The problem here is that the inner list of letters is rebuilt each time we go around the loop, with the cost of creating the list and stacking and appending each item to it. If we "disassemble" this function into internal Angort instructions (which can be done with `"foo" disasm`, see below) we see this in the generated code:

```
    0x1819ec0 [foo.ang:4] : 0000 : litint (2) (0)
    0x1819ec0 [foo.ang:4] : 0001 : litint (2) (1000)
    0x1819ec0 [foo.ang:4] : 0002 : func (4)  (std$range)
    0x1819ec0 [foo.ang:4] : 0003 : iterstart (44)
    0x1819ec0 [foo.ang:4] : 0004 : iterlvifdone (43) (offset 19)
    0x1819ec0 [foo.ang:5] : 0005 : newlist (47)
    0x1819ec0 [foo.ang:5] : 0006 : litstring (18) (a)
    0x1819ec0 [foo.ang:5] : 0007 : appendlist (48)
    0x1819ec0 [foo.ang:5] : 0008 : litstring (18) (b)
    0x1819ec0 [foo.ang:5] : 0009 : appendlist (48)
    0x1819ec0 [foo.ang:5] : 0010 : litstring (18) (c)
    0x1819ec0 [foo.ang:5] : 0011 : appendlist (48)
    0x1819ec0 [foo.ang:5] : 0012 : litstring (18) (d)
    0x1819ec0 [foo.ang:5] : 0013 : appendlist (48)
    0x1819ec0 [foo.ang:5] : 0014 : litstring (18) (e)
    0x1819ec0 [foo.ang:5] : 0015 : appendlist (48)
    0x1819ec0 [foo.ang:5] : 0016 : iterstart (44)
    0x1819ec0 [foo.ang:5] : 0017 : iterlvifdone (43) (offset 5)
    0x1819ec0 [foo.ang:6] : 0018 : func (4)  (std$i)
    0x1819ec0 [foo.ang:6] : 0019 : func (4)  (std$j)
    0x1819ec0 [foo.ang:6] : 0020 : globaldo (5) (user$something)
    0x1819ec0 [foo.ang:7] : 0021 : jump (15) (offset -4)
    0x1819ec0 [foo.ang:8] : 0022 : jump (15) (offset -18)
    0x1819ec0 [foo.ang:9] : 0023 : end (1)
```

Note the `newlist` followed by `litstring/appendlist` pairs. These take time, and this gets much worse with a longer list. It is possible to deal with this by generating the list and putting it in a variable:

```
:foo |:lst|
    ["a","b","c","d","e"] !lst
    0 1000 range each {
        ?lst each {
          i j dosomething
        }
    };
```

which generates the code:

```
    0x1da2ec0 [foo.ang:4] : 0000 : newlist (47)
    0x1da2ec0 [foo.ang:4] : 0001 : litstring (18) (a)
    0x1da2ec0 [foo.ang:4] : 0002 : appendlist (48)
    0x1da2ec0 [foo.ang:4] : 0003 : litstring (18) (b)
    0x1da2ec0 [foo.ang:4] : 0004 : appendlist (48)
    0x1da2ec0 [foo.ang:4] : 0005 : litstring (18) (c)
    0x1da2ec0 [foo.ang:4] : 0006 : appendlist (48)
    0x1da2ec0 [foo.ang:4] : 0007 : litstring (18) (d)
    0x1da2ec0 [foo.ang:4] : 0008 : appendlist (48)
    0x1da2ec0 [foo.ang:4] : 0009 : litstring (18) (e)
    0x1da2ec0 [foo.ang:4] : 0010 : appendlist (48)
    0x1da2ec0 [foo.ang:4] : 0011 : localset (8) (idx 0)
    0x1da2ec0 [foo.ang:5] : 0012 : litint (2)
    0x1da2ec0 [foo.ang:5] : 0013 : litint (2)
    0x1da2ec0 [foo.ang:5] : 0014 : func (4)  (std$range)
    0x1da2ec0 [foo.ang:5] : 0015 : iterstart (44)
    0x1da2ec0 [foo.ang:5] : 0016 : iterlvifdone (43) (offset 9)
```

```
      0x1da2ec0 [foo.ang:6] : 0017 : localget (7) (idx 0)
      0x1da2ec0 [foo.ang:6] : 0018 : iterstart (44)
      0x1da2ec0 [foo.ang:6] : 0019 : iterlvifdone (43) (offset 5)
      0x1da2ec0 [foo.ang:7] : 0020 : func (4)  (std$i)
      0x1da2ec0 [foo.ang:7] : 0021 : func (4)  (std$j)
      0x1da2ec0 [foo.ang:7] : 0022 : globaldo (5) (idx dosomething)
      0x1da2ec0 [foo.ang:8] : 0023 : jump (15) (offset -4)
      0x1da2ec0 [foo.ang:9] : 0024 : jump (15) (offset -8)
      0x1da2ec0 [foo.ang:10] : 0025 : end (1)
```

This is much better: the list is generated and stored before the loop starts at offset 0015, so it is only generated once per function call. However, we can improve on this for both legibility and speed using a constant expression:

```
:foo
    0 1000 range each {
        <<["a","b","c","d","e"]>> each {
          i j dosomething
        }
    }
;
```

This is the same as the original code, but with angle brackets around the code which makes the list. Here's the disassembly:

```
      0x13a1410 [foo.ang:4] : 0000 : litint (2)
      0x13a1410 [foo.ang:4] : 0001 : litint (2)
      0x13a1410 [foo.ang:4] : 0002 : func (4)  (std$range)
      0x13a1410 [foo.ang:4] : 0003 : iterstart (44)
      0x13a1410 [foo.ang:4] : 0004 : iterlvifdone (43) (offset 9)
      0x13a1410 [foo.ang:5] : 0005 : constexpr (61)
      0x13a1410 [foo.ang:5] : 0006 : iterstart (44)
      0x13a1410 [foo.ang:5] : 0007 : iterlvifdone (43) (offset 5)
      0x13a1410 [foo.ang:6] : 0008 : func (4)  (std$i)
      0x13a1410 [foo.ang:6] : 0009 : func (4)  (std$j)
      0x13a1410 [foo.ang:6] : 0010 : globaldo (5) (idx dosomething)
      0x13a1410 [foo.ang:7] : 0011 : jump (15) (offset -4)
      0x13a1410 [foo.ang:8] : 0012 : jump (15) (offset -8)
      0x13a1410 [foo.ang:9] : 0013 : end (1)
```

Here, we are no longer generating the list at all – it has already been generated *at compile time.* Instead, we compile a single `constexpr` instruction to push the list onto the stack. Here's what happens when we encounter a << during compilation:

1. Store the current compile context and begin a new one, as if we were compiling a new function.
2. Compile code until we reach a >> sequence.
3. Run the code we just compiled.
4. Pop a value from the top of the stack.
5. Restore the original compile context.
6. Compile a `constexpr` instruction containing the value we just popped off the stack.

The most important thing to realise about constant expressions is that they break lexical scope: they cannot access any of the variables of the functions in which they are embedded. This is because they are not actually part of those functions: they are a completely separate piece of code run at compile time, long before any containing function ever runs. Thus the following code is invalid:

```
:foo |:a|
    2!a
    << 2 ?a * >> # invalid
;
```

because the local variable a is not in the scope of the constant expression. Although the function contains the expression lexically, it is not a true part of the function. Globals (provided they are created before the constant expression) are fine, because Angort compiles and runs each line at a time (outside functions).

```
2 !A
:foo
    <<3 ?A *>>. # valid - A exists when this compiles
;
```

21.2 Using constant expressions in closures

We often write functions to create other functions. For example, consider an exponential smoothing function of the form

$$y_t = \beta x_t + (1 - \beta) y_{t-1}$$

We can write a function to generate a smoother:

```
:mksmoother |beta:y|
0!y
    (|x:| ?beta ?x * 1.0 ?beta - ?y * + dup !y)
;
```

which we can then use:

```
0.1 mksmoother !F
{ read tofloat ?F@ . }
```

but this requires us to create a variable for each smoother, which could become awkward if we need to create a large number of them and would separate their creation from where they are used. Let's imagine we are dealing with many streams of interleaved input, each of which should be smoothed separately and perhaps with a different constant:

```
0.1 mksmoother !F1
0.12 mksmoother !F2
0.3 mksmoother !F3
0.1 mksmoother !F4
(
    {
        read tofloat ?F1@.
        read tofloat ?F2@.
        read tofloat ?F3@.
        read tofloat ?F4@.
    }
)@
```

Instead, we could create the smoothers using constant expressions:

```
(
    {
        read tofloat <<0.1 mksmoother>>@.
        read tofloat <<0.12 mksmoother>>@.
        read tofloat <<0.3 mksmoother>>@.
        read tofloat <<0.1 mksmoother>>@.
    }
)@
```

which is much neater. Constant expressions have many creative uses!

21.3 Tracing and disassembling

tracing

21.4 The debugger

debugger

22 Writing Angort plugin libraries

Angort plugin libraries add C++ functions to Angort. They can also add properties (see Sec. **??**) and binary operator overrides.

Plugins are written in C++, but for convenience this C++ is passed through a Perl preprocessor called `makeWords.pl`. This automatically adds code for popping arguments, adding the functions to a special array, and initialising the plugin in a special entry point. Let's look at a simple "hello world" example.

```
#include <angort/angort.h>
#include <string>
#include <sstream>

// set the angort namespace.

using namespace angort;

// declare the name of the library and set a namespace for it -
// DO NOT use C++ namespace stuff after this.

%name hello

// say that we are making a shared library.

%shared

// "hello" takes a single argument, a string, which will automatically be
// converted into const char *p0. If the argument spec ("s") were longer,
    subsequent
// arguments would be converted into p1,p2... of the appropriate type.
// Type letters are
// n - float
// d - double
// l - long int
// i - int
// c - codeblock/closure
// C - codeblock/closure/none
// v - any value
// y - string/symbol/none
// l - list
// h - hash
// A-Z - user-defined type (see example_complex/complex.cpp)

%wordargs hello s (name -- hello world string)
{
    std::stringstream s;
    s << "Hello " << p0 << ", how are you?";
    // "a" is the angort object, which is automatically passed in. The
        pushString()
    // method does exactly as its name implies, taking a const char *.
    a->pushString(s.str().c_str());
}

// initialisation function

%init
{
   fprintf(stderr,"Initialising HELLO plugin, %s %s\n",__DATE__,__TIME__);
}
```

```
// optional shutdown function

%shutdown
{
   fprintf(stderr,"Closing HELLO plugin, %s %s\n",__DATE__,__TIME__);
}
```

This code will create a plugin with a `hello` function which takes a name and generates a string "Hello (name), how are you?"

22.1 Angort internals: values and types

All but the most simple plugins require some knowledge of how Angort stores and types its values. Every value in Angort is a `Value` object. This is a class containing a union, which holds the actual value; and a `Type` object pointer, defining its behaviour. Each Angort type is defined by a singleton instance of a subclass of `Type` – all the internal types are global variables within the `Types` namespace. For example, strings are defined by the `StringType` object, which has `Types::tString` pointing to its only instance.

In the case of simple types like integers, the type object defines accessors to simple members of the `Value` class' embedded union `d`. Integers, for example, store their data in `d.i` while floats use `d.f`. Garbage collected types all use the `d.gc` pointer, which points to a `GarbageCollected` object. Thus, lists and hash value classes – and many user classes – inherit `GarbageCollected`.

All the internal types are declared in `include/types/` and defined in `lib/types/` should you wish to study them. This is all quite complex at first, particularly the behaviour of garbage collected objects. It can, however, be boiled down to a few principles:

- `Values` are initialised to `none`; their union's value is irrelevant and their type field points to `Types::tNone`.
- Calling `clr()` on a `Value` will decrement its reference count and set the type field to `Types::tNone`.
- Each type object contains a method to set a value to a given type and initialise it to a value. For example

  ```
  Value v;
  Types::tInteger->set(&v,1);
  ```

  will set the value `v` to the integer 1. Some of these may return a new object: for example, to set a value to a list we would call

  ```
  Value v;
  ArrayList<Value> *list = Types::tList->set(&v);
  ```

  which will create a new garbage collected list, set the value's union to hold its pointer, set the value's type field to `Types::tList`, and return the new list object.
- Each type object will also contain a `get()` method to return the underlying C++ value of a `Value`, performing conversion if possible. For example,

  ```
  int i = Types::tInteger->get(&v);
  ```

  will try to get the integer value from a `Value`.
- Plugin functions will receive a pointer to a `Runtime` object containing Angort's context for running code, which can be used to manipulate values on the stack.

- `Value *Runtime::pushval()` will push a new value onto the Angort stack and return it. It does this by just returning a pointer to the stack top and incrementing that pointer. It will *not* call `clr()` on the value.
- `Value *Runtime::popval()` will pop the stack, returning a pointer to the previous stack top.
- There are helper `Runtime::push..()` and `Runtime::pop..()` methods for most basic types.
- There are `Value::to..()` helper methods for most types to convert and return values. For example, `toFloat()` will attempt to convert to, and return, a C++ float.
- Strings are a special case – popping a string returns a `StringBuffer` object, whose underlying buffer can be fetched with `get()`. This is a UTF-8 string, and `StringBuffer` has methods for converting to wide characters. A common idiom fetches the string buffer as a const reference and calls `get()` to return the underlying buffer:

```
const StringBuffer& s = v->toString();
...
printf("%s\n",s.get());
```

There is a small hierarchy of classes which inherit the `Type` class: `IntegerType` (with the singleton pointer `Types::tInteger` is just a subclass of `Type`, while `ListType` and `HashType` inherit `GCType`, which handles values which are garbage-collected.

Summing up the situation for `Hash` as an example,

- `HashObject` inherits `GarbageCollected` and wraps a raw `Hash`: a map from `Value*` to `Value*`.
- `Types::tHash` points to a singleton instance of `HashType`, which inherits `GCType`, which inherits `Type`.
- This singleton has methods which can create a and set a `Value` to a new `HashObject`, and retrieve a `Hash` from a value (if it is of the correct type, throwing an exception otherwise). The `set()` and `get()` methods deal entirely with the underlying `Hash`, wrapping and unwrapping in `HashObject`, which exists solely to define an interface for garbage collection and iteration.

While this is all very complex, there are plenty of examples of words using these concepts in the Angort source – particularly in `stdWords.cpp`, `stdString.cpp` and `stdColl.cpp`.

22.2 Building plugins

To build the plugin, it needs to be passed through `makeWords.pl` (which can be found in the Angort source distribution) and compiled and linked as a shared library. The steps for the example are:

```
perl makeWords.pl hello.cpp >hello.plugin.cpp
g++ -fPIC -c -Wall hello.plugin.cpp
g++ -shared -Wl,-soname,hello.angso hello.plugin.o
```

The result is `hello.angso` where `.angso` is the extension required for a plugin.

22.3 Special % directives

The first thing to note is that some lines start with special directives beginning with a percent sign. These are handled by `makeWords.pl` and generate extra code inside the `.plugin.cpp` file.

22.3.1 The %name directive

This gives the name of the plugin's namespace. This should be unique to each plugin. Functions in the plugin will have the fully qualified name `namespace$function`, so our example will generate `hello$hello`. This generates a namespace block wrapping the rest of the output, so we should be careful how we use the C++ `namespace` directive afterwards.

22.3.2 The %shared directive

This says we are making a shared library, and should always be included. It exists because `makeWords.pl` is also used to build the functions included in the standard Angort libraries. It adds the special entry point required by dynamically loaded plugins to register them with Angort.

22.3.3 The %init and %shutdown directives

These are used to define functions which the plugin calls when it is loaded and unloaded (the latter only occurs when Angort is shut down).

22.3.4 The %wordargs and %word directives

These is the most important directives: they define the actual functions in the plugin. The `makeWords.pl` script generates a name for the function and a definition, and adds the function to the function list for the library.

The `%word` directive is used to define a function with no arguments, or whose arguments you wish to pop off the stack by hand. It has the form:

```
%word wordname (stack picture) description
more description..
more description..
{
    c++ code
}
```

If we add a function which has no arguments to our example and run it through `makeWords.pl`, we can see what it does. Our function is:

```
%word hello2 (-- string) a test
{
    a->pushString("hello world");
}
```

and the resulting code in `hello.plugin.cpp` is:

```
static void _word__hello2(angort::Runtime *a){
    a->pushString("hello world");
}
```

All plugin functions become functions which take a pointer to the `Angort` object, which is always parameter `a`. Naturally this means you must not call any other variable `a` in your plugin. The above code simple pushes a string onto the stack.

The `%wordargs` directive is used if the function has arguments, and pops them automatically, converting the types, into local variables called `p0`, `p1` and so on. Consider a definition

```
%wordargs distance dddd (x1 y1 x2 y2 -- dist)
{
    float dx = (p0-p2);
    float dy = (p1-p3);
    float dist = sqrt(dx*dx+dy*dy);
    a->pushDouble(dist);
}
```

Here, the `dddd` indicates the types of the arguments: they are all double-precision floating point. If a value popped from the stack is not a double, Angort will attempt to convert it (see Sec. 3.5). Once popped, the distance is calculated and the resulting double is pushed. The code generated is

```
static void _word__distance(angort::Runtime *a)
{
Value *_parms[4];
a->popParams(_parms,"dddd");
double p0 = _parms[0]->toDouble();
double p1 = _parms[1]->toDouble();
double p2 = _parms[2]->toDouble();
double p3 = _parms[3]->toDouble();
    float dx = (p0-p2);
    float dy = (p1-p3);
    float dist = sqrt(dx*dx+dy*dy);
    a->pushDouble(dist);
}
```

It uses the internal `popParams` function to pop the parameters into an array, typechecking them as it does so, and then creates the local variables `p0, p1`... as discussed above, converting to the appropriate types. Then our code follows.

Lists and hashes are rather more complex. In the example below, we create a list and add two strings to it:

```
%wordargs stringstolist ss (string string -- list) list from two strings
{
    Value v;
    ArrayList<Value> *list = Types::tList->set(&v);
    Types::tString->set(list->append(),p0);
    Types::tString->set(list->append(),p1);
    a->pushval()->copy(&v);
}
```

The two parameters `p0` and `p1` are popped and initialised as `const char *` by the prologue created by `makeWords.pl`. Then a local `Value` variable is declared, and set to hold a new list by a call to the `set()` method of the list type object, `Types::tList` (see Sec. 22.1),. The new list is stored in the variable `list`. Then, for each string, a call is made to the list's `append()` method which returns a new `Value` pointer, and the string type object's `set()` method is used to set this new value to a string. Finally, a new value is pushed onto the stack, and `v` – the value which wraps the list – is copied into it.

22.3.5 Important "gotcha": overwriting stack values

Note that our code above doesn't look like this:

```
%wordargs stringstolist ss (string string -- list) list from two strings
{
    ArrayList<Value> *list = Types::tList->set(a->pushval());
    Types::tString->set(list->append(),p0);
    Types::tString->set(list->append(),p1);
}
```

That is, we don't push a new value onto the stack and start writing to it immediately. That's because the following sequence would occur:

- pop the stack into p0 and p1 (in the prologue)
- push the stack

- set the new stack top to be a list – overwriting the value which was popped into p0, replacing the string pointer with a list pointer

This is why we write our return value into a local variable which we push at the end of function.

22.3.6 %wordargs type specification

The string immediately following the function name in a `%wordargs` specification gives the types of the parameters, each by a single character. If you want a variadic function, use `%word` and the Angort `pop..()` functions to pop the values one by one. The type letters are given below.

Letter	C++ type	notes
n	`float`	
L	`long`	
b	`bool`	
i	`integer`	
d	`double`	
c or v	`Value *`	raw Angort value
s or S	`const char *`	string or symbol
y	`const char *`	string, symbol or none (NULL)
l	`ArrayList<Value>*`	list
h	`Hash *`	hash
A or B	user type (see below)	

22.3.7 The %type directive and user types

This is used to tell `makeWords.pl` about a user-defined type. It takes three arguments: the name of the type, the static global variable containing the type object, and the name of the object returned by calling the type object's `get()` method with a `Value` pointer.

To define a user type, we need to define a type object for it. Given that most user types are rather more than simple values, this example will cover creating new garbage-collected types. Let's consider complex numbers, which require the storage of two double-precision values: real and imaginary components. Our complex number class will look like this, inheriting `GarbageCollected`:

```
class Complex : public GarbageCollected {
public:
    double r,i;

    virtual ~Complex(){}
    Complex(double _r,double _i){
        r = _r;
        i = _i;
    }
};
```

Now we can define the type object:

```
class ComplexType : public GCType {
public:
    ComplexType(){
        add("complex","CPLT");
    }

    virtual ~ComplexType(){}

    Complex *get(const Value *v) const {
        if(v->t!=this)
            throw RUNT(EX_TYPE,"not a complex number");
        return (Complex *)(v->v.gc);
    }

    void set(Value *v,double r,double i){
        v->clr();
        v->t=this;
        v->v.gc = new Complex(r,i);
        incRef(v);
    }

    virtual const char *toString(bool *allocated,const Value *v) const {
        char buf[128];
        Complex *w = get(v);
        snprintf(buf,128,"%f+%fi",w->r,w->i);
        *allocated=true;
        return strdup(buf);
    }
};
```

This code should appear after the `%name` and `%shared` directives. The important members here are `get()`, which ensures the passed `Value` is the right type and returns the underlying structure; and `set()`, which clears the value of any previous data, sets the type pointer, initialises a new object and increments the reference count. Also of note is the constructor, which calls `add()` to register the type with Angort, passing in two unique identifiers – long and 4-character (the latter for quick comparison). Finally, we provide a string conversion method – this type will be automatically converted to a string when required.

Now we have our types, we can register them:

```
static ComplexType tC;
%type complex tC Complex
```

This declares and initialises the singleton type object, and registers the type with `makeWords.pl` under the name `complex`. We can now define words which use this type. First, the easy case of making a new complex number:

```
%wordargs complex dd (r i -- complex)
{
    tC.set(a->pushval(),p0,p1);
}
```

This just pushes a value, and passes it to our new type object's `set()` method with the parameters (two doubles).

If we want to write a word which takes a complex number as a parameter, we need to use the user type facility of `%wordargs`. If you specify A or B as a type, a user type name or two (comma-separated) should follow the specification after a vertical bar. This is the type name as given as the first argument of `%type`. Here are functions to extract the real and imaginary components:

```
%wordargs real A|complex (complex -- real part)
{
    a->pushDouble(p0->r);
}

%wordargs img A|complex (complex -- img part)
{
    a->pushDouble(p0->i);
}
```

22.3.8 Defining binary operators

We can define functions for the binary operators by using `%binop` and the names of two types, separated by the operator name, one of: `equals`, `nequals`, `add`, `mul`, `div`, `sub`, `and`, `or`, `gt`, `lt`, `le`, `lt`, `mod`, `cmp`. The types can be user-defined or internal:

```
%binop complex mul complex
{
    Complex *l = tC.get(lhs); // use tC to get Complex pointer from Value
    Complex *r = tC.get(rhs);
    // calculate result
    double real = (l->r * r->r) - (l->i * r->i);
    double img = (l->r * r->i) + (l->i * r->r);
    // push new complex.
    tC.set(a->pushval(),real,img);
}

%binop complex add complex
{
    Complex *l = tC.get(lhs);
    Complex *r = tC.get(rhs);
    tC.set(a->pushval(),l->r+r->r,l->i+r->i);
}

%binop complex add double
{
    Complex *l = tC.get(lhs);
    float r = rhs->toDouble();
    tC.set(a->pushval(),l->r+r,l->i);
}
// there would be more down here!
```

Note that binop functions take two `Value` pointers; no automatic parameter unwrapping *á la* `%wordargs` is done. Also note that the type handling in binops is quite strict: with the above definitions

```
4 5 complex 1.0d + .
```

will work, but

```
4 5 complex 1 + .
```

will not, because there is no registered binary operator which takes a complex number and an integer.

23 Topics missing from this document

There are several topics which will be discussed in later versions of this document

- `def`, `defconst` and `setglobal`
- routines (generators) and `yield`
- constant expressions using << and >>
- sourceline
- exception and signal handling
- barewords

- many more built-in words
- IO handling
- wrappers.h for plugins
- properties (which look like global variables but invoke Angort code)
- embedding an Angort interpreter in other systems

24 Standard function documentation

This section contains automatically generated documentation for all words in the standard imported namespaces, i.e. those which are compiled into Angort and do not require fully qualified named with `$` in them. It also includes words in the `future` and `deprecated` namespaces.

24.1 future

24.1.1 slice

`(iterable start end -- iterable)` produce a slice of a string or list

Return a slice of a string or list, returning another string or or list, given the start and end positions. The end index is exclusive; that element will not be included. Inappropriate types will throw ex$notcoll. If length of the iterable is greater than the length required, then the output will be truncated to the remainder of the iterable. Negative indices count from just past the end, so -1 is the last element. A zero end index thus means the end. Empty results will be returned if the requested slice does not intersect the iterable.

24.2 deprecated

24.2.1 slice

`(iterable start len -- iterable)` produce a slice of a string or list

Return a slice of a string or list, returning another string or or list, given the start and length. Inappropriate types will throw ex$notcoll. If length of the iterable is greater than the length requested, then the slice will go to the end. If the start<0 it is counted from the end. Empty results will be returned if the requested slice does not intersect the iterable.

24.3 cli

24.3.1 prompt

`(function --)` set prompt callback

Sets a function to be called to generate the prompt. This is a function which has the form (gc stkcount promptchar – string). Note that any ANSI escape sequences should be wrapped with characters 001 and 002.

24.3.2 args

`(-- args)` list of arguments, stripped of those handled by Angort

This returns the arguments passed to the program, except for those which are parsed by Angort itself (e.g. "-d"). Any arguments after "`--`" are not stripped and are not handled by Angort, and the "`--`" itself is not added to this list.

Index