Part II

Primer

Chapter 6

Introduction

Don't you hate it? You've just got a new programming language and you're trying to write your first program. You want to use a certain feature (you know it's got to be there) and you can't find it in the manual.

I've had that experience many times. So when I wrote 4tH I promised myself, that would not happen to 4tH-users. In this manual you will find many short features on all kind of topics. How to input a number from the keyboard, what a cell is, etc.

I hope this will enable you to get quickly on your way. If it didn't, email me at 'The.Beez.-speaks@gmail.com'. You will not only get an answer, but you will help future 4tH users as well.

You can use this manual two ways. You can either just get what you need or work your way through. Every section builds on the knowledge you obtained in the previous sections. All sections are grouped into levels. We advise you to use what you've learned after you've worked your way through a level.

There are six levels. First, 4tH fundamentals. It assumes a working knowledge of programming and covers the basics. Second, 4tH arrays. We'll try to explain to you what an address is and teach you basic string handling.

Third, 4tHs Character Segment. We'll explain you how it is laid out and what you can do with it. Fourth, 4tHs Integer Segment and Code Segment. We'll explain you how it is laid out and what you can do with it.

Finally, advanced programming techniques. First the builtin facilities 4tH offers and after that the extra features the 4tH library offers. We'll teach you how to program multilevel exits, write interpreters, use jump-tables, emulate floating point calculation and a lot more!!

I don't think it is enough to teach you Forth, from which 4tH was derived, but you can always get a good textbook on Forth, like "Starting Forth" by Leo Brodie. Have fun!

Chapter 7

4tH fundamentals

7.1 Making calculations without parentheses

To use 4tH you must understand Reverse Polish Notation. This is a way to write arithmetic expressions. The form is a bit tricky for people to understand, since it is geared towards making it easy for the computer to perform calculations; however, most people can get used to the notation with a bit of practice.

Reverse Polish Notation stores values in a stack. A stack of values is just like a stack of books: one value is placed on top of another. When you want to perform a calculation, the calculation uses the top numbers on the stack. For example, here's a typical addition operation:

```
1 2 +
```

When 4tH reads a number, it just puts the value onto the stack. Thus 1 goes on the stack, then 2 goes on the stack. When you put a value onto the stack, we say that you push it onto the stack. When 4tH reads the operator '+', it takes the top two values off the stack, adds them, then pushes the result back onto the stack. This means that the stack contains:

```
3
```

after the above addition. As another example, consider:

```
2 3 4 + *
```

(The '*' stands for multiplication.) 4tH begins by pushing the three numbers onto the stack. When it finds the '+', it takes the top two numbers off the stack and adds them. (Taking a value off the stack is called popping the stack.) 4tH then pushes the result of the addition back onto the stack in place of the two numbers. Thus the stack contains:

```
2 7
```

When 4tH finds the '*' operator, it again pops the top two values off the stack. It multiplies them, then pushes the result back onto the stack, leaving:

```
14
```

The following list gives a few more examples of Reverse Polish expressions. After each, we show the contents of the stack, in parentheses.

```
7 2 -          (5)
2 7 -          (-5)
12 3 /         (4)
-12 3 /        (-4)
4 5 + 2 *      (18)
4 5 2 + *      (28)
4 5 2 * -      (-6)
```

7.2 Manipulating the stack

You will often find that the items on the stack are not in the right order or that you need a copy. There are stack-manipulators which can take care of that.

To display a number you use '.', pronounced "dot". It takes a number from the stack and displays it. 'SWAP' reverses the order of two items on the stack. If we enter

```
2 3 . . cr
```

4tH answers:

```
3 2
```

If you want to display the numbers in the same order as you entered them, you have to enter:

```
2 3 swap . . cr
```

In that case 4tH will answer:

```
2 3
```

You can duplicate a number using 'DUP'. If you enter:

```
2 . . cr
```

4tH will complain that the stack is empty. However, if you enter:

```
2 dup . . cr
```

4tH will display:

```
2 2
```

Another way to duplicate a number is using 'OVER'. In that case not the topmost number of the stack is duplicated, but the number beneath. E.g.

```
2 3 dup . . . cr
```

will give you the following result:

```
3 3 2
```

But this one:

```
2 3 over . . . cr
```

will give you:

```
2 3 2
```

Sometimes you want to discard a number, e.g. you duplicated it to check a condition, but since the test failed, you don't need it anymore. 'DROP' is the word we use to discard numbers. So this:

```
2 3 drop .
```

will give you "2" instead of "3", since we dropped the "3".

The final one I want to introduce is 'ROT'. Most users find 'ROT' the most complex one since it has its effects deep in the stack. The thirdmost item to be exact. This item is taken from its place and put on top of the stack. It is 'rotated', as this small program will show you:

```
1 2 3               \ 1 is the thirdmost item
. . . cr            \ display all numbers
                    ( This will display '3 2 1' as expected)
1 2 3               \ same numbers stacked
rot                 \ performs a 'ROT'
. . . cr            \ same operation
                    ( This will display '1 3 2'!)
```

7.3 Deep stack manipulators

No, there are no manipulators that can dig deeper into the stack. A stack is NOT an array! So if there are some Forth-83 users out there, I can only tell you: learn Forth the proper way. Programs that have so many items on the stack are just badly written. Leo Brodie agrees with me.

If you are in 'deep' trouble you can always use the returnstack manipulators. Check out that section.

7.4 Passing arguments to functions

There is no easier way to pass arguments to functions as in 4tH. Functions have another name in 4tH. We call them "words". Words take their "arguments" from the stack and leave the "result" on the stack.

Other languages, like C, do exactly the same. But they hide the process from you. Because passing data to the stack is made explicit in 4tH it has powerful capabilities. In other languages, you can get back only one result. In 4tH you can get back several!

All words in 4tH have a stack-effect-diagram. It describes what data is passed to the stack in what order and what is returned. The word '*' for instance takes numbers from the stack, multiplies them and leaves the result on the stack. It's stack-effect-diagram is:

```
n1 n2 -- n3
```

Meaning it takes number n1 and n2 from the stack, multiplies them and leaves the product (number n3) on the stack. The rightmost number is always on top of the stack, which means it is the first number which will be taken from the stack. The word '.' is described like this:

```
n --
```

Which means it takes a number from the stack and leaves nothing. Now we get to the most powerful feature of it all. Take this program:

```
2     ( leaves a number on the stack)
3     ( leaves a number on the stack on top of the 2)
*     ( takes both from the stack and leaves the result)
.     ( takes the result from the stack and displays it)
```

Note that all data between the words '*' and '.' is passed implicitly! Like putting LEGO stones on top of another. Isn't it great?

7.5 Making your own words

Of course, every serious language has to have a capability to extend it. So has 4tH. The only thing you have to do is to determine what name you want to give it. Let's say you want to make a word which multiplies two numbers and displays the result.

Well, that's easy. We've already seen how you have to code it. The only words you need are '*' and '.'. You can't name it '*' because that name is already taken. You could name it 'multiply', but is that a word you want to type in forever? No, far too long.

Let's call it '*.'. Is that a valid name? If you've programmed in other languages, you'll probably say it isn't. But it is! The only characters you can't use in a name are whitespace characters (<CR>, <LF>, <space>, <TAB>). Note that 4tH is not case-sensitive!

So '*.' is okay. Now how do we turn it into a self-defined word. Just add a colon at the beginning and a semi-colon at the end:

```
: *. * . ;
```

That's it. Your word is ready for use. So instead of:

```
2 3 * .
```

We can type:

```
: *. * . ;
2 3 *.
```

And we can use our '*.' over and over again. Hurray, you've just defined your first word in 4tH!

7.6 Adding comment

Adding comment is very simple. In fact, there are two ways to add comment in 4tH. That is because we like programs with a lot of comments.

You've already encountered the first form. Let's say we want to add comment to this little program:

```
: *. * . ;
2 3 *.
```

So we add our comment:

```
: *. * . ;          This will multiply and print two numbers
2 3 *.
```

4tH will not understand this. It will desperately look for the words 'this', 'will', etc. However the word '\' will mark everything up to the end of the line as comment. So this will work:

```
: *. * . ;        \ This will multiply and print two numbers
2 3 *.
```

There is another word called '(' which will mark everything up to the next ')' as comment. Yes, even multiple lines. Of course, these lines may not contain a ')' or you'll make 4tH very confused. So this comment will be recognized too:

```
: *. * . ;        ( This will multiply and print two numbers)
2 3 *.
```

Note that there is a whitespace-character after both '\' and '('. This is mandatory! However the closing paren) does not have to have a leading blank space. It is optional.

7.7 Text-format of 4tH source

4tH source is a simple ASCII-file. And you can use any layout as long a this rule is followed:

> All words are separated by at least one whitespace character!

Well, in 4tH everything is a word or becoming a word. Yes, even '\' and '(' are words! And you can add all the empty lines or spaces or tabs you like, 4tH won't care and your harddisk supplier either.

7.8 Displaying string literals

Displaying a string is as easy as adding a comment. Let's say you want to make the ultimate program, one that is displaying "Hello world!". Well, that's almost the entire program. The famous 'hello world' program is simply this in 4tH:

```
." Hello world!"
```

Compile this and it works. Yes, that's it! No declaration that this is the main function and it is beginning here and ending there. May be you think it looks funny on the display. Well, you can add a carriage return by adding the word 'CR'. So now it looks like:

```
." Hello world!" cr
```

Still pretty simple, huh?

7.9 Creating variables

One time or another you're going to need variables. Declaring a variable is easy.

```
variable one
```

The same rules for declaring words apply for variables. You can't use a name that already has been taken. A variable is a word too! And whitespace characters are not allowed. Note that 4tH is not case-sensitive!

7.10 Using variables

Of course variables are of little use when you could not assign values to them. This assigns the number 6 to variable 'ONE':

```
6 one !
```

We don't call '!' bang or something like that, we call it 'store'. Of course you don't have to put a number on the stack to use it, you can use a number that is already on the stack. To retrieve the value stored in 'ONE' we use:

```
one @
```

The word '@' is called 'fetch' and it puts the number stored in 'one' on the stack. To display it you use '.':

```
one @ .
```

There is a shortcut for that, the word '?', which will fetch the number stored in 'ONE' and displays it:

```
one ?
```

7.11 Built-in variables

4tH has only three built-in variables. They are called 'BASE', '>IN' and 'OUT'. 'BASE' controls the radix at run-time, '>IN' is used by 'PARSE' and 'OUT' returns a value to the host program.

7.12 What is a cell?

A cell is simply the space a number takes up. So the size of a variable is one cell. The size of a cell is important since it determines the range 4tH can handle. It also helps make code portable across machines with different cell sized, for example 16 bit and 32 big systems. We'll come to that further on.

7.13 What is a literal expression?

A literal expression is simply anything that *compiles* to a literal. All numbers, all defined constants and some expressions are compiled to a literal. In the glossary you can find what compiles to a literal, but we list them here too:

```
'           <name>
['']        <name>
[DEFINED]   <name>
[UNDEFINED] <name>
CHAR   <char>
[CHAR] <char>
<literal> [NOT]
<literal> [SIGN]
<literal> NEGATE
<literal> 1+
<literal> 1-
<literal> 2*
<literal> /FIELD
<literal> +FIELD <name>
<literal> ENUM   <name>
<literal> <literal> *
<literal> <literal> /
<literal> <literal> +
<literal> <literal> -
<literal> <literal> [=]
<literal> <literal> [MAX]
```

7.14 Declaring arrays of numbers

You can make arrays of numbers very easily. It is very much like making a variable. Let's say we want an array of 16 numbers:

```
16 array sixteen
```

That's it, we're done! You must omit the word 'CELLS', since 'ARRAY' implicates that you want an array of numbers, not characters. The size is a literal expression. You can't take it from the stack, so this is invalid:

```
3 dup + array sixteen
```

4tH will let you know that this is not a valid construction, but in case you wonder..

7.15 Using arrays of numbers

You can use arrays of numbers just like variables. The array cells are numbered from 0 to N, N being the size of the array minus one. Storing a value in the 0th cell is easy. It works just like a simple variable:

```
5 sixteen 0 th !
```

Which will store '5' in the 0th cell. So storing '7' in the 8th cell is done like this:

```
7 sixteen 8 th !
```

Of course when you want to store a value in the first, second or third cell you have to use 'TH' too, since it is a word. If you don't like that try defining 'ST', 'ND' and 'RD' yourself:

```
: st th ;
: nd th ;
: rd th ;
4 sixteen 1 st !
5 sixteen 2 nd !
6 sixteen 3 rd !
```

Isn't 4tH wonderful? Fetching is done the same of course:

```
sixteen 0 th @
sixteen 4 th @
```

Plain and easy.

7.16 Copying arrays of numbers

If you want to move chunks of data around, there is 'SMOVE':

```
1024 array a
1024 array b

a b 512 smove
```

This wil define two arrays of 1024 cells. 'SMOVE' will move the first 512 cells of array "a" to array "b".

7.17 Declaring and using constants

Declaring a simple constant is easy too. Let's say we want to make a constant called "FIVE":

```
5 constant five
```

Now you can use "FIVE" like you would '5'. E.g. this will print five spaces:

```
five spaces
```

The same rules for declaring words apply for constants. You can't use a name that already has been taken. A constant is a word too! And whitespace characters are not allowed. Note that 4tH is not case-sensitive. By the way, '5' is a literal expression. You can't take it from the stack or calculate it.

A special kind of constant is the so-called "plus-constant". This constant will automatically add itself to the top of the stack when executed, e.g.:

```
10 +constant tenplus
20 tenplus .
```

Will print "30". First we define a '+CONSTANT' named "tenplus", then we throw "20" on the stack, finally we execute "tenplus" and print the result. It is equivalent to:

```
10 constant ten
20 ten + .
```

Yes, you guessed it, a '+CONSTANT' is a constant with built-in addition! Cool, huh? And if you thought it couldn't get any better, there are also '*CONSTANT' and '/CONSTANT', which have built-in multiplication and division.

7.18 Built-in constants

There are several built-in constants. Of course, they are all literals in case you wonder. Here's a list. Refer to the glossary for a more detailed description:

```
/PAD
/TIB
/HOLD
/CELL
/CHAR
MAX-N
MAX-CHAR
CHAR-BITS
(ERROR)
BL
FALSE
LO
APP
PAD
STACK-CELLS
TIB
TRUE
VARS
WIDTH
INPUT
OUTPUT
STDOUT
STDIN
APPEND
PIPE
FILES
4TH#
```

7.19 Using booleans

Booleans are expressions or values that are either true or false. They are used to conditionally execute parts of your program. In 4tH a value is false when it is zero and true when it is non-zero. Most booleans come into existence when you do comparisons. This example will determine whether the value in variable 'VAR' is greater than 5. Try to predict whether it will evaluate to true or false:

```
variable var
4 var !
var @ 5 > .
```

No, it wasn't! But hey, you can print booleans as numbers. Well, they are numbers. But with a special meaning as we will see in the next section.

7.20 IF-ELSE constructs

Like most other languages you can use IF-ELSE constructs. Let's enhance our previous example:

variable var

```
4 var !
var @ 5 >
if ." Greater" cr
else ." Less or equal" cr
then
```

So now our program tells you when it's greater and when not. Note that contrary to other languages the condition comes before the 'IF' and 'THEN' ends the IF-clause. In other words, whatever path the program takes, it always continues after the 'THEN'. A tip: think of 'THEN' as 'ENDIF'..

7.21 FOR-NEXT constructs

4tH has FOR-NEXT constructs as well. The number of iterations is known in this construct. E.g. let's print the numbers from 1 to 10:

```
11 1 do i . cr loop
```

The first number represents the limit. When the limit is reached or exceeded the loop terminates. The second number presents the initial value of the index. That's where it starts off. So remember, this loop iterates at least once! You can use '?DO' instead of 'DO'. That will not enter the loop if the limit and the index are the same to begin with:

```
0 0 ?do i . cr loop
```

'i' represents the index. It is not a variable or a constant, it is a predefined word, which puts the index on the stack, so '.' can get it from the stack and print it.

But what if I want to increase the index by two? Or want to count downwards? Is that possible. Sure. There is another construct to do just that. Okay, let's take the first question:

```
11 1 do i . cr 2 +loop
```

This one will produce exactly what you asked for. An increment by two. This one will produce all negative numbers from -1 to -10:

```
-11 -1 do i . cr -1 +loop
```

Note that the step is not a literal expression. You can change the step if you want to, e.g.:

```
32767 1 do i . i +loop
```

This will print: 1, 2, 4, 8, all up to 16384. Pretty flexible, I guess. You can break out of a loop by using 'LEAVE'. Note that 'LEAVE' only sets the index to the value of the limit: it doesn't branch or anything. Make sure that there is no code left between 'LEAVE' and 'LOOP' that you don't want to execute. So this is okay:

```
10 0 do i dup 5 = if drop leave else . cr then loop
```

And this is not:

```
10 0 do i dup 5 = if drop leave then . cr loop
```

Since it will still get past the '.' before leaving. In this case you will catch the error quickly, because the stack is empty.

7.22 WHILE-DO constructs

A WHILE-DO construction is a construction that will perform zero or more iterations. First a condition is checked, then the body is executed. Then it will branch back to the condition. In 4tH it looks like this:

BEGIN <condition> WHILE <body> REPEAT

The condition will have to evaluate to TRUE in order to execute the body. If it evaluates to FALSE it branches to just after the REPEAT. This example does a Fibbonaci test.

```
: fib 0 1
  begin
    dup >r rot dup r> >     \ condition
  while
    rot rot dup rot + dup . \ body
  repeat
  drop drop drop ;          \ after loop executed
```

You might not understand all of the commands, but we'll get to that. If you enter "20 fib" you will get:

```
1 2 3 5 8 13 21
```

This construct is particularly handy if you are not sure that all data will pass the condition.

7.23 REPEAT-UNTIL constructs

The counterpart of WHILE-DO constructs is the REPEAT-UNTIL construct. This executes the body, then checks a condition at 'UNTIL'. If the expression evaluates to FALSE, it branches back to the top of the body (marked by 'BEGIN') again. It executes at least once. This program calculates the largest common divisor.

```
: lcd
  begin
    swap over mod          \ body
    dup 0=                 \ condition
  until drop . ;
```

If you enter "27 21 lcd" the programs will answer "3".

7.24 Infinite loops

In order to make an infinite loop one could write:

```
begin ." Diamonds are forever" cr 0 until
```

But there is a nicer way to do just that:

```
begin ." Diamonds are forever" cr again
```

This will execute until the end of times, unless you exit the program another way.

7.25 Including source files

4tH has a vocabulary of over 200 words. If you use them in one of your 4tH programs 4tH will recognize them instantly. These words are *internal*.

But if you take a look at the glossary, you'll find that there are a lot of other words too. Words that 4tH will not recognize; they have to be *included* first. These words are *external*.

These words are defined in an *include file*. An include file is just an ordinary ASCII file with 4tH source. You can read them if you want. In order to use these words, you have to tell 4tH where it can find the include file.

This is done by the '[NEEDS' directive, which is equivalent to the COMUS word 'INCLUDE' (which 4tH also supports). Everything up to the next "]" is considered to be a filename, so the path may contain embedded spaces. You can use absolute paths or relative paths, *just make sure that you're starting 4tH from the proper directory*. E.g. this one includes additional ANS-Forth CORE-words from the directory just above 'lib'[1]:

```
[needs lib/anscore.4th]
```

Or:

```
include lib/anscore.4th
```

4tH comes with a rich library of words, which covers a large part of ANS-Forth and COMUS[2] standard words and beyond. They are all located in the 'lib' directory. In the next level we're going to need a lot of these words, so you'd better know how to include them.

[1] If you're not sure where that is, enter the 'lib' directory and execute "cd ..".

[2] In case you wonder, COMUS stands for COMmon USage.

7.26 Getting a number from the keyboard

The word to enter a number from the keyboard can be found in the 'lib' directory and is defined in the `enter.4th` file. To include it you have to tell 4tH. We *assume* your working directory is just above the 'lib' directory[3]:

```
[needs lib/enter.4th]
```

That's all! Now you can use 'ENTER' just like any 4tH word. This will allow you to enter a number and print it:

```
[needs lib/enter.4th]
enter . cr
```

By the way, this is the end of the first level. Take our advise and give it a try!

[3] As a matter of fact, we will *always* assume that! If you don't know what we mean, execute "cd <path to lib directory>" and then "cd ..". Now you're there for sure! Note none of this is any concern to you if you set the **DIR4TH** variable.

Chapter 8

4tH arrays

8.1 Aligning numbers

You may find that printing numbers in columns (I prefer "right-aligned") can be pretty hard. That is because the standard word to print numbers ('.') prints the number and then a trailing space. That is why '.R' was added.

The word '.R' works just like '.' but instead of just printing the number with a trailing space '.R' will print the number right-aligned in a field of N characters wide. Try this and you will see the difference:

```
140 . cr
150 5 .r cr
```

In this example the field is five characters wide, so '150' will be printed with two leading spaces.

8.2 Creating arrays of constants

Making an array of constants is quite easy. First you have to define the name of the array by using the word 'TABLE' or 'CREATE' (which is ANS-Forth). Then you specify all its elements. Note that every element is a literal expression. All elements (even the last) are terminated by the word ','. An example:

```
create sizes 18 , 21 , 24 , 27 , 30 , 255 ,
```

Please note that ',' is a word! It has to be separated by spaces on both ends.

8.3 Using arrays of constants

Accessing an array of constants is very much like accessing an array of numbers. In an array of numbers you access the 0th element like this:

```
sixteen 0 th @
```

When you access the first element of an array of constants you use this construction:

```
sizes 0 th @c
```

The only difference is the word '@C', which is exclusively used to access arrays of constants.

8.4 Using values

A value is a cross-over between a variable and a constant. May be this example will give you an idea:

declaration:

```
variable a                ( No initial value)
1 constant b              ( Literal expression assigned at compiletime)
2 b + value c             ( Expression assigned at runtime)
```

fetching:

```
a @                       ( Variable throws address on stack)
b                         ( Constant throws value on stack)
c                         ( Value throws value on stack)
```

storing:

```
2 b + a !                 ( Expression can be stored at runtime)
                          ( Constant cannot be reassigned)
2 b + to c                ( Expression can be stored at runtime)
```

In many aspects, values behave like variables and can replace variables. The only thing you cannot do is make arrays of values. A value is *not* a literal expression either, so you can't use them to size arrays. In fact, a value is a variable that behaves in certain aspects like a constant.

Note that although 'VALUE' and 'TO' are aliases, it is more portable and more readable to use 'VALUE' for declaration and 'TO' for reassignment.

8.5 Creating string variables

In 4tH you have to define the maximum length of the string, like Pascal:

```
10 string name
```

You cannot add the 'CHARS' keyword, since 'STRING' already implies that you are creating an array of characters. Note that the string variable includes the terminator. That is a special character that tells 4tH where the string ends (see section 8.14). You usually don't have to add that yourself because 4tH will do that for you. But you will have to reserve space for it.

That means that the string "name" we just declared can contain up to nine characters *AND* the terminator. These kind of strings are usually referred to as ASCIIZ strings.

E.g. when you want to define a string that has to contain "Hello!" (without the quotes) you have to define a string that is at least 7 characters long:

```
7 string hello
```

8.6 What is an address?

An address is a location in memory. Usually, you don't need to know addresses, because 4tH will take care of that. But if you want it, you can retrieve them as we will show you later. Think of memory like a city. It has roads and houses and inhabitants. There are three roads in 4tH city:

1. INTEGER SEGMENT, that is where the cells live;
2. CHARACTER SEGMENT, that is where the strings live;
3. CODE SEGMENT, that is where the instructions that form your program live.

If you want to visit a certain person, you go to the city where he lives, find the right street and knock on the door. If you want to retrieve a certain string or integer, you do the same.

When you define a string, you actually create a constant with the address of that string. When you later refer to the string you just defined its address is thrown on the stack. An address is simply a number that refers to its location. As you will see you can work with string-addresses without ever knowing what that number is. But *because* it is a number you can manipulate it like any other number. E.g. this is perfectly valid:

```
16 string hello

hello                       \ address of string on stack
dup                         \ duplicate it
drop drop                   \ drop them both
```

Later, we will tell you how to get "Hello!" into the string.

8.7 String literals

In 4tH a string literal is created by the word 'S"'. The word 'S"' is very much like '."', but instead of printing it to the screen you will just be defining a string literal.

```
s" This is a string"
```

4tH is a stack oriented language, so what does 'S"' leave on the stack? In 4tH, a string is usually represented by on the stack by its address and its count. So in order to get its length, you only have to get the first value on the stack. In order to get its address you have to get the second value on the stack, which is demonstrated by this small program:

```
s" This is a string"        \ create a temporary string
." Length : " . cr          \ show the length
." Address: " . cr          \ show the address
```

And what about string literals with quotes. Easy, there is an equivalent to 'S"' that does the same thing:

```
s| "This is a string with quotes"|
." Length : " . cr          \ show the length
." Address: " . cr          \ show the address
```

Instead of a quote, the string is delimited by a bar. And what about string literals that include them both? Bad luck? Well, almost but not quite. Just take a look at section 13.21.

8.8 String constants

String constants work the same way as numeric constants:

```
10 constant ten             \ define a string constant
ten . cr                    \ equivalent to: 10 . cr
```

In fact, you give a name to a literal value. After that, you can refer to that literal throughout your program by using its name. String constants do the same thing. Take a look at this little piece of code:

```
s" This is a string"        \ create a temporary string
." Length : " . cr          \ show the length
." Address: " . cr          \ show the address
```

Now we do the same thing, but this time we define a string constant by using 'SCONSTANT':

```
s" This is a string" sconstant mystring
                            \ define a string constant
mystring                    \ now we use the string constant
." Length : " . cr          \ show the length
." Address: " . cr          \ show the address
```

Why use string constants? Well, first of all, if you use a string constant throughout your program, it will save you some editing when you have to change your program for one reason or another. Second, it will make your program a little smaller.

8.9 Initializing string variables

You can initialize a string with the 'S"' word. If you want the string to contain your first name use this construction:

```
s" Hello!" name place
```

The word 'PLACE' copies the contents of a string literal into a string-variable.

If you still don't understand it yet, don't worry. As long as you use this construction, you'll get what you want. Just remember that assigning a string literal to a string that is too short will result in an error or even worse, corrupt other strings.

8.10 Initializing a NULL string variable

If you're not sure what that means, it means we're initializing a string variable to an empty string. Well, that's very easy:

```
0 name c!
pad 0 name place
0 dup name place
```

Choose any of these three. And all these constucts are compatible with ANS-Forth.

8.11 Getting the length of a string variable

You get the length of a string variable by using the word 'COUNT'. It will not only return the length of the string variable, but also the string address. It is illustrated by this short program:

```
32 string greeting          \ define string greeting
s" Hello!" greeting place   \ set string to 'Hello!'
greeting count              \ get string length
." String length: " . cr    \ print the length
drop                        \ discard the address
```

Most string handling words return or take an address/count pair. One of the exceptions is the string variable itself (see section 8.9). To copy the contents of an address/count pair represented string into a string variable, we use 'PLACE'. In order to convert a string variable back to an address/count pair represented string, we use 'COUNT':

```
32 string my-string         \ create a string variable
                            \ create an address/count
s" This is a string"        \ pair represented string
my-string place             \ copy it into the variable
my-string count             \ convert it into an address/count pair
." Length : " . cr          \ show the length
." Address: " . cr          \ show the address
```

Note that the contents of the string variable do not change by a 'COUNT' conversion!

8.12 Printing a string variable

Printing a string variable is pretty straight forward. The word that is required to print a string variable is 'TYPE'. It requires an address/count pair. Yes, that are the values that are left on the stack by 'COUNT'! So printing a string means issuing both 'COUNT' and 'TYPE':

```
32 string greeting          \ define string greeting
s" Hello!" greeting place   \ set string to 'Hello!'
greeting count type cr      \ print the string
```

If you don't like this you can always define a word like 'PRINT$':

```
: print$ count type ;
32 string greeting          \ define string greeting
s" Hello!" greeting place   \ set string to 'Hello!'
greeting print$ cr          \ print the string
```

8.13 Copying a string variable

You might want to copy one string variable to another. Let's take a look at this example:

```
32 string one                \ define the first string
32 string two                \ define the second string

s" Greetings!" one place     \ initialize string one
one count                    \ get the length of string one
two place                    \ and copy it into string two
two count type cr            \ print string two
```

First we place the string "Greetings!" into a string variable. 'S"' will put an address/count pair on the stack, that is consumed by 'PLACE'. Variable "ONE" only puts its address on the stack, that is converted into an address/count pair by 'COUNT'. After it has been consumed again by 'PLACE' we need 'COUNT' again to provide 'TYPE' with an address/count pair.

8.14 The string terminator

In order for 'COUNT' to work, it has to know where the string stops. So a special character at the end of the string, the string terminator, is used to indicate the end of an ASCIIZ string. It has nothing to do with Arnold Schwarzenegger obliterating innocent strings! It is simply a character, having the ASCII value zero. It may also be referred to as the NULL-character. Although most strings in 4tH will be terminated automatically it is considered bad style to rely on that.

If you have doubts, you can always convert an address/count pair to a terminated string by applying '>STRING', although you have to take care that nothing is overwritten and enough space is available. As a rule of the tumb you might say that '>STRING' can safely be applied to all string variables.

8.15 Slicing strings

Slicing strings is just like copying strings. We just don't copy all of it and we don't always start copying at the beginning of a string. We'll show you what we mean:

```
32 string one                \ define string one
s" Hans Bezemer" one place   \ initialize string one
one count 2dup type cr       \ duplicate and print it
1 /string                    \ move one character forward
2dup type cr                 \ duplicate and print it again
1 /string                    \ move one character forward
2dup type cr                 \ duplicate and print it again
1 /string                    \ move one character forward
type cr                      \ print it for the last time
```

First it will print "Hans Bezemer", then "ans Bezemer", then "ns Bezemer" and finally "s Bezemer". The word '/STRING' adjusts the address/count pair by a given number of characters, in this case one character. The word '2DUP' is much like 'DUP', but it copies the top *two* values on the stack. It is functionally equivalent to:

```
over over
```

If we want to discard the first name at all we could even write:

```
32 string one                \ define string one
s" Hans Bezemer" one place   \ initialize string one
one count 5 /string type cr  \ print sliced string
```

The five characters we want to skip are the first name (which is four characters) and a space (which adds up to five). There is a special word for slicing strings in the library member `slice.4th`. You call it with:

```
address count position-to-start position-to-end
```

Both positions start counting at zero. So this will copy the first name to string "two" and print it:

```
[needs lib/slice.4th]

32 string one                \ declare string one
32 string two                \ declare string two
s" Hans Bezemer" one place   \ initialize string one
one count 0 3 slice          \ slice the first name
two place                    \ copy it to string two
two count type cr            \ print string two
```

This will slice the last name off and store it in string "two":

```
[needs lib/slice.4th]

32 string one                \ declare string one
32 string two                \ declare string two
s" Hans Bezemer" one place   \ initialize string one
one count 5 11 slice         \ slice the last name
two place                    \ copy it to string two
two count type cr            \ print string two
```

Since the last name is seven characters long and starts at position five (start counting with zero!).

8.16 Appending strings

The word '+PLACE[1]' appends two strings. In this example string "one" holds the first name. The second string literal is appended to string "one" to form the full name. Finally string "one" is printed.

```
32 string one                \ define string one

s" Hans " one place          \ initialize first string
s" Bezemer" one +place       \ append 'Bezemer' to string
one count type cr            \ print first string
```

[1]There is a COMUS word called 'APPEND' which works exactly the same.

8.17 Comparing strings

If you ever sorted strings you know how indispensable comparing strings is. As we mentioned before, there are very few words in Forth that act on strings. Here is a word that can compare two strings. It is located in the library member `compare.4th`.

```
[needs lib/compare.4th]
                              \ compare two chars
32 string one                 \ define string one
s" Hans Bezemer" one place    \ initialize string one
32 string two                 \ define string two
s" HANS BEZEMER" two place    \ initialize string two

one count two count compare   \ compare two strings
if
  ." Strings differ"          \ message: strings ok
else
  ." Strings are the same"    \ message: strings not ok
then
cr                            \ send CR
```

Simply pass two strings (represented by their address/count pairs) to 'COMPARE' and it will return a TRUE flag when the strings are different. This might seem a bit odd, but `strcmp()` does exactly the same. If you don't like that you can always add '0=' to the end of 'COMPARE' to reverse the flag.

You'll soon find out that ANS-Forth's 'COMPARE' is case sensitive. Lucky for you, you can modify the behaviour of 4tH's 'COMPARE'. Just define this *before* the '[NEEDS' directive:

```
[pragma] casesensitive
[needs lib/compare.4th]
                              \ compare two chars
32 string one                 \ define string one
s" Hans Bezemer" one place    \ initialize string one
32 string two                 \ define string two
s" HANS BEZEMER" two place    \ initialize string two

one count two count compare   \ compare two strings
if
  ." Strings differ"          \ message: strings ok
else
  ." Strings are the same"    \ message: strings not ok
then
cr                            \ send CR
```

Now 'COMPARE' will do a case sensitive comparison.

8.18 Finding a substring

Sometimes you need to find a string within a string. ANS-Forth has defined a word for that too. It is called 'SEARCH'. You need to include `search.4th` in order to use it. Now lets find "the" in this string:

```
[needs lib/search.4th]

s" How the cow catches the hare"
s" the" search                \ search for 'the'
0= if ." not " then ." found: "
type                          \ print the result
```

'SEARCH' always returns a flag and a address/count pair. If it returns true, the substring was found; if it returns false, the substring was *not* found. Now that's pretty straightforward, isn't it? That means that the small program above will print:

```
found:
```

When the substring was found and:

```
not found:
```

When the substring was not found. But what kind of string does it return when the substring was not found? Well, the entire string you fed it, so this *would have been* its output if we had been looking for the substring "now" instead of "the":

```
not found: How the cow catches the hare
```

But in this specific example we are looking for "the". When found, 'SEARCH' returns the string after the first occurrence of the substring we were looking for:

```
found: the cow catches the hare
```

Why that? Why not a position? Well, first of all, you can look for the same substring again:

```
[needs lib/search.4th]

s" How the cow catches the hare"
s" the" search drop          \ drop the flag
2dup type                    \ print the string
s" the" search drop          \ now search again
type                         \ print the string
```

This will print:

```
the cow catches the hare
the hare
```

But if you still want to see a position instead of a string, you can simply define this:

```
[needs lib/search.4th]

: position
  2>r over swap 2r> search 0= >r drop swap - r> if 1- then
;

s" How the cow catches the hare"
s" the" position . cr
```

That will take care of your problems. If the substring was found, "POSITION" will return a positive number. If it wasn't found, it will return a negative number. Note that 'SEARCH' can be persuaded to do a case-sensitive comparison, just like 'COMPARE':

```
[pragma] casesensitive
[needs lib/search.4th]
```

Now 'SEARCH' will do a case sensitive comparison, just like 'COMPARE'.

8.19 Replacing substrings

Sometimes finding is not enough. You have replace it by something else. You can do that very easily with 4tH. Just include `replace.4th`. It contains a word that will do all that. Take this example:

```
[needs lib/replace.4th]

s" How the cow catches the hare" s" the" s" a"
replaceall type cr
```

It will print:

```
How a cow catches a hare
```

Yes, this one replaces all occurrences of ”the” by ”a”. Note that like ’COMPARE’ and ’SEARCH’ this one can be made case sensitive too:

```
[pragma] casesensitive
[needs lib/replace.4th]
```

8.20 Deleting substrings

Yes, we even got a word for ’search-and-destroy’ missions. You only have to include `replace.4th`:

```
[needs lib/replace.4th]

s" How the cow catches the hare" s" the"
deleteall type cr
```

This will print:

```
How cow catches hare
```

Yes, it deletes all occurrences of ”the”. Note that like ’COMPARE’, ’SEARCH’ and ’REPLACE’ this one can be made case sensitive too:

```
[pragma] casesensitive
[needs lib/replace.4th]
```

8.21 Removing trailing spaces

You probably know the problem. The user of your well-made program types his name and hits the spacebar before hitting the enter-key. There you go. His name will be stored in your datafile with a space and nobody will ever find it.

In 4tH there is a special word called ’-TRAILING’ that removes the extra spaces at the end with very little effort. Just paste it after ’COUNT’. Like we did in this example:

```
32 string one                \ define a string
s" Hans Bezemer    "         \ string with trailing spaces
one place                    \ now copy it to string one

one dup                      \ save the address

." ["                        \ print a bracket
count type                   \ old method of printing
." ]" cr                     \ print bracket and newline

." ["                        \ print a bracket
count -trailing type         \ new method of printing
." ]" cr                     \ print a bracket and newline
```

You will see that the string is printed twice. First with the trailing spaces, second without trailing spaces.

8.22 Removing leading spaces

And what about leading spaces? Patience, old chap. You've got a lot of ground to cover. There is no built-in word for that, but we can use a library member like we did in this example:

```
[needs lib/leading.4th]

32 string one                \ define a string
s"     Hans Bezemer"         \ string with leading spaces
one place                    \ now copy it to string one

one dup                      \ save the address

." ["                        \ print a bracket
count type                   \ old method of printing
." ]" cr                     \ print bracket and newline

." ["                        \ print a bracket
count -leading type          \ new method of printing
." ]" cr                     \ print a bracket and newline
```

You will see that the string is printed twice. First with the leading spaces, second without leading spaces. Happy?

8.23 Upper and lower case

Sometimes you will have to convert a string to upper or lower case. 4tH has a library member for that too. Just include:

```
[needs lib/ulcase.4th]
```

This will define several easy to use conversion words. E.g. in order to convert a string to upper case, just enter:

```
s" Convert this!" s>upper    \ convert addr/count string to uppercase
type cr                      \ type the string
```

Its lower case counterpart is:

```
s" Convert this!" s>lower     \ convert addr/count string to lowercase
type cr                       \ type the string
```

Like most string words it takes and returns an address/count pair. Note that the string in question is *modified*, so if you still need the original, copy it first. You can also convert an individual character:

```
char A char>lower emit        \ convert a character and show it
```

And consequently, its counterpart is:

```
char a char>upper emit        \ convert a character and show it
```

These words take an ASCII value from the stack, convert it and put the converted ASCII value back on the stack. If the value does not represent a alphabetic character, it is left unchanged.

8.24 String literals and string variables

Most computer languages allow you to mix string literals and string variables. Not in 4tH. In 4tH they are two distinct datatypes. To print a string literal you use the word '."'. To print a string variable you use the 'COUNT TYPE' construction.

There are only three different actions you can do with a string literal. First, you can define one using 'S"'. Second, you can print one using '."' Finally, you can compile a string into your program using ',"'.

This may seem a bit mind-boggling to you now, but we'll elaborate a bit further on this subject later.

8.25 Printing individual characters

"I already know that!"

Sure you do. If you want to print "G" you simply write:

```
." G"
```

Don't you? But what if you want to use a TAB character (ASCII 9)? You can't type in that one so easily, huh? You may even find it doesn't work at all!

Don't ever use characters outside the ASCII range 32 to 127 decimal. It may or may not work, but it won't be portable anyway. the word 'EMIT' may be of some help. If you want to use the TAB-character simply write:

```
9 emit
```

That works!

8.26 Distinguishing characters

Like in a novel, not all characters are created equal. There are upper case characters, lower case characters, control characters, whitespace, etc. Sometimes it is necessary to find out what kind of character we are dealing with. Of course, 4tH can help you there. You need to include `istype.4th` in order to use it:

```
char a is-lower . cr
char a is-upper . cr
```

4tH will first print a TRUE value (because 'a' is a lower case character) and then a FALSE value. This table tells you what words 4tH offers and the ranges of valid characters:

WORD	RANGE (ASCII)	DESCRIPTION
IS-ASCII	0 - 127	All 7-bit ASCII characters
IS-PRINT	32 - 127	As above, without control characters
IS-WHITE	0 - 32	All control characters plus space
IS-DIGIT	'0' - '9'	All digits
IS-LOWER	'a' - 'z'	All lower case characters
IS-UPPER	'A' - 'Z'	All upper case characters
IS-ALPHA	'a'-'z', 'A' - 'Z'	All alphabetic characters
IS-ALNUM	'0' - '9', 'a' - 'z', 'A' - 'Z'	All alphanumeric characters

Table 8.1: Character typing words

8.27 Getting ASCII values

Ok, 'EMIT' is a nice addition, but it has its drawbacks. What if you want to emit the character "G". Do you have to look up the ASCII value in a table? No. 4tH has another word that can help you with that. It is called 'CHAR'. This will emit a "G":

```
char G emit
```

The word 'CHAR' looks up the ASCII-value of "G" and leave it on the stack. You can also use '[CHAR]'. It does exactly the same thing. It is included for compatibility with ANS-Forth versions. Note that 'CHAR' only works with printable characters (ASCII 33 to 127 decimal).

8.28 Printing spaces

If you try to print a space by using this construction:

```
char   emit
```

You will notice it won't work. Sure, you can also use:

```
."  "
```

But that isn't too elegant. You can use the built-in constant 'BL' which holds the ASCII-value of a space:

```
bl emit
```

That is much better. But you can achieve the same thing by simply writing:

```
space
```

Which means that if you want to write two spaces you have to write:

```
space space
```

If you want to write ten spaces you either have to repeat the command 'SPACE' ten times or use a DO-LOOP construction, which is a bit cumbersome. Of course, 4tH has a more elegant solution for that:

```
10 spaces
```

Which will output ten spaces. Need I say more?

8.29 Fetching individual characters

Take a look at this small program:

```
32 string one                \ define string one
s" Hans" one place           \ initialize string one
```

What is the second character of string "one"? Sure, its an "a". But how can you let your program determine that? You can't use '@' because that word can only access variables.

Sure, you can do that in 4tH, but it requires a new word, called 'C@'. Think of a string as an array of characters and you will find it much easier to picture the idea. Arrays in 4tH always start with zero instead of one. So accessing the first character might be done with:

```
one 0 th c@
```

We do not recommend using this construction, although it will work perfectly. If you never want to convert your program to Forth you might even choose to keep it that way. We recommend the construction:

```
one 0 chars + c@
```

Which is slightly more wordy. 4tH will compile both constructions in exactly the same way. Anyway, accessing the second character is easy now:

```
one 1 chars + c@
```

This is the complete program:

```
32 string one                \ define string one
s" Hans" one place           \ initialize string one
one 1 chars + c@             \ get the second character
emit cr                      \ print it
```

8.30 Storing individual characters

Storing individual characters works just the same. Keep that array of characters in mind. When we want to fetch a variable we write:

```
my_var @
```

When we want to store a value in a variable we write:

```
5 my_var !
```

Fetching only requires the address of the variable. Storing requires both the address of the variable *and* the value we want to store. On top of the stack is the address of the variable, below that is value we want to store. Keep that in mind, this is very important.

Let's say we have this program:

```
32 string one                   \ define string one
s" Hans" one place              \ initialize string one
```

Now we want to change "Hans" to "Hand". If we want to find out what the 4th character of string "one" is we write:

```
32 string one                   \ define string one
s" Hans" one place              \ initialize string one
one 3 chars + c@                \ get the fourth character
```

Remember, we start counting from zero! If we want to store the character "d" in the fourth character, we have to use a new word, and (yes, you guessed it right!) it is called 'C!':

```
32 string one                   \ define string one
s" Hans" one place              \ initialize string one
one 3 chars +                   \ address of the fourth char
char d                          \ we want to store 'd'
swap                            \ get the order right
c!                              \ now store 'd'
```

If we throw the character "d" on the stack before we calculate the address, we can even remove the 'SWAP':

```
32 string one                   \ define string one
char d                          \ we want to store 'd'
s" Hans" one place              \ initialize string one
one 3 chars +                   \ address of the fourth char
c!                              \ now store 'd'
```

We will present the very same programs, but now with stack-effect-diagrams in order to explain how this works. We will call the index 'i', the character we want to store 'c' and the address of the string 'a'. By convention, stack-effect-diagrams are enclosed by parenthesis.

If you create complex programs this technique can help you to understand more clearly how your program actually works. It might even save you a lot of debugging. This is the first version:

```
32 string one                   ( --)
s" Hans" one place              ( --)
one 3 chars                     ( a i)
+                               ( a+i)
char d                          ( a+i c)
swap                            ( c a+i)
c!                              ( --)
```

Now the second, optimized version:

```
32 string one                   ( --)
char d                          ( c)
s" Hans" one place              ( c)
one 3 chars                     ( c a i)
+                               ( c a+i)
c!                              ( --)
```

8.31 Getting a string from the keyboard

Of course, you don't want to initialize strings all your life. Real applications get their input from the keyboard. We've already shown you how to get a number from the keyboard. Now we turn to strings.

When programming in BASIC, strings usually have an undefined length. Some BASICs move strings around in memory, others have to perform some kind of "garbage-collection". Whatever method they use, it takes up memory and processor-time.

4tH forces you to think about your application. E.g. when you want to store somebodies name in a string variable, 16 characters will be too few and 512 characters too many. But 64 characters will probably do.

But that poses a problem when you want to get a string from the keyboard. How can you prevent that somebody types a string that is just too long? And how do you terminate it?

The word 'ACCEPT' takes two arguments. First, the string variable where you want to save the input and second, the maximum number of characters it can take. It automatically terminates the string when reading from the keyboard. But there is a catch. This program can get you into trouble:

```
64 constant #name               \ length of string
#name string name               \ define string 'name'

name #name accept               \ input string
name swap type cr               \ swap count and print
```

Since 64 characters *plus* the terminator add up to 65 characters. The word 'ACCEPT' always returns the number of characters it received. You will find that you won't need that information most of the time.

This is the end of the second level. Now you should be able to understand most of the example programs and write simple ones. I suggest you do just that. Experience is the best teacher after all.

Chapter 9

Character Segment

9.1 The Character Segment

Wonder where all these strings are created? I bet you do. Well, when you define a string, memory is allocated in the Character Segment. When you define another one, space is allocated after the first string. That means that if you go beyond the boundaries of the first string, you'll end up in the space allocated to the second string.

After the second string there is a void. If you end up there your program will end with an error-message. And what about the space before the first string? Well, take a look at figure 9.1.

The lower memory is at the bottom. Yes, before your strings there are two other areas, the TIB and the PAD. We'll elaborate on that in the next section.

The Character Segment is created at run-time. That means that it isn't there when you compile a program. The compiler just keeps track of how much memory would be needed to create such a Character Segment and stores that information in the header.

When you run the program the header is read first. Then the Character Segment is created, so it is already there when your program starts executing. When you exit the program, the Character Segment is destroyed and all information stored there is lost (unless you save it first).

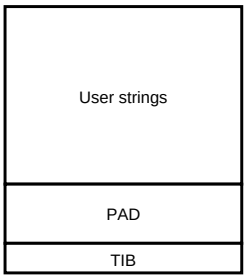

Figure 9.1: Character segment

9.2 What is the TIB?

The TIB stands for "Terminal Input Buffer" and is used by one single, but very important word called 'REFILL'. In essence, 'REFILL' does the same thing as 'ACCEPT', except that it has a dedicated area to store its data and sets up everything for parsing. Whatever you type when you call 'REFILL', it is stored in the TIB.

9.3 What is the PAD?

The PAD is short for "scratch-pad". It is a temporary storage area for strings. It is heavily used by 4tH itself, e.g. when you print a number the string is formed in the PAD. Yes, that's right: when you print a number it is first converted to a string. Then that string is 'COUNT'ed and 'TYPE'd. You can even program that subsystem yourself as we will see when we encounter formatted numbers (see section 9.8).

Furthermore, string constants (compiled by 'S"' or ',"') are temporarily stored in the PAD. Finally, 'NUMBER' and 'ARGS' also use the PAD. The PAD is actually a circular buffer. That means that strings are stored in the PAD until it runs out of space. Then it starts to overwrite the oldest strings. Usually, they have turned into garbage that is no longer used, but sometimes they still have some significance to your program. In that case, you'll have to save the string that was overwritten into a variable. Don't rely on the PAD to keep your strings alive!

9.4 How do I use TIB and PAD?

In general, you don't. The TIB is a system-related area and it is considered bad practice when you manipulate it yourself. The PAD can be used for temporary storage, but beware! Temporary really means temporary. A few words at the most, provided you don't use any string constants.

Think of both these areas as predefined strings. You can refer to them as 'TIB' and 'PAD'. You don't have to declare them in any way. This program is perfectly alright:

```
s" Hello world" pad place      \ store a string in pad
pad count type cr              \ print contents of the pad
```

If you want to know how big TIB and PAD are, you can use the predefined constants '/TIB' and '/PAD':

```
." Size of TIB: " /TIB . cr    \ print sizeof TIB
." Size of PAD: " /PAD . cr    \ print sizeof PAD
```

Note, this does *not* print the length of a string stored in the area, but the maximum size of the string that can be stored there. Some space of the PAD is reserved for number generation (see section 9.3). You can get the size of this area by the predefined constant '/HOLD'. This will print the size of this area and the size of PADs circular buffer:

```
." Size of HOLD  : " /HOLD . cr \ print sizeof HOLD
." Size of buffer: " /PAD /HOLD - . cr
```

If that area did not exist even printing a number could corrupt the circular buffer. In some unusual circumstances, the PAD can get corrupted. If so, identify the temporary string that gets corrupted and store it explitly into a string variable.

9.5 Simple parsing

We have already discussed 'REFILL' a bit. We've seen that it is closely related to 'ACCEPT'. 'REFILL' returns a true flag if all is well. When you use the keyboard it usually is, so we can safely drop it, but we will encounter a situation where this flag comes in handy.

If you want to get a string from the keyboard, you only have to type:

```
refill drop                    \ get string from keyboard
```

Every next call to 'REFILL' will overwrite any previously entered string. So if you want to do something with that string you've got to get it out of there, usually to one of your own strings.

But if accessing the TIB directly is not the proper way, what is? The use of 'REFILL' is closely linked to the word 'PARSE-WORD', which is a parser. 'PARSE-WORD' looks for the delimiter, whose ASCII code is on the stack.

If the string starts with the delimiter, it will skip this and all subsequent occurrences until it finds a string. Then it will look for the delimiter again and slice the string right there. It then returns its address and count.

This extremely handy when you want to obtain filtered input. E.g. when you want to split somebodies name into first name, initials and lastname:

```
Hans L. Bezemer
```

Just use this program:

```
." Give first name, initials, lastname: "
refill drop                     \ get string from keyboard
bl parse-word                   \ parse first name
." First name: "                \ write message
type cr                         \ type first name
bl parse-word                   \ parse initials
." Initials  : "                \ write message
type cr                         \ type initials
bl parse-word                   \ parse last name
." Last name : "                \ write message
type cr                         \ write last name
```

You don't have to parse the entire string with the same character. This program will split up an MS-DOS filename into its components:

```
." DOS filename: " refill      \ input a DOS filename
drop cr                        \ get rid of the flag

char : parse-word              \ parse drive
." Drive: " type ." :" cr
                               \ print drive
begin
  char \ parse-word            \ parse path
  dup 0<>                      \ if not a NULL string
while                          \ print path
  ." Path : " type cr
repeat                         \ parse again
drop drop                      \ discard string
```

If 'PARSE-WORD' reaches the end of the string and the delimiter is still not found, it returns the remainder of that string. If you try to parse beyond the end of the string, it returns a NULL string. That is an empty string or, in other words, a string with length zero.

Therefore, we checked whether the string had zero length. If it had, we had reached the end of the string and further parsing was deemed useless.

9.6 Converting a string to a number

We now learned how to parse strings and retrieve components from them. But what if these components are numbers? Well, there is a way in 4tH to convert a string to a number, but like every number-conversion routine it has to act on invalid strings. That is, strings that cannot be converted to a valid number.

4tH uses an internal error-value, called '(ERROR)'. The constant '(ERROR)' is a strange number. You can't negate it, you can't subtract any number from it and you can't print it. If 4tHs number-conversion word 'NUMBER' can't convert a string it returns that constant. 'ERROR?' checks the return value and leaves an additional true flag if an error occured (which means: '(ERROR)' was returned). Let's take a look at this program:

```
." Enter a number: "           \ write prompt
refill drop                    \ enter string
bl parse-word                  \ parse string
number                         \ convert to a number
error?                         \ test for valid number
if                             \ if not valid
  ." You didn't enter a valid number!" drop cr
else                           \ print if valid
  ." The number was: " . cr
then
```

You first enter a string, then it is parsed and 'PARSE-WORD' returns the address and count. 'NUMBER' tries to convert it. If 'NUMBER' returns '(ERROR)' it wasn't a valid string. Otherwise, the number is right on the stack, waiting to be printed. That wasn't so hard, was it?

9.7 Controlling the radix

If you are a programmer, you know how important this subject is to you. Sometimes, you want to print numbers in octal, binary or hex. 4tH can do that too. Let's take the previous program and alter it a bit:

```
." Enter a number: "           \ write prompt
refill drop                    \ enter string
bl parse-word                  \ parse string
number                         \ convert to a number
error?                         \ test for valid number
if                             \ if not valid
  ." You didn't enter a valid number!" drop cr
else                           \ print if valid
  hex
  ." The number was: " . cr
then
```

We added the word 'HEX' just before printing the number. Now the number will be printed in hexadecimal. 4tH has a number of words that can change the radix, like 'DECIMAL' and 'OCTAL'. They work in the same way as 'HEX'.

4tH always starts in decimal. After that you are responsible. Note that all radix control follows the flow of the program. If you call a self-defined word that alters the radix all subsequent conversion is done too in that radix:

```
: .hex hex . ;                  \ print a number in hex

." Enter a number: "            \ write prompt
refill drop                     \ enter string
bl parse-word                   \ parse string
number                          \ convert to a number
error?                          \ test for valid number
if                              \ if not valid
  ." You didn't enter a valid number!" drop cr
else                            \ print if valid
  ." The number was: " .hex cr
then
```

In this example not only that single number is printed in hex, but also all subsequent numbers will be printed in hex! A better version of the ".HEX" definition would be:

```
: .hex hex . decimal ;
```

Since that one resets the radix back to decimal. Words like 'HEX' do not only control the output of a number, but the input of numbers is also affected:

```
." Enter a number: "            \ write prompt
refill drop                     \ enter string
bl parse-word                   \ parse string
hex                             \ convert hexadecimal
number                          \ convert to a number
error?                          \ test for valid number
if                              \ if not valid
  ." You didn't enter a valid number!" drop cr
else                            \ print if valid
  dup
  ." The number was: " decimal . ." decimal" cr
  ." The number was: " hex . ." hex" cr
then
```

'NUMBER' will now also accept hexadecimal numbers. If the number is not a valid hexadecimal number, it will return '(ERROR)'. You probably know there is more to radix control than 'OCTAL', 'HEX' and 'DECIMAL'. No, we have not forgotten them. In fact, you can choose any radix between 2 and 36. This slightly modified program will only accept binary numbers:

```
: binary 2 base ! ;

." Enter a number: "            \ write prompt
refill drop                     \ enter string
bl parse-word                   \ parse string
binary                          \ convert hexadecimal
number                          \ convert to a number
error?                          \ test for valid number
if                              \ if not valid
  ." You didn't enter a valid number!" drop cr
else                            \ print if valid
  dup                           \ both decimal and hex
  ." The number was: " decimal . ." decimal" cr
  ." The number was: " hex . ." hex" cr
then
```

'BASE' is a predefined variable that enables you to select any radix between 2 and 36. This makes 4tH very flexible. However, this won't work:

```
hex 02B decimal . cr
```

4tH will try to compile "02B", but since it isn't a word or a valid decimal number, it will fail. Words like 'HEX' and the 'BASE' variable work only at run-time, not at compile-time! Isn't there a way to compile non-decimal numbers?

Sure, there is, although it is not that flexible. There are four words that control the interpretation of numbers at compile-time:

1. [BINARY]
2. [OCTAL]
3. [DECIMAL]
4. [HEX]

They work fundamentally different than their run-time equivalents. First, they only work at compile-time. Second, they are interpreted sequentially and do not follow the flow of the program. Let's take a look at these two programs:

```
[binary] 101 . cr
[octal] 101 . cr
[decimal] 101 . cr
[hex] 101 . cr
```

This will print the decimal numbers "5", "65", "101" and "257", since each one of them is compiled with a specific radix.

```
: binary 2 base ! ;
binary 101 . cr
octal 101 . cr
decimal 101 . cr
hex 101 . cr
```

Now the decimal number "101" is printed in four different radixes, since at compile-time the radix was set to decimal (which is the default). Now take a look at this program:

```
: do_binary [binary] ;
: do_decimal [decimal] ;
do_binary 101 decimal . cr
do_decimal 101 decimal . cr
```

The program will print "101" two times! Haven't we selected binary at compile-time? No, both '[BINARY]' and '[DECIMAL]' are interpreted sequentially!

When '[BINARY]' is encountered at the first time, it will set the radix at compile-time to binary. When '[DECIMAL]' is encountered in the second line, it will set the radix to decimal. When the third line it compiled, the radix is still set to decimal. If you want to make this program work, try this:

```
[binary]
101 decimal . cr
[decimal]
101 decimal . cr
```

When the first line is encountered, it sets the radix (at compile-time) to binary. So the number "101" at line two is compiled as a binary number. 'DECIMAL' will just be compiled. It will only influence the radix at run-time. The third line sets the radix at compile-time to decimal. So the number "101" at line four is compiled as a decimal number.

Since the run-time of 4tH starts up in decimal, both occurrences of 'DECIMAL' have little value. We can even eliminate 'DECIMAL' from the program altogether without affecting the result:

```
[binary] 101 . cr
[decimal] 101 . cr
```

Note that both the compile-time radix control words and the run-time radix control words stay in effect until they are superseded by others:

```
[binary]                    \ compile-time binary
101                         \ first binary number
1011                        \ second binary number
[decimal]                   \ compile-time decimal
5                           \ decimal 5
do                          \ set run-time radix
  i base !                  \ to loop-index
  dup . cr                  \ print number
loop
drop                        \ clean stack
```

9.8 Pictured numeric output

You probably have used this before, like when writing Basic. Never heard of "PRINT USING.."? Well, it is a way to print numbers in a certain format. Like telephone-numbers, time, dates, etc. Of course 4tH can do this too. In fact, you've probably used it before. Both '.' and '.R' use the same internal routines. They are called just before a number is printed.

This numeric string is created in the PAD and overwritten with each new call. But we'll go into that a bit later on.

What you have to remember is that you define the format reverse. What is printed first, is defined last in the format. So if you want to print:

```
060-5556916
```

You have to define it this way:

```
6196555-060
```

Formatting begins with the word '<#' and ends with the word '#>'. A single number is printed using '#' and the remainder of the number is printed using '#s' (which is always at least one digit). Let's go a bit further into that:

```
: print# <# #s #> type cr ;
256 print#
```

This simply prints a single number (since only '#S' is between the '<#' and the '#>' and goes to a new line. There is hardly any difference with '.'. You can try any (positive) number. Note that the values that '#>' leaves on the stack can directly be used by 'TYPE'.

This is a slightly different format:

```
: print3# <# # # # #> type cr ;
256 print3#
1 print3#
1000 print3#
```

This one will print "256", "001" and "000". Always the last three positions. The '#' simply stands for 'print a single digit'. So if you want to print a number with at least three digits, the format would be:

```
#s # #
```

That is: print the remainder of the number (at least one digit) and then two more. Now reverse it:

```
# # #s
```

Enclose it by '<#' and '#>' and add 'TYPE CR':

```
<# # # #s #> type cr
```

And that's it! Is it? Not quite. So far we've only printed *positive* numbers. If you try a negative number, you will find it prints garbage. This behavior can be fixed with the word 'SIGN'.

'SIGN' simply takes the number from the stack and prints a "-" when it is negative. The problem is that all other formatting words can only handle positive numbers. So we need the same number twice. One with the sign and one without. A typical signed number formatting word looks like:

```
: signed# dup abs <# #s sign #> type ;
```

Note the 'DUP ABS' sequence. First the number is duplicated (for 'SIGN') and then the absolute value is taken (for the other formatting words). So we got the number on the stack twice. First with sign (for 'SIGN'), second without sign (for the other formatting words). Does that make sense to you?

We can place 'SIGN' wherever we want. If we want to place the sign after the number (like some accountants do) we would write:

```
: account# dup abs <# sign #s #> type ;
```

But that is still not enough to write "$2000.16" is it? Well, in order to do that there is another very handy word called 'HOLD'. The word 'HOLD' just copies any character into the formatted number. Let's give it a try:

```
$2000.16
```

Let's reverse that:

```
61.0002$
```

So we first want to print two numbers, even when they are zero:

```
# # .0002$
```

Then we want to print a dot. This is where 'HOLD' comes in. 'HOLD' takes an ASCII code and places the equivalent character in the formatting string. We don't have to look up the ASCII code for a dot of course. We can use 'CHAR':

```
# # char . hold 0002$
```

Then we want to print the rest of the number (which is at least one digit):

```
# # char . hold #s $
```

Finally we want to print the character "$". Another job for 'HOLD':

```
# # char . hold #s char $ hold
```

So this is our formatting word:

```
: currency <# # # char . hold #s char $ hold #> type cr ;
```

And we call it like this:

```
200016 currency
```

You can do some pretty complex stuff with these formatting words. Try to figure out this one from the master himself, Leo Brodie:

```
: sextal 6 base ! ;
: :00 # sextal # decimal 58 hold ;
: time# <# :00 :00 #S #> type cr ;
3615 time#
```

Yeah, it prints the time! Pretty neat, huh? Now try the telephone-number we discussed in the beginning. That shouldn't be too hard. Still, you may think it's all a bit too complicated for your taste.

Well, there is a solution to that, but you'll need to include a library. Let's say you want to format that darn telephone number - or that currency thing:

```
include lib/picture.4th

605556916 s" ###-#######" picture type cr
200016 s" $?.##" picture type cr
```

That's much easier, isn't it? All you have to do is to issue a string and the library handles the whole thing. The library works very simple: there are five special formatting characters, the rest is copied verbatim. And yes, it handles the sign for you automatically - unless you've specified the position yourself with a '+' character.

You can also use this library to print number in a fixed width field, e.g.:

```
include lib/picture.4th

605616 s" $_____.__" picture type cr
```

This will print the number in a field nine characters wide, including the decimal point and the currency sign. Note you can print *any* character you want by postfixing it with a '!' - yes, even a '!'.

CHARACTER	MEANING
#	Prints a single digit
?	Prints the remainder of the number
_	Prints a single digit, unless all digits have been printed. Then a space is printed
+	Prints a '-' if the number is negative
!	Prints the previous character verbatim, even if it is a formatting character

Table 9.1: Picture library formatting characters

9.9 `printf()` like formatting

Some people can just not be pleased, neither by 4tH's native pictured numeric output, nor by the libraries we offer to make it all a bit easier. They just want `printf()`. And that's alright with me.

Although `printf()` is quite bulky, much slower and is known to have a few quirks of its own, it's not too hard too implement. 4tH has three `printf()` like functions:

printf Supports some basic specifiers, width and writes to 'STDOUT';

sprintf Supports some basic specifiers, one flag, width and writes to a string buffer;

fsprintf Supports most specifiers, all flags, width, precison and writes to a string buffer.

But how do you use it? First you have to include the proper file, either `printf.4th`, `sprintf.4th` or `fsprintf.4th`:

```
include lib/sprintf.4th
```

`printf()` like functions require a format string, optionally with embedded format specifiers. A format specifier has the following layout:

```
%[flags][width][.precision]specifier
```

Flags, width and precision are optional. If you want to print a decimal number, you use the "d" specifier. If you want to print a string, you use the "s" specifier, e.g.:

```
80 string mybuf
s" World" 100 s" A %d hello's from the %s!" mybuf sprintf type cr
```

And this will print(f):

```
A 100 hello's from the World!
```

You may wonder why the number comes before the string, but that is easily explained. First, this is 4tH - we do everything the other way around. Second, the format string is evaluated from left to right, so the number is consumed first. Consequently, it is on the top of the stack.

Now let's say we want to place the number in a field, five positions wide and pad it with spaces where needed:

```
80 string mybuf
s" World" 100 s" A %5d hello's from the %s!" mybuf sprintf type cr
```

Now it comes out this way:

```
A   100 hello's from the World!
```

Of course we can do the same thing for the string, but this time we make it ten positions wide:

```
80 string mybuf
s" World" 100 s" A %5d hello's from the %10s!" mybuf sprintf type cr
```

And this is what comes out of that one:

```
A   100 hello's from the      World!
```

Want to specify that width on the stack? You can:

```
80 string mybuf
s" World" 10 100 s" A %5d hello's from the %*s!" mybuf sprintf type cr
```

Again, the width of "10" on the stack is consumed before the string is printed. Default, everything is right-aligned, but you can change that with the "-" flag:

```
80 string mybuf
s" World" 10 100 s" A %5d hello's from the %-*s!" mybuf sprintf type cr
```

Which gives:

```
A   100 hello's from the World     !
```

Note that although some people may find it easier to use, these words are quite fragile. Make an error in the format string, the order or number of arguments and everything goes haywire. It's your choice.

It goes far beyond the scope of this manual to discuss all the possibilities of `fsprintf.4th`. Even more, because you can only use it in a floating point environment. And that is still to come[1]. For the time being you'll have to do with the following tables.

9.10 Converting a number to a string

Since there is no special word in 4tH which will convert a number to a string, we'll have to create it ourselves. In the previous section we have seen how a numeric string is created in the PAD. We can use this to create a word that converts a number to a string.

Because the PAD is highly volatile, we have to save the string immediately after its creation. So we'll create a word that not only creates the string, but places it directly in its proper location:

```
( n a -- )
: n>string >r dup abs <# #s sign #> r> place ;
```

It takes a number, the address of a string and returns nothing. Example:

```
16 string num$
-1024 num$ n>string
num$ count type cr
```

[1] In sections 12.23 and 12.24, to be exact.

flags	description	printf.4th	sprintf.4th	fsprintf.4th
-	Left-justify within the given field width; Right justification is the default (see width sub-specifier).	N	Y	Y
+	Forces to preceed the result with a plus or minus sign (+ or -) even for positive numbers. By default, only negative numbers are preceded with a - sign.	N	N	Y
(space)	If no sign is going to be written, a blank space is inserted before the value.	N	N	Y
#	Used with o or x specifiers the value is preceeded with 0 or 0x respectively. Used with e, f, or g it forces the written output to contain a decimal point even if no more digits follow. By default, if no digits follow, no decimal point is written.	N	N	Y
0	Left-pads the number with zeroes (0) instead of spaces when padding is specified (see width sub-specifier).	N	N	Y

Table 9.2: List of supported `printf()` flags

9.11 Aborting a program

Some conditions are so grave you can consider them to be fatal errors. In such cases the only thing you can do is abort the program as soon as possible. Of course, there is a way in 4tH to do just that. You can use either 'ABORT' or 'QUIT'. Same thing. Both will terminate your program immediately. This small program prints nothing:

```
abort
." This will never be printed." cr
```

But there is more. Let's say you only want to exit a program when a certain condition is met, e.g. a word left a non-zero value on the stack. In that case you would have to write something like this:

```
if
    ." We have an error condition!" cr quit
then
```

You can write that much shorter by using the word 'ABORT"':

```
abort" We have an error condition!"
```

'ABORT"' will print the message following it and abort, but *only* when there is a non-zero value on the stack. So this program does *not* abort:

```
false abort" This will not be printed!"
." This will be printed!"
```

You will find that 'ABORT"' is a very handy tool when processing error conditions.

width	description	printf.4th	sprintf.4th	fsprintf.4th
(number)	Minimum number of characters to be printed. If the value to be printed is shorter than this number, the result is padded with blank spaces. The value is not truncated even if the result is larger.	Y	Y	Y
*	The width is not specified in the format string, but as an additional integer value argument following the argument that has to be formatted.	Y	Y	Y
.precision				
.number	For integer specifiers (ld, d, o, u, x): precision specifies the minimum number of digits to be written. If the value to be written is shorter than this number, the result is padded with leading zeros. The value is not truncated even if the result is longer. For e, g and f specifiers: this is the number of digits to be printed after the decimal point. For s: this is the maximum number of characters to be printed. By default all characters are printed until the ending null character is encountered. If the period is specified without an explicit value for precision, 0 is assumed.	N	N	Y
.*	The precision is not specified in the format string, but as an additional integer value argument following the argument that has to be formatted.	N	N	Y

Table 9.3: Width and precision of `printf()`

9.12 Opening a file

You probably don't want to write programs that only write to the screen and read from the keyboard. So 4tH has a few words that allow you to work with files. Since 4tH is a scripting language, its capabilities are limited. But you will find that you can perform most common operations.

One of the limitations is that you can have a limited number of open files, but it will do in most situations.

Opening a output-file is pretty simple. Just throw the address and length of a filename and a file access mode on the stack and execute the word 'OPEN'. The value 'OPEN' returns is a simple number which bears little significance. However, you have to save it to a variable or value, for you will need it later. We'd like to use values for storing file pointers, so we created the word 'FILE'. 'FILE' simply creates a value and initializes it, so if you use it prematurely 4tH will issue an error message.

```
file myfile
s" outfile.dat" output open error?   ( a1 n1 fam -- h f)
abort" File could not be opened"     ( h)
to myfile                            ( --)
```

'OUTPUT' is a file access mode and will open a file for writing. 'OPEN' leaves a value on the stack. If it equals '(ERROR)', something was not quite right. If not, the file was

specifier	output	example	printf.4th	sprintf.4th	fsprintf.4th
d	Signed decimal number	392	Y	Y	Y
c	Character	a	Y	Y	Y
s	String of characters	sample	Y	Y	Y
%	Single % character	%	Y	Y	Y
ld	Signed decimal double number	7235	N	N	Y
u	Unsigned decimal number	7235	N	N	Y
o	Signed octal number	610	N	N	Y
x	Signed hexadecimal number	7fa	N	N	Y
f	Decimal floating point, lowercase	392.65	N	N	Y
e	Scientific notation (mantissa/exponent)	3,93E+06	N	N	Y
g	Use the shortest representation: %e or %f	392.65	N	N	Y

Table 9.4: List of supported `printf()` specifiers

successfully opened. 'FILE' is nothing but an initialized value, so you can assign it with 'TO'. 'ERROR?' leaves the handle intact, but leaves an additional true flag if an error occurred, which makes it much easier to evaluate.

The syntax for opening an input file is the same, except for the read-flag 'INPUT' of course:

```
file myotherfile
s" infile.dat" input open error?
abort" File could not be opened"
to myotherfile
```

9.13 Reading and writing from/to a file

There are no special words to read from or write to a file. You can use all the words you used for keyboard-input and screen-output.

But if you open a file and do some I/O you will notice nothing has changed. Of course not. You should be able to determine whether you write to a file or to the screen. There are special words to do just that:

```
file OutFile                  \ file variable
s" outfile.dat" output open error?
abort" File could not be opened"
to OutFile                    \ open the file

OutFile use                   \ write to file
." This is written to disk" cr
stdout use                    \ write to screen
." This is written to screen" cr
```

After you've opened the file, the program will still write to the screen. When 'USE' executes, all output will be redirected to the file. When 'USE' executes again, but this time with the 'OUTPUT' flag, all output will go to the screen again, but the output-file will not be closed! Both words take the same read/write-flags as 'OPEN'.

You can call 'USE' again and again, without closing or opening any files. Here is an example using an input-file:

```
file OutFile
s" outfile.dat" output open error?
abort" File could not be opened"
to OutFile                    \ open output file

OutFile use                   \ write to file
." This is written to disk" cr
stdout use                    \ write to screen
." This is written to screen" cr
OutFile close                 \ close file

s" outfile.dat" input open error?
abort" File could not be opened"
to OutFile

OutFile use                   \ read from disk
pad dup 32 accept             \ read 32 characters
type                          \ write string to screen
stdin use                     \ read from keyboard
OutFile close                 \ close file
```

The output of this program is:

```
This is written to screen
This is written to disk
```

Note that files are always opened in binary mode. If you're a Microsoft user and you worry about your text files, don't. 4tH is much smarter than that as you will learn later on.

9.14 Closing a file

There is usually no need to close any files. When you quit the program all files are closed. It seems like there is no need at all to close files manually, but that is a mistake.

If you want to open a file for reading to which you've just written, you will find it doesn't work. Of course, you can open a file only once.

No, there is a word which closes either the input- or the output-file, using the same read/write-flags. You've already seen it, it is called 'CLOSE'. When you close an active file, the input (or output) is redirected to the keyboard (or screen).

9.15 Writing text-files

Writing text to a file is just as easy as writing text to screen. Open the file, redirect the output, and write like you would write to the screen:

```
file OutFile                  \ value for file
s" outfile.dat"               \ put the filename on the stack
output                        \ add the modifier
open error?                   \ open the file
abort" File could not be opened"
to OutFile

OutFile use                   \ write to file
." This is written to disk" cr
```

That's all! Note that if you execute your program on a Microsoft Operating System, it will write a Microsoft text file. If you do so on a Unix Operating System, it will write a Unix text file. If you want to override that you'll have to issue the end-of-line sequences yourself using 'EMIT'.

9.16 Reading text-files

Reading text-files is pretty straightforward. You don't even have to open a file in text-mode contrary to other languages. Just open the file and call 'REFILL' until it signals end-of-file (EOF):

```
\ Example program. It reads a file line by line
\ and prints it to the screen.

file InFile
s" readln.4th" input open error?
abort" Could not open file"   \ open file
to InFile                     \ save handle
InFile use                    \ read from file

begin
  refill                      \ read a line
while                         \ while EOF not found
  0 parse-word                \ parse the entire line
  type                        \ print it
  cr                          \ terminate line
repeat                        \ read next line
```

You will find that if you run this program, it will print itself to the screen.

'REFILL' will return a non-zero value if EOF was *not* detected. By using the word '0=' you can invert this value. Finally, it will read Unix ASCII-files as well as DOS ASCII-files, no matter where your program is executed.

9.17 Reading long lines

The TIB is only /TIB characters long. If you read a line that is longer than that, only /TIB - 1 characters are read. The rest of the line is read when you invoke 'REFILL' again. Although you don't lose any information that way, it might not be what you want. Fortunately, you can define your own TIB:

```
2048 constant /mytib          \ length of your TIB
/mytib string mytib           \ define your own TIB

mytib /mytib source!          \ tell the system about your TIB
```

The next time you invoke 'REFILL', it will use your TIB instead of the system TIB, so it will now read lines up to 2047 characters. 'SOURCE!' takes an address/count pair and makes it the current TIB. So if you want to use the system TIB again you issue:

```
tib /tib source!
```

And if you have forgotten which TIB you're using try this:

```
source . . cr
```

'SOURCE' will return the address/count pair of the TIB you're currently using. In fact, this definition does absolutely *nothing*:

```
: doesnothing source source! ;
```

For the simple reason that it reassigns the TIB it is already using.

9.18 Reading binary files

If you process binary files, you won't get far reading it line by line. You want to read chunks of data. 4tH can do that too by using 'ACCEPT'. You feel there must be a catch, since 'ACCEPT' terminates strings automatically. Well, there isn't. When 'ACCEPT' does not read from the keyboard, it won't add that extra byte.

Reading blocks of data usually means defining buffers. If maintainability is an issue, define a constant for the sizes of these buffers. You cannot only use this constants when defining buffers, but also when calling 'ACCEPT'.

Furthermore, 'ACCEPT' returns the number of characters actually read. If this value is compared to the number of characters we actually wanted to read, we can determine whether a reading error or EOF occurred:

```
1024 constant bufsize         \ actual buffersize
bufsize string buffer         \ define buffer
file InFile                   \ value for file
                              \ open input file
s" infile.dat" input open error?
abort" File could not be opened"

to InFile                     \ save handle
InFile use                    \ redirect input

begin                         \ using bufsize
  bufsize                     ( n1)
  buffer over                 ( n1 a n1)
  accept                      ( n1 n2)
  <>                          ( f)  \ make EOF flag
until                         \ until EOF
```

Note that "BUFFER" is actually not a string, but a chunk of memory. But since a character in 4tH takes up a single address-unit (=byte), raw chunks of memory are allocated in the Character Segment. This is not an uncommon practice in both Forth and C.

9.19 Writing binary files

Writing binary files is very easy. Of course you need a buffer, like we discussed in the previous section. The program is not much different than the previous one:

```
1024 constant bufsize         \ actual buffersize
bufsize string buffer         \ define buffer
file OutFile                  \ value for file

buffer bufsize char H fill    \ fill the buffer
                              \ open output file
s" infile.dat" output open error?
abort" File could not be opened"

to Outfile                    \ save handle
OutFile use                   \ redirect input
buffer bufsize type           \ write to file
```

This will write 1024 "H"s to "infile.dat". The actual command that does all writing is 'TYPE'. The word 'TYPE' does not return anything. You can be assured that everything was alright, since if it wasn't, 4tH would have caught the error itself.

9.20 Reading and writing block files

Block files are a special kind of files used by Forth compilers. In the old days Forth controlled the *entire* computer and directly communicated with all peripherals, including disks. To Forth, a disk is just a bunch of numbered blocks. Each block is divided into 16 lines of *exactly* 64 characters. A block file simply mimics that layout.

Before we can begin, you need to create a block file. Well, that's easy:

```
include lib/ansblock.4th
4 s" blocks.scr" create-blockfile
```

This creates a file with four blocks. Then we have to tell 4tH which file to use:

```
s" blocks.scr" open-blockfile
```

Note that apart from creating the file, we haven't performed any I/O yet. First, we have to request a block. When a block is requested, its contents are transferred to a memory buffer. You can manipulate this buffer any way you want with the standard words. If you request another block its contents are transferred to the buffer too, overwriting whatever is there. All changes you have made are lost, *unless* you have flagged the block as dirty, which means its contents are different from the block on disk. If a block is dirty, it is written to disk *before* the next block is read. 'CLEAR' is a special word, assigning an empty buffer to a block without reading it first. The buffer is BLANKed. 'UPDATE' will flag the buffer as dirty. 'FLUSH' writes the dirty buffer to disk and unassigns the buffer. So, first we clear block 0:

```
0 clear
```

Then we clear block 1, copy a string to it and flag it as dirty:

```
1 clear
s" Hello world!" >r 1 buffer r@ cmove update
```

'BUFFER' returns the address of the buffer assigned to that block[2]. If the buffer is dirty, it is FLUSHed before assigning a buffer to that block. We can also write the dirty buffer to disk, without unassigning it:

```
save-buffers
```

Note that the buffer is not dirty anymore, since it has been synchronized. Let's write something to another block:

```
s" Goodbye cruel world!" 0 block swap cmove update
```

It is always a nice game to figure out what will happen now. The current block is block 1. Since we haven't UPDATEd it since 'SAVE-BUFFERS', it is clean. That means that 4tH won't perform a write. Since block 0 isn't current, 'BLOCK' reads it into the buffer. The 'UPDATE' will flag the buffer as dirty.

[2]That is particularly handy if your implementation can handle multiple buffers. In this implementation we have only one buffer, so we always return the same address.

```
1 block r> type cr
```

This is fun! The current block is block 0. It is dirty, so it is written to disk. Since block 1 isn't current, it is read into the buffer. You catch my drift? If you want to print the contents of a block, you can use 'LIST'. Of course, 'LIST' uses 'BLOCK' and applies to the same rules:

```
0 list
." This block has been listed: " scr ? cr
```

'SCR' is a variable containing the last screen LISTed. Note that is not the same thing as the current block! Finally, we can discard all our changes:

```
empty-buffers
```

'EMPTY-BUFFERS' does not perform any I/O nor does it change the contents of the buffer. It just unassigns the buffer and flags it as clean. Note that you don't *have* to close a block file since all I/O is block-oriented. Just don't forget to "FLUSH". You can use different block files within the same program, but you'll have to close the current block file - since it performs "FLUSH" as well and it's portable:

```
close-blockfile
```

It is also important to know that ANS-Forth doesn't define any error conditions, so if anything happens all the poor thing can do is throw an exception[3].

Note that the standard block file library implements the absolute bare minimum required by ANS-Forth. The most important restriction is that is has just one single buffer. There is also a libary that is able to handle *thousands* of buffers very efficiently. Standard, it will allocate 16 buffers. If you want a different number of buffers, you'll have to define a constant before including it:

```
32 constant #blocks           \ allocate 32 block buffers
include lib/multiblk.4th
400 s" blocks.scr" create-blockfile
```

It is a drop-in replacement, so you won't have to change your code too much.

9.21 Parsing textfiles

As we've already seen, it is very easy to enter a line using 'REFILL' and parse it. You can also use 'REFILL' to read lines from a text-file. It is quite similar to reading lines from the keyboard, except that you have to open a file first. This little program prints all the words of a textfile on a new line:

```
file InFile                   \ value for file
s" file.txt" input open error?
abort" File could not be opened"
                              \ open the file
to InFile                     \ save handle
InFile use                    \ redirect input to file
```

[3] See section 11.5.

```
begin
  refill                        \ get a line from file
while                           \ check if EOF
  begin
    bl parse-word               \ if not, parse line
    dup 0<>                     \ check if zero length
  while
    type cr                     \ if not, print word
  repeat                        \ parse next word
  drop drop                     \ drop address/count
repeat                          \ get next line
```

Now that flag left by 'REFILL' makes sense! If it is zero, we have reached the end of the line. Note that you don't have to open a file in text-mode and both Microsoft ASCII and Unix ASCII files are supported.

9.22 Parsing binary files

And what about binary files, like classic Forth blockfiles? Well, you could use 'REFILL' in that context too, but it would probably break up words since it can't find an end-of-line marker and its buffer is smaller than 1024 characters. Does that mean it can't be done? No! But 'REFILL' makes it easier for you, because it handles a few tasks automatically.

First, it has its own buffer (TIB). When you're not using 'REFILL' you have to define one yourself. Second, it terminates the string for you. You don't want 'PARSE-WORD' to wander into new territory, do you? Third, it sets '>IN' for you every time its receives new input. You have to take care of that one too.

Never heard of '>IN'? Well, the only way for 'PARSE-WORD' to know on what position the previous scan ended is to store that information into a variable. This variable is called '>IN'.

Not all internal 4tH variables are accessible, mostly because we can't imagine what use they could have to you. Some variables are just better left alone. But '>IN' is available for some very obvious reason: you can reset it and make 'PARSE-WORD' work for you. Note that for '>IN' to work, you have to make the buffer the parsing area by using 'SOURCE!'

The following program will read the first screen of a block-file for you and print out all the words. You will see that all spaces are eliminated and every word is printed on a new line, just the behavior you would expect from 'PARSE-WORD'.

```
1025 constant /buffer           \ screensize + terminator
/buffer 1- constant c/scr       \ size of the block
file InFile                     \ value for file

/buffer string buffer           \ 1: our own buffer

: openfile                      \ open the block file
  s" romans.blk" input open error?
  abort" Cannot open file"
  to InFile                     \ save handle
  InFile use                    \ read from file
;

: readfile                      \ fill the buffer
  buffer c/scr 2dup             \ address and count
  bl fill                       \ clear the buffer
  accept drop                   \ fill the buffer
  InFile close                  \ close the file
```

```
;

: initparse                    \ configures parsing
  0 buffer c/scr chars + c!    \ 2: terminate screen
  buffer /buffer source!       \ 3: make buffer the parse area
  0 >in !                      \ 4: reset >IN
;

: parseblock
  begin
    bl parse-word              \ get word
    dup 0<>                    \ length zero?
  while
    type cr                    \ if so, print it
  repeat
  2drop                        \ else drop addr/cnt
  ." End of block" cr          \ signal "End of block"
;

: parsefile                    \ do it all
  openfile                     \ open the file
  readfile                     \ read it
  initparse                    \ set up parsing
  parseblock                   \ parse it
;

parsefile
```

Note there is no need to reset '>IN' if you use 'REFILL', since it will be reset *automatically*. In this case, if you want to parse another block, you will have to reset '>IN' again.

9.23 Parsing comma-delimited files

'PARSE-WORD' is a powerful and very useful word, but it is less than useful when parsing comma-delimited files. Why? Well, because 'PARSE-WORD' skips leading delimiters. So when you have a file like this it doesn't work:

```
FIRSTNAME,NAME,EMAIL,TELEPHONE,HOMEPAGE,FAX
Hans,Bezemer,The.Beez.speaks@gmail.com,,http://hansoft.come.to,
```

Again, 'PARSE-WORD' skips leading delimiters, so instead of an empty string we get the homepage when we're trying to read the (non-existant) telephone number. Fortunately, we got a word like 'PARSE'. 'PARSE' also takes a delimiter from the stack, just like 'PARSE-WORD', but it acts on leading delimiters. Take a look at this program:

```
file OutFile                   \ value for output file
file InFile                    \ value for input file

: WriteCommaFile               ( --)
  s" address.csv" output open error?
  abort" Could not write CSV file"
  to OutFile                   \ save handle
  OutFile use                  \ redirect output to file
  ." FIRSTNAME,NAME,EMAIL,TELEPHONE,HOMEPAGE,FAX" cr
  ." Hans,Bezemer,,,http://hansoft.come.to," cr
  OutFile close                \ close file
  stdout use                   \ redirect output to screen
;
                               \ reads a field
: field> [char] , parse ;      ( -- a n)
```

```
: ProcessLine                ( --)
  refill                     \ get line
  0= abort" Read error"
  field> type cr             \ parse first name
  field> type cr             \ parse name
  field> type cr             \ parse email
  field> type cr             \ parse telephone
  field> type cr             \ parse homepage
  field> type cr cr          \ parse fax
;

: ReadCommaFile              ( --)
  s" address.csv" input open error?
  abort" Could not read CSV file"
  to InFile                  \ save handle
  InFile use                 \ redirect input to file
  ." _Headerline_" cr        \ this is the headerline
  ProcessLine                \ now process headerline
  ." _First record_" cr      \ this is the first record
  ProcessLine                \ now process first record
  InFile close               \ close file
;

WriteCommaFile               \ write the CSV file
ReadCommaFile                \ read the CSV file
```

With "WriteCommaFile" we write a simple comma-delimited file to disk. We got to read something, don't we? Then we read the file we've just written with "ReadCommaFile". "ProcessLine" does the actual job. Since we have six fields we use 'PARSE' six times - that is: each time we call "FIELD>". We cannot do this with a loop. Why not?

Well, 'PARSE' not only returns a NULL-string when we've reached the end of a line, but also when a field is empty. So we've got to know how many fields we actually want to read. Of course, you could parse the headerline with 'PARSE-WORD' to find that out, but you already know how to do this.

9.24 Parsing fixed-width text files

A fixed-width text file is a file consisting of records with a fixed-width fields. They are not as common anymore as they used to be in the mainframe era, but you still find them every now and them. They look like this:

```
New Jersey          NJTrenton       8958013 22592  12
Texas               TXAustin        27469114695660 36
Washington          WAOlympia       7170351 184661 10
```

And yes, you can process them with 4tH. First thing you need to do is to include a small library:

```
include lib/fixedfld.4th
```

Next, you need to create a small array containing all the field lengths, first one first, and terminate it with a zero:

```
create states-fields 20 , 2 , 14 , 8 , 7 , 2 , 0 ,
```

Note there are fixed-width files with line termination. You can read those with 'REFILL'. If they are not line terminated, don't dispair[4]. Let's say a record is 64 bytes, then you can read a record at the time like this:

```
64 constant /record           \ length of record
/record buffer: record        \ define a buffer
                              \ use like REFILL
: read-record record /record accept /record = ;
```

But in this example we will assume our file is line terminated. The next thing to do is to integrate the library into your reading routine. Since we use a line-terminated file, 'REFILL' will do:

```
/layout array states-layout   \ this is the layout variable
                              \ enhance REFILL
: !refill states-fields tib states-layout fields! refill ;
```

The "FIELDS!" word takes three parameters. First, the array with the field lengths. Second, your read buffer. If we would have used 'ACCEPT' from our previous example, we would have used the "RECORD" buffer. Here we use 'REFILL', so we use the standard buffer, 'TIB'.

Finally the so-called "layout variable". This is variable you have to define for each fixed-width file you want to process with this library, so if you parse two different files at the same time, you have to define two "layout variables".

Now the only thing you have to do is to use "!REFILL" instead of 'REFILL' and we're all set. "FIELDS!" will make sure the library is properly set up each time we read a fresh line.

When we construct our "FIELD>" routine[5] we do something similar. "NEXT-FIELD" will ensure you get the next field, provided you feed it the proper "layout variable", of course. Note that "NEXT-FIELD" returns the *entire* field, which is probably padded. Let's assume it's padded with spaces, which we can easily remove with '-TRAILING':

```
: field> states-layout next-field -trailing ;
```

Since the stack diagram of this "FIELD>" is identical to the one in section section 9.23, the entire thing becomes almost completely transparent:

```
: ProcessLine                 ( --)
  !refill                     \ get record
  0= abort" Read error"
  field> type cr
  field> type cr
  field> type cr
  field> type cr
  field> type cr
  field> type cr cr
;
```

Yes, it takes some setting up, but in the end reading a fixed-width text file is not much harder than reading an ordinary .CSV file.

[4] See also section 9.22.
[5] See section 9.23.

9.25 Advanced parsing

Let's think of something difficult to parse, e.g.:

```
;;;;;;;;;.........Can you parse me.
;;;;„„,And me too, huh?
```

If we would use 'PARSE' we would have to know how many semicolons to skip and there is a different number of them on each line. If we would use 'PARSE-WORD' we'd lose all the semicolons, but the parsed string would have all these nasty leading dots.

Even worse, if we were able to skip the semicolons and use 'PARSE-WORD' with the leading delimiter we'd get "Can you parse me" and "And me too" instead of "Can you parse me." and "And me too, huh?". What can we do?

Fortunately, 4tH doesn't really know about 'PARSE-WORD' but translates it into a sequence of words[6]. We can also use them directly. 'OMIT' is very handy. It doesn't actually do anything, it just skips leading delimiters and sets '>IN' accordingly. It takes an ASCII value from the stack as its delimiter. This will correctly parse the first line:

```
char ; omit                  \ omit the semicolon
char . omit                  \ omit the dot
0 parse                      \ parse the remainder of the line
```

This will correctly parse the second line:

```
char ; omit                  \ omit the semicolon
char , omit                  \ omit the comma
0 parse                      \ parse the remainder of the string
```

Please note that this are special 4tH words! Unfortunately you cannot port this to ANS-Forth, where only a limited version of 'PARSE-WORD' and 'PARSE' are available.

9.26 Appending to existing files

You can use a so-called modifier to signal 4tH, it shouldn't overwrite the file it opens, but append to it:

```
file OutFile                 \ value for output file

s" outfile.dat" output open error?
abort" Cannot open file"     \ open the file
to OutFile                   \ save handle
OutFile use                  \ now write to disk
10 0 do i . loop             \ write 0 to 9
OutFile close                \ close the file

s" outfile.dat" output append + open error?
abort" Cannot open file"     \ reopen in append mode
to OutFile                   \ save handle
OutFile use                  \ now write to disk
20 10 do i . loop            \ write 10 to 19
OutFile close                \ close the file
```

Take a look at the contents of this file after you've run the program and you'll find it contains the number 0 to 19.

[6] If you're really curious, 'PARSE-WORD' is equivalent to 'DUP OMIT PARSE'.

9.27 Using pipes

If you're using Windows 95 OSR2 (and up), Windows NT (and up), OS/X, Linux or another Unix system you're in for a treat! With Unix you can do neat tricks like this:

```
ls | mail root
```

Which means you can redirect the output of 'ls' to 'mail', so in effect you send an email to root with the contents of your current working directory. Yes, 4tH can do this too, but you can do even more. You can start 'ls' and read its output line by line as if it were a file. You can also start 'mail' and write the output of a 4tH program to it. We do that by opening a *pipe* to a program.

If you've ever written a program using C, you know this is a bit cumbersome, since you've got to use special functions to use pipes. In 4tH you don't. Just let 4tH know it's a pipe that you're opening and not a file:

```
file InFile                  \ value for input file
file OutFile                 \ value for output file

s" ls" input pipe + open error?
s" mail hans" output pipe + open error? rot or
abort" Cannot open pipe"
to OutFile to InFile
```

The only thing you have to do to signal 4tH that you're using a pipe is add the word 'PIPE', just like 'APPEND'. The filename is replaced by the command you want to execute. That's all. If one of the pipes in this program fails, the program aborts.

```
InFile use OutFile use
." These are the contents of my current working directory:"
cr cr
```

Now we can treat our pipes just as if they are ordinary files. We redirect input and output and write a nice header to our email. Now we can start to process the output of 'ls':

```
begin
  refill
while
  0 parse-word type cr
repeat
```

Note that we don't have to signal that we're reading the pipe to 'ls' as a text file. We just read it line by line until 'REFILL' returns zero. Then we can parse the line and 'TYPE' it to 'mail'.

```
InFile close OutFile close
```

Of course you don't *have to* close the pipe, but it won't harm when you don't. 4tH knows what to do. After executing this program, Hans will receive this email:

```
From: Hans Bezemer <The.Beez.speaks@gmail.com>
Message-Id: <200202252017.VAA00712@gmail.com>
To: hans@localhost.org
Status: RO
```

```
These are the contents of my current working directory:

4th.c
4thc.c
4thd.c
4thg.c
4thx.c
```

Well, that wasn't too hard, was it?

9.28 Opening a file in read/write mode

For special purposes you might want to open a file in read/write mode. That's quite easy:

```
file InOutFile                \ value for output file

s" outfile.dat" output input + open error?
abort" Cannot open file"      \ open the file
to InOutFile                  \ save handle
```

Just add both together like you adding a modifier. Note that once you 'USE' this file, you're both reading *and* writing to this file. Furthermore, the file has to exist otherwise you get an error. If you want to write to a new file, you first have to open it in write mode:

```
file InOutFile                \ value for output file
                              \ create a new file
s" outfile.dat" output open close
s" outfile.dat" output input + open error?
abort" Cannot open file"      \ open the file
to InOutFile                  \ save handle
```

9.29 Using random access files

Upto now we've always accessed a file sequentially, but it is also possible to use random access files. Two words are crucial here, 'SEEK' and 'TELL'. 'SEEK' will seek for the desired file position and 'TELL' will tell you that you're there. It is as simple as that..!

Let's take a look at this example. We've got a block-file called "Messages.scr" with the following contents:

```
Scr # 0
  0 (0) No errors
  1 (1) Out of memory
  2 (2) Bad object
  3 (3) Stack overflow
  4 (4) Stack empty
  5 (5) Return stack overflow
  6 (6) Return stack empty
  7 (7) Bad string
  8 (8) Bad variable
  9 (9) Bad address
 10 (A) Divide by zero
 11 (B) Bad token
 12 (C) Bad radix
 13 (D) Undefined name
 14 (E) I/O error
 15 (F) Assertion failed
```

First, let's define a word that reads a message and then displays it:

```
: next-msg pad dup 64 accept -trailing type cr ;
```

Since this is a simple program we can safely use the 'PAD' to store our messages. Every message has the length of a standard block-file line, which is 64 characters. Trailing spaces are stripped by '-TRAILING'. Now we need a word that tells us what our file position is:

```
: tell-msg cr ." Current position: " dup tell . cr ;
```

'TELL' needs a file pointer and leaves the current position of that file pointer on the stack. This word assumes that the top of the stack contains a valid file pointer. Finally we need a word that sets the file position:

```
: seek-msg over seek abort" Seek failed" ;
```

'SEEK' needs a file position and a file pointer. If it returns false, it was successful; if it returns true there was an error. This word assumes that the top of the stack contains a valid file pointer. We're ready now, let's play. First we open the file and use it:

```
s" Messages.scr" input open dup use
```

This leaves a file pointer on the top of the stack, assuming everything went OK. Now let's read some messages:

```
next-msg next-msg next-msg tell-msg
```

You'll see these messages appear on the screen:

```
(0) No errors
(1) Out of memory
(2) Bad object

Current position: 192
```

After reading three messages we've obviously reached position 192 in the file. That makes sense, since 3 lines of 64 characters makes 192 characters in total. Let's see what 'SEEK' does:

```
0 seek-msg tell-msg next-msg
```

This should take us back to the very beginning of the file, as if we've freshly opened it. And yes, it does:

```
Current position: 0
(0) No errors
```

After executing 'SEEK', 'TELL' confirms that we've actually returned to the very beginning of the file. Reading the next message reconfirms that again. When you feed 'SEEK' positive values, it always starts seeking from the *beginning* of the file. When you feed 'SEEK' negative values, it seeks from the *end* of the file. So this one takes you to the last line:

```
-64 seek-msg tell-msg next-msg
```

On screen it looks like this:

```
Current position: 960
(F) Assertion failed
```

Finally, we clean up the mess we made:

```
close
```

This will consume the file pointer we left on the stack and close the file. Note that 'SEEK' and 'TELL' come with a few restrictions. Pipes are out of the question and so are the standard streams 'STDIN' and 'STDOUT'. Apart from that you can pretty much do with them what you want.

Another tip: if you use '(ERROR)' as an offset for 'SEEK', it will take you to the very end of the file, since 'MAX-N' won't do the trick.

9.30 The layout of the I/O system

You're probably quite confident manipulating files now, so I guess it is time to offer you a view under the hood. 4tH has two channels, an input channel and an output channel. All words read from the input channel or write to the output channel. At startup, the input *channel* is connected a *stream* that reads from the keyboard ('STDIN') and the output *channel* is connected to a *stream* that writes to the screen ('STDOUT').

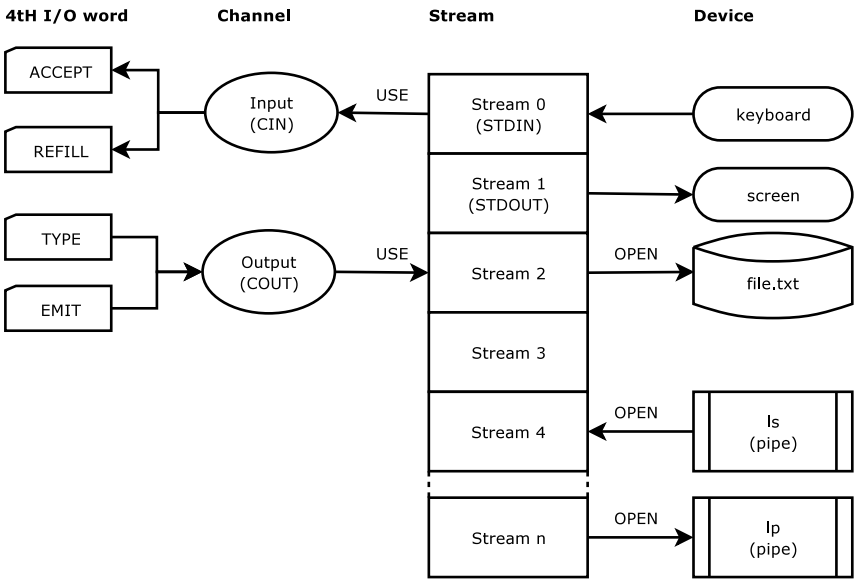

Figure 9.2: The 4tH I/O system

With 'OPEN' you can open *additional* streams, which are connected to a file or a pipe. The return value of 'OPEN' points to the stream that was opened. There are few words that directly handle streams, 'USE', 'CLOSE', 'TELL' and 'SEEK' being the exceptions. 'USE' attaches a stream to one or both channels, which results in redirecting all in- and/or

output to that stream. E.g. if a file is opened in read/write mode using 'OPEN', a stream is returned. If we 'USE' that stream, both the input and the output channel are connected to that stream. If it had only been opened in read mode, only the input channel would have been connected to the stream.

'CLOSE' closes a stream, even if it is still attached to a channel. If that is the case, the appropriate default streams ('STDIN', 'STDOUT') are reattached. We can find out which streams are currently used by 'CIN' and 'COUT'. 'CIN' returns the stream that is currently attached to the input channel, 'COUT' returns the stream that is currently attached to the output channel.

9.31 Using a printer

How you access a printer depends on the operating system you're working on. That is not a flaw of 4tH, you will encounter this problem with every programming language. If you're working with MS-DOS or MS-Windows it is quite basic:

```
file printer                  \ value for printer

s" lpt1" output open error?
if
  drop
else
  to printer
  printer use
  ." This will be printed." cr
  stdout use
then
```

Just open the port as a file and print to it. Unix isn't that different, but instead of opening a file, you open a pipe:

```
file printer                  \ value for printer

s" lp" output pipe + open error?
if
  drop
else
  to printer
  printer use
  ." This will be printed." cr
  stdout use
then
```

If you're using a different Operating System, you may have to check your manual.

9.32 The layout of the Character Segment

The final topic of this chapter again. You already know that 4tH checks whether an operation is still within the Character Segment. However, sometimes you want to check this yourself.

You already know how you can obtain the size of TIB and PAD. Yes, you can using '/TIB' and '/PAD'. But TIB and PAD have their addresses too. And when you query them, you will find that PAD comes after TIB:

```
." Address of TIB: " tib . cr
." Size of TIB : " /tib  . cr
." Address of PAD: " pad . cr
." Size of PAD : " /pad  . cr
```

And beyond PAD, what is there? Well, allocated memory of course. Things you defined using 'STRING'. There are two words which can give you information about allocated memory. First, 'LO'. 'LO' gives you the lowest address of allocated memory. Second, 'HI'. 'HI' gives you the highest valid address of the Character Segment. That means that:

```
0 hi c!
```

Is always valid and:

```
0 hi char+ c!
```

Is always invalid. It you try it, 4tH will stop executing the program with an error-message. 'LO' and 'HI' are addresses. Addresses are just numbers, so you can print and compare them. E.g.

```
hi char+ lo - . cr
```

Will print how much memory as allocated to your strings. And:

```
lo hi >
```

Will indicate whether you allocated any memory at all. If 'LO' is greater than 'HI', you didn't. If 'HI' is greater or equal to 'LO' you did. Experiment a bit with the knowledge you obtained in this chapter and continue with the next one where we will go much deeper into the secrets of the Integer Segment and Code Segment.

Chapter 10

Integer Segment and Code Segment

10.1 The Code Segment

It is known by designers of microprocessors that a processor can run much faster when every instruction has the same length. In fact, 4tH has his own virtual microprocessor. The compiler is nothing more than an assembler and the interpreter nothing more than an emulator on top of the real microprocessor.

In order to speed up 4tH, all instructions have the same length. They consist of a token (which is the real instruction) and an argument. The argument is a value that gives meaning to the instruction, e.g. the 'LITERAL' token means that a number is compiled here. The argument is the actual number.

Some instructions wouldn't need an argument, but for speeds sake, they have: it is always zero. Isn't that a lot overhead? Not really. Half the instructions in an actual program need an argument. Decoding a more elaborate scheme would need more processor time and more programming. So in the end, it would make hardly any difference. Except for the speed.

A token with its argument is called a word. And the Code Segment is one large array of words. Each of these words has an address and can be accessed by the word '@C'. In fact, '@C' throws the argument on the stack. Where have we seen '@C' before?

Yes, when fetching from an array of constants. These arrays are compiled into the Code Segment. How come that 4tH isn't confused by these arrays? Because they have the token 'NOOP', which does absolutely nothing.

10.2 The address of a colon-definition

You can get the address of a colon definition by using the word "' (tick):

```
: add + ;                       \ a colon definition
' add . cr                      \ display address
```

Very nice, but what good is it for? Well, first of all the construction "' ADD" throws the address of "ADD" on the stack. In fact, it is a literal expression. You can assign it to a variable, define a constant for it, or compile it into an array of constants:

```
' add constant add-address

variable addr
' add addr !

create addresses ' add ,
```

Are you with us so far? If we would simply write "ADD", "ADD" would be executed right away and no value would be left on the stack. Tick forces 4tH to throw the address of "ADD" on the stack instead of executing "ADD".

Note this only works for your own colon-definitions. It doesn't work for 4tHs built-in words. If you try to, you'll get an error-message. What you can actually do with it, we will show you in the next section.

10.3 Vectored execution

This is a thing that can be terribly difficult in other languages, but is extremely easy in Forth. Maybe you've ever seen a BASIC program like this:

```
10 LET A=40
20 GOSUB A
30 END
40 PRINT "Hello"
50 RETURN
60 PRINT "Goodbye"
70 RETURN
```

If you execute this program, it will print "Hello". If you change variable "A" to "60", it will print "Goodbye". In fact, the mere expression "GOSUB A" can do two different things. In 4tH you can do this much more comfortable:

```
: goodbye ." Goodbye" cr ;
: hello ." Hello" cr ;

variable a

: greet a @ execute ;

' hello a !
greet

' goodbye a !
greet
```

What are we doing here? First, we define a few colon-definitions, called "HELLO" and "GOODBYE". Second, we define a variable called "A". Third, we define another colon-definition which fetches the value of "A" and executes it by calling 'EXECUTE'. Then, we get the address of "HELLO" (by using "' HELLO") and assign it to "A" (by using "A !"). Finally, we execute "GREET" and it says "Hello".

It seems as if "GREET" is simply an alias for "HELLO", but if it were it would print "Hello" throughout the program. However, the second time we execute "GREET", it prints "Goodbye". That is because we assigned the address of "GOODBYE" to "A".

The trick behind this all is 'EXECUTE'. 'EXECUTE' takes the address of e.g. "HELLO" from the stack and calls it. In fact, the expression:

```
hello
```

Is equivalent to:

```
' hello execute
```

This can be extremely useful as we will see in the next chapter when we build a full-fledged interpreter. We'll give you a little hint:

```
create subs ' hello , ' goodbye ,
```

Does this give you any ideas?

10.4 Execution tokens

The address of a colon definition is called an "execution token". It's much like a token in a subway: you insert a token into the appropriate slot and the turnstile opens. Here you pass an execution token to 'EXECUTE' and the definition is executed. Of course, you can define an execution token this way:

```
: hello ." Hello world!" ;
' hello constant hello-xt
```

But you can also define it more directly:

```
:token hello-xt ." Hello world!" ;
```

It will behave in exactly the same way our previous definition of "HELLO-XT" did. There is still another way to define execution tokens. We'll come around that in section 11.7.

10.5 The Integer Segment

Wonder where all these variables are created? Or where that infamous stack really is? I bet you do. Well, when you define a variable, memory is allocated in the Integer Segment. When you define another one, space is allocated after the first variable. That means that if you go beyond the boundaries of the first variable, you'll end up in the space allocated to the second variable.

After the second variable there is a void. If you end up there your program will end with an error-message. However, if you define an 'ARRAY', single variable is created with a number of additional cells. You can only access these additional by referring to the array itself.

And what about the space before the first variable? There are other variables and they are not defined by you. Well, take a look at figure 10.1.

Lower memory is at the bottom. The user variables are the variables you defined yourself. The application variables can differ from host program to host program. Refer to your documentation on that subject. You have already seen the 4tH variables, which are 'BASE' and '>IN'. There are also variables you cannot access. These variables are hidden and only used by the system. All these variables are located in the Variable Area.

There is also a Stack Area, which contain the datastack and the returnstack. If you enter a number like "5", it is thrown on the datastack. Most words in 4tH take or put numbers on the datastack. It is very heavily used. We'll come to the returnstack later on.

The datastack and the returnstack share the same memory space. The datastack grows upward and the returnstack downward. If they clash the stack is full and 4tH will issue an error-message.

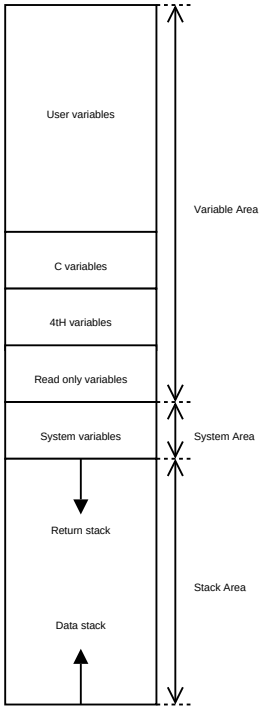

Figure 10.1: Integer segment

10.6 A portable way to access application variables

A host program can add special variables to the 4tH environment. If 4tH is used as a scripting language in e.g. a printer program, the programmer can "send" variables to 4tH. These variables are called "application variables". Do not confuse them with 4tH variables, like 'BASE' or '>IN' which are used internally by 4tH. 4tH doesn't do anything with application variables.

If the creator of the host program provided special names for each of these variables, he will probably have documented them. However, even if he didn't there is another way to access these variables.

They are stored in a predefined array called 'APP' and its values can be fetched like any other array, e.g.:

```
app 1 th @
```

Which fetches the value of the second element in the array. This also enables you to write programs that can be compiled and run under all "standard" versions of 4tH.

10.7 Returning a result to the host program

The 'APP' array can feed values from the host program to yours, but it can't return any. For that you need 'OUT', the third 4tH variable. Returning a value is very easy. Just store it in 'OUT'. Let's assume the host program has send two values to the 'APP' array and you want to return the sum. All you have to do is add them and store the result in 'OUT':

```
app 0 th @ app 1 th @ + out !
```

Nothing to it..

10.8 Using commandline arguments

A host program can also transfer an array of strings to the 4tH environment. Usually, commandline arguments will be transferred this way, although any string array with the correct format can be used. If so, you will probably find it in the documentation of the host program.

If you are familiar with C, the concept is probably quite easy to understand. There are two words, 'ARGN' and 'ARGS'. 'ARGN' will leave the number of commandline arguments on the stack. The commandline arguments itself are numbered from 0 to (ARGN - 1), e.g.

```
argn 0>                     \ test if there are
if                          \ any arguments
  argn 0 do                 \ loop through them
    i args type cr          \ print them
  loop
then
```

First, we test if there are any commandline arguments. Second, if that is the case we loop through them with 'ARGN' as upper limit. Why? Since "ARGN 1- ARGS" is always the last valid commandline argument!

Third, when 'ARGS' executes, it takes a number from the stack as index. Then it leaves the address of the Character Segment (where it is temporarily stored, usually PAD) and its count on the stack.

Using the expression "TYPE CR" we can print that string. Because it is already stored in the Character Segment we can treat it like any other string. Remember, that if you don't save it anywhere else it won't last long!

10.9 The layout of the Variable Area

There are special words that allow you to get information about the layout of the Variable Area. They are called 'VARS', 'APP', 'FIRST' and 'LAST'.

'VARS' is the address of the very first variable. Before that is the Stack Area and other variables you are not allowed to touch. 'APP' is the address of the first application variable. All variables before that are 4tHs own built-in variables. 'FIRST' is the address of the first user-variable, a variable you defined yourself in your 4tH program. 'LAST' is the address of the last accessible variable, so

```
last ?
```

will *never* fail. The first question that will pop in your mind is, what can I do with them. Well, you can use it to see how many variables there are of a certain kind, so you can prevent runtime errors:

```
." number of 4tH variables: "         app vars - . cr
." number of application variables: " first app - . cr
." number of user variables: "        last first - 1+ . cr
```

These tests are possible too:

```
app vars -   0= if ." No 4tH variables" cr then
first app -  0= if ." No application variables" cr then
last first - 1+ 0= if ." No user variables" cr then
```

This is a general test to see whether the address of any variable is within range:

```
dup 0<
dup last >
or
if ." Out of range" cr then
```

You can use this check on numeric arrays too, of course.

10.10 The stacks

The Stack Area contains two stacks. So far we've talked about one stack, which is the Data Stack. The Data Stack is heavily used, e.g. when you execute this code:

```
2 3 + .
```

Only the Data Stack is used. First, "2" is thrown on it. Second, "3" is thrown on it. Third, '+' takes both values from the stack and returns the sum. Fourth, this value is taken from the stack by '.' and displayed. So where do we need the other stack for?

Well, we need it when we want to call a colon-definition. Before execution continues at the colon-definition, it saves the address of the currently executed token in the Code Segment on the other stack, which is called the Return Stack for obvious reasons.

Then execution continues at the colon-definition. Every colon-definition is terminated by ';', which compiles into 'EXIT'. When 'EXIT' is encountered, the address on top of the Return Stack is popped. Execution then continues at that address, which in fact is the place where we came from.

If we would store that address on the Data Stack, things would go wrong, because we can never be sure how many values were on that stack when we called the colon-definition, nor would be know how many there are on that stack when we encounter 'EXIT'. A separate stack takes care of that.

Try and figure out how this algorithm works when we call a colon-definition from a colon-definition and you will see that it works (4tH is proof of that).

It now becomes clear how 'EXECUTE' works. When 'EXECUTE' is called, the address of the colon-definition is on the Data Stack. All 'EXECUTE' does is copy its address on the Return Stack, take the address from the Data Stack and call it. 'EXIT' never knows the difference..

But the Return Stack is used by other words too. Like 'DO' and 'LOOP'. 'DO' takes the limit and the counter from the Data Stack and puts them on the Return Stack. 'LOOP' takes both of them from the Return Stack and compares them. If they don't match, it continues execution after 'DO'. That is one of the reasons that you cannot split a 'DO..'LOOP'.

However, if you call a colon-definition from within a 'DO'..'LOOP' you will see it works: the return address is put on top of the limit and the counter. As long as you keep the Return Stack balanced (which isn't too hard) you can get away with quite a few things as we will see in the following section.

10.11 Saving temporary values

We haven't shown you how the Return Stack works just for the fun of it. Although it is an area that is almost exclusively used by the system you can use it too.

We know we can manipulate the Data Stack only three items deep (using 'ROT'). Most of the time that is more than enough, but sometimes it isn't.

In 4tH there are special words to manipulate stack items in pairs, e.g. '2DUP' (n1 n2 – n1 n2 n1 n2) or '2DROP' (n1 n2 –). Although they are already part of 4tH, we could easily define those two ourselves:

```
: 2dup over over ;
: 2drop drop drop ;
```

You will notice that '2SWAP' (n1 n2 n3 n4 – n3 n4 n1 n2) becomes a lot harder. How can we get this deep? You can use the Return Stack for that..

The word '>R' takes an item from the Data Stack and puts it on the Return Stack. The word 'R>' does it the other way around. It takes the topmost item from the Return Stack and puts it on the Data Stack. Let's try it out:

```
: 2swap ( n1 n2 n3 n4)        \ four items on the stack
  rot   ( n1 n3 n4 n2)        \ rotate the topmost three
  >r    ( n1 n3 n4)           \ n2 is now on the Return Stack
  rot   ( n3 n4 n1)           \ rotate other items
  r>    ( n3 n4 n1 n2)        \ get n2 from the Return Stack
;
```

And why does it work in this colon-definition? Why doesn't the program go haywire? Because the Return Stack is and was perfectly balanced. The only thing we had to do was to get off "n2" before the semi-colon was encountered. Remember, the semi-colon compiles into 'EXIT' and 'EXIT' pops a return-address from the Return Stack. Okay, let me show you the Return Stack effects:

```
: 2swap ( r1)
  rot   ( r1)
  >r    ( r1 n2)
  rot   ( r1 n2)
  r>    ( r1)
;       ( --)
```

Note, these are the Return Stack effects! "R1" is the return-address. And it is there on top on the Return Stack when 'EXIT' is encountered. The general rule is:

> "Clean up your mess inside a colon-definition"

If you save two values on the Return Stack, get them off there before you attempt to leave. If you save three, get three off. And so on. This means you have to be very careful with looping and branching. Otherwise you have a program that works perfectly in one situation and not in another:

```
: this-wont-work ( n1 n2 -- n1 n2)
  >r             ( n1)
  0= if          ( --)
    r>           ( n2)
    dup          ( n2 n2)
  else
    1 2          ( 1 2)
  then
;
```

This program will work perfectly if n1 equals zero. Why? Let's look at the Return Stack effects:

```
: this-wont-work ( r1)
  >r             ( r1 n2)
  0= if          ( r1 n2)
    r>           ( r1)
    dup          ( r1)
  else           ( r1 n2)
    1 2          ( r1 n2)
  then
;
```

You see when it enters the 'ELSE' clause the Return Stack is never cleaned up, so 4tH attempts to return to the wrong address. Avoid this, since this can be very hard bugs to fix.

10.12 The Return Stack and the DO..LOOP

We've already told you that the limit and the counter of a DO..LOOP (or DO..+LOOP) are stored on the Return Stack. But how does this affect saving values in the middle of a loop? Well, this example will make that quite clear:

```
1       ( n)
10 0 do ( n)
  >r    ( --)
  i .   ( --)
  r>    ( n)
loop    ( n)
cr      ( n)
drop    ( --)
```

You might expect that it will show you the value of the counter ten times. In fact, it doesn't. Let's take a look at the Return Stack:

```
1       ( --)
10 0 do ( l c)
  >r    ( l c n)
  i .   ( l c n)
  r>    ( l c)
loop    ( --)
cr      ( --)
drop    ( --)
```

You might have noticed that it prints ten times the number "1". Where does it come from? Usually 'I' prints the value of the counter, which is on top of the Return Stack.

This time it isn't: the number "1" is there. So 'I' thinks that "1" is actually the counter and displays it. Since that value is removed from the Return Stack when 'LOOP' is encountered, it doesn't do much harm.

We see that we can safely store temporary values on the Return Stack inside a DO..LOOP, but we have to clean up the mess, before we encounter 'LOOP'. So, this rule applies here too:

> "Clean up your mess inside a DO..LOOP"

But we still have to be prepared that the word 'I' will not provide the expected result (which is the current value of the counter). In fact, 'I' does simply copy the topmost value on the Return Stack. Which is usually correct, unless you've manipulated the Return Stack yourself.

Note that there are other words beside 'I', which do exactly the same thing: copy the top of the Return Stack. But they are intended to be used outside a DO..LOOP. We'll see an example of that in the following section.

10.13 Other Return Stack manipulations

The Return Stack can avoid some complex stack acrobatics. Stack acrobatics? Well, you know it by now. Sometimes all these values and addresses are just not in proper sequence, so you have to 'SWAP' and 'ROT' a lot until they are.

You can avoid some of these constructions by just moving a single value on the Return Stack. You can return it to the Data Stack when the time is there. Or you can use the top of the Return Stack as a kind of local variable.

No, you don't have to move it around between both stacks all the time and you don't have to use 'I' out of its context. There is a well-established word, which does the same thing: 'R@'. This is an example of the use of 'R@':

```
: delete     ( n )
  >r #lag +  ( a1)
  r@ #lag    ( a1 a2 n2)
  r@ negate  ( a1 a2 n2 n3)
  r# +!      ( a1 a2 n2)
  #lead +    ( a1 a2 n2 a3)
  swap cmove ( a1)
  r> blanks  ( --)
;
```

'R@' copies the top of the Return Stack to the Data Stack. This example is taken from the 4tH-editor. It deletes "n" characters left of the cursor. By putting the number of characters on the Return Stack right away, its value can be fetched by 'R@' without using 'DUP' or 'OVER'. Since it can be fetched at any time, no 'SWAP' or 'ROT' has to come in.

10.14 Altering the flow with the Return Stack

The mere fact that return addresses are kept on the stack means that you can alter the flow of a program. This is hardly ever necessary, but if you're a real hacker you'll try this anyway, so we'd better give you some pointers on how it is done. Let's take a look at this program. Note that we comment on the Return Stack effects:

```
: soup ." soup " ;              ( r1 r2)
: dessert ." dessert " ;        ( r1 r6)
: chicken ." chicken " ;        ( r1 r3 r4)
: rice ." rice " ;              ( r1 r3 r5)
: entree chicken rice ;         ( r1 r3)
: dinner soup entree dessert ;  ( r1)
dinner cr                       ( --)
```

And this is the output:

```
soup chicken rice dessert
```

Before we execute "DINNER" the Return Stack is empty. When we enter "DINNER" the return address to the main program is on the Return Stack (r1).

"DINNER" calls "SOUP". When we enter "SOUP" the return address to "DINNER" is on the Return Stack (r2). When we are done with "SOUP", its return address disappears from the Return Stack and execution continues within "DINNER".

Then "ENTREE" is called, putting another return address on the Return Stack (r3). "ENTREE" on its turn, calls "CHICKEN". Another return address (r4) is put on the Return Stack. Let's take a look on what currently lies on the Return Stack:

-	Top Of Return Stack (TORS)
r4	returns to ENTREE
r3	returns to DINNER
r1	returns to main program

As we already know, ';' compiles an 'EXIT', which takes the TORS and jumps to that address. What if we lose the current TORS? Will the system crash?

Apart from other stack effects (e.g. too few or the wrong data are left on the Data Stack) nothing will go wrong. Unless the colon-definition was called from inside a DO..LOOP, of course. But what DOES happen? The solution is provided by the table: it will jump back to "DINNER" and continue execution from there.

```
: soup ." soup " ;                ( r1 r2)
: dessert ." dessert " ;          ( r1 r6)
: chicken ." chicken " r> drop ;  ( r1 r3 - r4 gets lost!)
: rice ." rice " ;                ( r1 r3 r5)
: entree chicken rice ;           ( r1 r3)
: dinner soup entree dessert ;    ( r1)
dinner cr                         ( --)
```

Since "CHICKEN" gets rid of the return address to "ENTREE", "RICE" is never called. Instead, a jump is made to "DINNER" that assumes that "ENTREE" is done, so it continues with "DESSERT". This is the output:

```
soup chicken dessert
```

Note that this is *not* common practice and we do *not* encourage its use[1]. However, it gives you a pretty good idea how the Return Stack is used by the system.

10.15 Leaving a colon-definition

You can sometimes achieve the very same effect by using the word 'EXIT' on a strategic place. We've already encountered 'EXIT'. It is the actual word that is compiled by ';'.

What you didn't know is that you can compile an 'EXIT' without using a ';'. And it does the very same thing: it pops the return address from the Return Stack and jumps to it. Let's take a look at our slightly modified previous example:

[1] As a matter of fact, it *may* interfere with 4tH's peephole optimizer.

```
: soup ." soup " ;              ( r1 r2)
: dessert ." dessert " ;        ( r1 r6)
: chicken ." chicken " ;        ( r1 r3 r4)
: rice ." rice " ;              ( is never reached)
: entree chicken exit rice ;    ( r1 r3)
: dinner soup entree dessert ;  ( r1)
dinner cr                       ( --)
```

After "CHICKEN" has been executed by "ENTREE", an 'EXIT' is encountered. 'EXIT' works just like ';', so 4tH thinks the colon-definition has come to an end and jumps back to "DINNER". It never comes to calling "RICE", so the output is:

```
soup chicken dessert
```

'EXIT' is mostly used in combination with some kind of branching like IF..ELSE..THEN. Compare it with 'LEAVE' that leaves a DO..LOOP early.

But now for the big question: what is the difference between 'EXIT' and ';'? Both compile an 'EXIT', but they are not aliases. 4tH will try to match every ';' with a ':'. If it doesn't succeed, it will issue an error message. This matching is not performed by 'EXIT'.

10.16 The layout of the Stack Area

Before we tell you how to obtain information on the Stack Area, we first have to explain you how it is laid out. We've already seen that there are two stacks: the Data Stack and the Return Stack. We also know what they are used for.

The next question is what part of the Stack Area is used by the Data Stack and what part is used by the Return Stack. In fact, both stacks share the very same Stack Area.

The Data Stack grows upward from the bottom and the Return Stack grows downward from the top. When they meet, you're in trouble. If the Return Stack causes the overflow, 4tH will report that the Return Stack overflowed. If it was the Data Stack, it will report that the Data Stack overflowed.

If an overflow happens, you can't say which stack actually overflowed. If the Data Stack filled up the Stack Area and a colon-definition tries to put a return address on the Return Stack, the Return Stack will get the blame.

Now for the good news. Because of this shared stack space, programs with different requirements can run without having to modify stack sizes (you can't do that; only the programmer of your application can). It can be a program that heavily uses the Return Stack (recursive colon-definitions) or a program that needs lots of data on the Data Stack.

What you can check is how big the Stack Area actually is. It is a constant named 'STACK'. It will report the size in cells. Every value on any stack (address or value) takes up a single cell.

You can also ask 4tH how many values are on the Data Stack using 'DEPTH'. It will report the number of values, before you executed 'DEPTH'. Let's elaborate on that a little more:

```
." Begin" cr                \ no values on the stack
10                          \ 1 value on the stack
5                           \ 2 values on the stack
9                           \ 3 values on the stack
depth                       \ 4 values on the stack
. cr                        \ 4tH reports "3"
```

If you want to know what values the actual stack pointers have, you have to use 'SP@' and 'RP@'. By subtracting 'SP@' from 'RP@' you can see how much space is left in the Stack Area:

```
rp@ sp@ -
." Space left: " . ." cells" cr
```

10.17 Booleans and numbers

You might have expected we had discussed this subject much earlier. But we haven't and for one very good reason. We've told you a few chapters ago that 'IF' branches if the top of the stack is non-zero. Any number will do. So you would expect that this program will print "I'm here":

```
1 2 and
if
  ." I'm here"
then
```

In fact, it doesn't! Why? Well, 'AND' is a BINARY operator, not a LOGICAL operator. That means it reacts on bit-patterns. Given two numbers, it will evaluate bits at the same position.

The number "1" is "01" in binary. The number "2" is "10" in binary. 'AND' will evaluate the first bit (binary digit, now you know where that came from!). The first bit is the rightmost bit, so "0" for the number "2" and "1" for the number "1".

'AND' works on a simple rule, if both bits are "1" the result will be "1" on that position. Otherwise it will be "0". So "1" and "0" are "0". The evaluation of the second bit has the same result: "0". We're stuck with a number that is "0". False. So 'IF' concludes that the expression is not true:

```
2 base ! [binary]             \ set radix to binary
10                            \ binary number "2"
01 AND                        \ binary number "1"
. cr                          \ binary result after AND
```

It will print "0". However, "3" and "2" would work just fine:

```
2 base ! [binary]             \ set radix to binary
10                            \ binary number "2"
11 AND                        \ binary number "3"
. cr                          \ binary result after AND
```

It will print "10". The same applies to other binary operators as 'OR' and 'INVERT'. 'OR' works just like 'AND' but works the other way around. If both bits are "0" the result will be "0" on that position. Otherwise it will be "1":

```
2 base ! [binary]             \ set radix to binary
10                            \ binary number "2"
01 OR                         \ binary number "1"
. cr                          \ binary result after OR
```

It will print "11". We do not encourage the use of 'INVERT' for logical operations. You should use '0=' instead.

'0=' takes the top of the stack and leave a true-flag if it is zero. Otherwise it will leave a false-flag. That means that if a condition is true (non-zero), it will leave a false-flag. Which is exactly what a logical NOT should do.

Take a look at his brother '0<>'. '0<>' takes the top of the stack and leaves a true-flag if it is non-zero. Otherwise it will leave a false-flag.

The funny thing is 'AND' and 'OR' work perfectly with flags and behave as expected. '0<>' will convert a value to a flag for you. So this works:

```
1 0<>
2 0<>
and if
  ." I'm here" cr
then
```

Of course, you don't have to use '0<>' when a word returns a flag. You should check the glossary for details on that.

10.18 Using ' with other names

So far we've only used "' (tick) with colon-definitions, but you can also use it with all constants, variables, values, strings, vectors (see section 11.7) and constant arrays. However, the information it provides is not always useful. E.g. the expression:

```
10 constant ten
' ten
```

Does not compile differently from:

```
10 constant ten
ten
```

The same applies to constant arrays and strings. It will give you possibly information on the address of variables, vectors, arrays and values, e.g.:

```
variable ten
' variable ." Relative address of ten: " . cr
```

Yes, *relative* address! What does that mean? When a 4tH program is compiled it has no idea how many application variables a host program will provide. So it stores a *relative* address. This address is relative to the address returned by 'FIRST'. You might call it an offset if you want to. 4tH provides a word which will convert the relative address of vectors, variables, values and numeric arrays to an absolute address, called '>BODY'. So this piece of code does exactly the same thing:

```
variable ten
ten                          \ throw address of 'ten' on stack
dup                          \ duplicate address
10 swap !                    \ store 10 at address
? cr                         \ show value stored at address
```

As this piece of code:

```
variable ten
' ten >body                  \ calculate address
dup                          \ duplicate address
10 swap !                    \ store 10 at address
? cr                         \ show value stored at address
```

There are not too many occasions where this is useful, but it let's take a look at this one:

```
0 value ten                  \ define a value
' ten >body                  \ calculate address of value
dup                          \ duplicate address
10 swap !                    \ store 10 at address
? cr                         \ show value stored at address
```

We already know that values, numeric arrays and vectors are stored in the very same area of the Integer Segment. This construction makes it possible to access them as variables.

You can access string constants or arrays of string constants with tick, but they will return a value which only has a meaning to 4tH itself. You won't be able to do anything useful with those values.

You should avoid these kind of constructions, but there might be some situations out there where it might come in handy. Note that you can only tick your *own* names. All of 4tHs built-in variables, strings, words, etc. cannot be accessed by tick.

10.19 Deleting files

4tH can also delete files on disk. That's very easy, just feed it the name or path of the file in question:

```
s" delete.me" delete-file .
```

If the deletion was successful, it prints zero, otherwise non-zero.

10.20 Querying environment variables

You can also retrieve the contents of every environment variable. Just feed 4tH with the name of your environment variable and it will return the address and count of its contents:

```
s" DIR4TH" environ@ type cr
```

This will print the contents of your DIR4TH environment variable. Note that the contents may be truncated if they would overflow the PAD. In such cases, you may want to use the libraryfile `getenv.4th`. This allows you to determine the string variable where the contents will be written to. However, it doesn't work on systems that do not support pipes (see section 9.27) and is a bit slower:

```
include lib/getenv.4th
1024 constant /buffer
/buffer string buffer
s" DIR4TH" buffer /buffer getenv type cr
```

Note that the contents are truncated as well if you don't make the buffer large enough. Like 'ENVIRON@' it will return an empty string on error.

10.21 What is not implemented

When writing a product like 4tH that is modelled after an existing programming language like Forth one has to cut a few corners somewhere.

Forth has a fundamentally different architecture, which allows you to extend the compiler with ease. 4tH is much more like conventional programming languages and many still wonder how we got this far.

When you're learning 4tH to learn Forth you will find there are things you can't do in 4tH. This section sums up most of the restrictions 4tH has in comparison to Forth and other languages.

Datatypes There are no words that allow you to define your own datatypes, although you can change the behaviour of individual variables.

Interpreter Since 4tH is a conventional compiler, you won't find a built-in interpreter. There is a library-source, which will enable you to make an interpreter for specific applications with ease. Next chapter we will show you how to use it.

If you have more questions concerning the functionality of 4tH, please read the ANS-Forth document. This describes the compliance of 4tH to the ANS-Forth standard. Further information can be obtained by studying the glossary.

10.22 Known bugs and limitations

Like every software product, 4tH has bugs. Because a work-around is available, fixing these bugs has no high priority.

- When you use '\' without any actual comment in a Unix ASCII file the complete next line will be marked as comment. With MS-DOS ASCII files 4tH will correctly detect a null string and terminate with an error. Add at least a dot or something, just to be safe.
- There can be only *one space* between '[DEFINED]', '[UNDEFINED]', '[CHAR]', 'CHAR' and the string following it. If you don't comply, 4tH will complain about empty string constants.
- You cannot use 'HIDE' conditionally. If you specify a 'HIDE' it must *always* be executed in 4tH.

Chapter 11

Advanced programming

11.1 Compiletime calculations

When you've reached this chapter, you must have quite some experience with 4tH. This chapter will help you to use 4tH to its full capacity. You'll be able to use software exceptions, conditional compilation, compiletime calculation, enumerations, private declarations and much, much more.

We've already explained that when you define a string, it has to be preceded by a literal expression:

```
64 constant name
16 constant #names

name #names * string name_space
```

The word '*' takes two subsequent literal expressions and multiplies them to a new single literal expression as if you'd written "1024" yourself! Everything that is evaluated at compiletime, *must* already be known at compiletime, thus a literal expression. Although not all arithmetic words compile to a literal expression, you can use the most common ones, like '*', '/', '+' and '-'. Another useful word is 'NEGATE', e.g. when you need to assign a negative value to a constant:

```
             16 constant +range
+range negate constant -range
```

In this example the value of "-RANGE" is -16. You can even mix and chain compiletime calculations. This will compile to the literal "500":

```
5 25 75 + *
```

Just as if you'd just written "500" in the sourcecode yourself. You can also write:

```
25 75 + 5 *
```

Because it's just simple postfix notation. Note that there must be two subsequent literal expressions available at any time, so this doesn't work:

```
5 5 * dup *
```

Since 'DUP' isn't a literal expression, but a word which is simply compiled. But don't worry: 4tH will notify you when you make an error like this. The final word, we'd like to present you is '[NOT]', which logically inverts a flag at compiletime, just like '0=' at runtime. This expression will compile to a true flag:

```
false [not]
```

You might wonder why we included this one, but that will become clear when you read the next section.

11.2 Conditional compilation

This is something which can be very handy when you're designing a 4tH program for different environments or even different Forth compilers. Let's say you've written a general ledger program in 4tH that is so good, you can sell it. Your customers want a demo, of course. You're willing to give one to them, but you're afraid they're going to use the demo without ever paying for it.

One thing you can do is limit the number of entries they can make. So, you copy the source and make a special demo version. But you have to do that for every new release. Wouldn't it just be easier to have one version of the program and just change one single constant? You can with conditional compilation:

```
true constant DEMO

DEMO [if]
256 constant #Entries
[else]
limit constant #Entries
[then]

variable CurrentEntry
#Entries array Entries
```

We defined a constant, called "DEMO", which is true. So, when the compiler reaches the "DEMO [IF]" line, it knows that it has to compile "256 constant Entries", since "DEMO" is true. When it comes to '[ELSE]', it knows it has to skip everything up to next '[THEN]'. So, in this case the compiler behaves like you've written:

```
256 constant #Entries
variable CurrentEntry
#Entries array Entries
```

Would you change "DEMO" to false, the compiler would behave as if you wrote:

```
limit constant #Entries
variable CurrentEntry
#Entries array Entries
```

The word '[IF]' only works at compile time and is *never* compiled into the object. '[IF]' takes a literal expression. If this expression is true, the code following the '[IF]' is compiled, just as '[IF]' wasn't there. Is this expression false, everything up to '[ELSE]' or '[THEN]' is discarded as if it wasn't there.

That also means you can discard any code that is superfluous in the program. E.g. when you're making a colon-definition to check whether you can make any more entries. If you didn't use conditional compilation, you might have written it like this:

```
: CheckIfFull                 ( n -- n)
  dup #Entries =              ( n f)
  if                          ( n)
    drop                      ( --)

    DEMO                      ( f)
    if                        ( --)
      ." Buy the full version"
    else                      \ give message and exit program
      ." No more entries"
    then                      ( --)

    cr quit
  then                        ( n)
;
```

But his one is nicer and will take up less code:

```
: CheckIfFull                 ( n -- n)
  dup #Entries =              ( n f)
  if                          ( n)
    drop                      ( --)

DEMO [if]                     ( n f)
    ." Buy the full version"
[else]
    ." No more entries"
[then]

    cr quit
  then                        ( n)
;
```

You can also use conditional compilation to discard large chunks of code. This is a much better way than to comment all the lines out, e.g. this won't work anyway:

```
(
: room?                       \ is it a valid variable?
  dup                         ( n n)
  size 1- invert and          ( n f)
  if                          \ exit program
    drop ." Not an element of ROOM" cr quit
  then
;
)
```

This is pretty cumbersome and prone to error:

```
\ : room?                     \ is it a valid variable?
\   dup                       ( n n)
\   size 1- invert and        ( n f)
\   if                        \ exit program
\     drop ." Not an element of ROOM" cr quit
\   then
\ ;
```

But this is something that can easily be handled:

```
false [if]
: room?                       \ is it a valid variable?
  dup                         ( n n)
  size 1- invert and          ( n f)
  if                          \ exit program
    drop ." Not an element of ROOM" cr quit
  then
;
[then]
```

Just change "false" to "true" and the colon-definition is part of the program again. Note that '[IF]..[ELSE]..[THEN]' can be nested! Conditional compilation is very powerful and one of the easiest features a language can have. And it's ANS-Forth compatible!

11.3 Checking the environment at compiletime

Let's say you've written something which works perfectly on your own machine and you want to use it on the mainframe at work. It turns out to be it doesn't work. Why? Because your program assumed that a cell was four address units wide. And it didn't turn out to be that way.

You could have prevented that if you had used a check at compiletime. You can do that this way:

```
/cell 4 [=] [NOT] [IF]
( do something)
[THEN]
```

'/CELL' is a constant which holds the number of address units in a cell. '/CELL' has got a little brother called '/CHAR', which will tell you how many address units there are in a character. '[=]' will check whether a cell has four address units and '[NOT]' will reverse that flag. Neat huh?

But then again, what do we do if it doesn't turn out to be that way. Any action will first be executed at runtime so a message or 'ABORT' won't do. Further compilation will be useless, so we actually want to stop. You're in luck, since we have a special word that will stop the compiler regardless. It's called '[ABORT]'. So this is our complete snippet:

```
/cell 4 [=] [NOT] [IF]
[ABORT]
[THEN]
```

But suppose you want to check whether a cell is *at least* 4 address units. '[=]' won't do in that case. Of course, you can check every imaginable cellsize, but that is not very pretty. That is where '[SIGN]' comes in. '[SIGN]' will take a previously compiled literal expression and compile -1 if the number was negative, zero if the number was zero and 1 if the number was positive. You may wonder how that does help. Well, consider this one:

```
/cell 4 - [SIGN] -1 [=] [IF]
[ABORT]
[THEN]
```

What have we been doing here? First, we substract the minimal cellsize from the actual cellsize. If the sign of the sum is -1, compilation is aborted. The sum can *only* be negative when '/CELL' is three or smaller. Get it? By using '[SIGN]' you can make all kinds of compiletime comparisons, which makes it a real asset.

11.4 Checking a definition at compiletime

We've already encountered 'COMPARE' in section 8.17. 'COMPARE' is word that compares two strings. It can do that both case sensitive and case insensitive. If you define a constant called `casesensitive` before the '[NEEDS' directive, it will perform a case sensitive comparison. If you don't, it will do a case insensitive comparison by default.

Most approaches would require the definition of `casesensitive`, regardless which mode you select. This one doesn't:

```
: compare                              ( a1 n1 a2 n2 -- f )
  rot over over swap - >r              ( a1 a2 n2 n1)
  min 0 tuck                           ( a1 a2 0 n 0)
  ?do                                  ( a1 a2 f)
    drop                               ( a1 a2)
    over i + c@                        ( a1 a2 c1)
[UNDEFINED] casesensitive [IF]
    dup [char] A - max-n and 26 < if bl or then
[THEN]
    over i + c@                        ( a1 a2 c1 c2)
[UNDEFINED] casesensitive [IF]
    dup [char] A - max-n and 26 < if bl or then
[THEN]
    - dup                              ( a1 a2 f f)
    if leave then                      ( a1 a2 f)
  loop
  >r drop drop r> r> swap dup          ( f1 f2 f2)
  if swap then drop                    ( f)
;
```

'[UNDEFINED]' checks whether the word following it has been defined and leaves a TRUE flag if it wasn't. It doesn't matter whether the word is built-in, included or defined in your program. It can be a variable, a word, a constant, anything you like.

In this case it checks whether `casesensitive` has been defined. If it isn't, a line of code is compiled. If an 'IF' had been used, the code would always be compiled with the added overhead of testing a constant at runtime. This construction allows for tighter and faster code.

Note that '[UNDEFINED]' has also a counterpart called '[DEFINED]'. It leaves a true flag when the word following it has been defined.

11.5 Exceptions

You know when you violate the integrity of 4tH, it will exit and report the cause and location of the error. Wouldn't it be nice if you could catch these errors within the program? It would save a lot of error-checking anyway. It is quite possible to check every value within 4tH, but it takes code and performance, which makes your program less compact and slower.

Well, you can do that too in 4tH. And not even that, you can trigger your own errors as well. This simple program triggers an error and exits 4tH when you enter a "0":

```
[needs lib/enter.4th]          \ get a number
                               \ if non-zero, return it
                               \ if zero, throw exception
: could-fail                   ( -- n)
  enter dup 0=
```

```
  if 1 throw then
;
                                \ drop numbers and
                                \ call COULD-FAIL
: do-it                         ( -- )
  drop drop could-fail
;
                                \ put 2 nums on stack and
                                \ execute DO-IT
: try-it                        ( -- )
  1 2 ['] do-it execute
  ." The number was" . cr
;
                                \ call TRY-IT
try-it
```

"TRY-IT" puts two numbers on the stack, gets the execution token of "DO-IT" and executes it. "DO-IT" drops both numbers and calls "COULDFAIL". "COULD-FAIL" gets a number and compares it against "0". If zero, it calls an exception. If not, it returns the number.

The expression "1 THROW" has the same effect as calling 'QUIT'. The program exits, but with the error message "Unhandled exception". You can use any positive number for 'THROW', but "0 THROW" has no effect. This is called a "user exception", which means you defined and triggered the error.

There are also system exceptions. These are triggered by the system, e.g. when you want to access an undefined variable or print a number when the stack is empty. These exceptions have a negative number, so:

```
throw -4
```

Will trigger the "Stack empty" error. You can use these if you want but we don't recommend it, since it will confuse the users of your program.

You're probably not interested in an alternative for 'QUIT'. Well, 'THROW' isn't. It just enables you to "throw" an exception and exceptions can be caught by your program. That means that 4tH won't exit, but transfers control back to some routine. Let's do just that:

```
[needs lib/enter.4th]

: could-fail                    ( -- n)
  enter dup 0=
  if 1 throw then
;

: do-it                         ( -- )
  drop drop couldfail
;

: try-it                        ( -- )
  1 2 ['] do-it catch
  if drop drop ." There was an exception" cr
  else ." The number was" . cr
  then
;

try-it
```

The only things we changed is a somewhat more elaborate "TRY-IT" definition and we replaced 'EXECUTE' by 'CATCH'.

'CATCH' works just like 'EXECUTE', except it returns a result-code. If the result-code is zero, everything is okay. If it isn't, it returns the value of 'THROW'. In this case it would be "1", since we execute "1 THROW". That is why "0 THROW" doesn't have any effect.

If you enter a nonzero value at the prompt, you won't see any difference with the previous version. However, if we enter "0", we'll get the message "There was an exception", before the program exits.

But hey, if we got that message, that means 4tH was still in control! In fact, it was. When "1 THROW" was executed, the stack-pointers were restored and we were directly returned to "TRY-IT". As if "1 THROW" performed an 'EXIT' to the token following 'CATCH'.

Since the stack-pointers were returned to their original state, the two values we discarded in "DO-IT" are still on the stack. But the possibility exists they have been altered by previous definitions. The best thing we can do is discard them.

So, the first version exited when you didn't enter a nonzero value. The second version did too, but not after giving us a message. Can't we make a version in which we can have another try? Yes we can:

```
[needs lib/enter.4th]

: could-fail                    ( -- n)
  enter dup 0=
  if 1 throw then
;

: do-it                         ( -- )
  drop drop could-fail
;

: retry-it                      ( -- )
  begin
    1 2 ['] do-it catch
  while
    drop drop ." Exception, keep trying" cr
  repeat
  ." The number was " . cr
;

retry-it
```

This version will not only catch the error, but it allows us to have another go! We can keep on entering "0", until we enter a nonzero value. Isn't that great? But it gets even better! We can exhaust the stack, trigger a system exception and still keep on going. But let's take it one step at the time. First we change "COULD-FAIL" into:

```
: could-fail                    ( -- n)
  enter dup 0=
  if drop ." Stack: " depth . cr 1 throw then
;
```

This will tell us that the stack is exhausted at his point. Let's exhaust is a little further by redefining "COULD-FAIL" again:

```
: could-fail                    ( -- n)
  enter dup 0=
  if drop drop then
;
```

Another 'DROP'? But wouldn't that trigger an "Stack empty" error? Yeah, it does. But instead of exiting, the program will react as if we wrote "-4 THROW" instead of "DROP DROP". The program will correctly report an exception when we enter "0" and act accordingly.

This will work with virtually every runtime error. Which means we won't have to protect our program against every possible user-error, but let 4tH do the checking.

We won't even have to set flags in every possible colon-definition, since 4tH will automatically skip every level between 'THROW' and 'CATCH'. Even better, the stacks will be restored to the same depth as they were before 'CATCH' was called.

You can handle the error in any way you want. You can display an error message, call some kind of error-handler, or just ignore the error. Is that enough flexibility for you?

11.6 Enumerations

Sometimes you need a lot of constants:

```
0 constant Monday
1 constant Tuesday
2 constant Wednesday
3 constant Thursday
4 constant Friday
5 constant Saturday
6 constant Sunday
```

A little error here may ruin your program. This does the very same thing, except it is easier to maintain:

```
0 enum Monday   enum Tuesday enum Wednesday
  enum Thursday enum Friday enum Saturday
  enum Sunday   drop
```

'ENUM' is much like a 'CONSTANT', but increments and leaves a value after the constant has been created. That is why we need to add 'DROP' after the final enumeration. To show you that 'ENUM' and 'CONSTANT' are much alike, you could also write the declaration above as:

```
0 enum Monday   enum Tuesday enum Wednesday
  enum Thursday enum Friday enum Saturday
  constant Sunday
```

Since 'CONSTANT' just consumes the value, you don't need the final 'DROP'.

11.7 Forward declarations

It doesn't happen very often, but sometimes you have a program where two colon-definitions call each other. When you look at 4tHs source you find several examples. The throw() function calls the rpop() function, because 'THROW' takes items from the Return Stack. On the other hand, when the Return Stack underflows, it has to call 'THROW'.

There is a special instruction in 4tH to do this, called 'DEFER'. 'DEFER' doesn't create an executable word, but a vector containing an execution token[1], which is executed when called. You might want to consult section 10.3 first to see how this works. But for all purposes you might consider it to be an executable word, because it behaves the same way.

[1] See section 10.4.

```
defer Step2
```

Now we can create "STEP1" without a problem:

```
: Step1 1+ dup . cr Step2 ;
```

But "STEP2" does not have a body yet. Of course, you could create a new colon-definition, tick it and assign the execution token to "STEP2" manually, but it is much neater to use ':NONAME'. ':NONAME' can be used like a normal ':', but it doesn't require a name. Instead, it pushes the execution token of the colon-definition it created on the stack. No, ':NONAME' does *not* create a literal expression, but it is just what we need:

```
:noname 1+ dup . cr Step1 ; is Step2
```

Now we are ready! We can simply execute the program by calling "STEP1":

```
1 Step1
```

Note that if you run this program, you'll get stack errors! Sorry, but the example has been taken from a Turbo Pascal manual ;-). If you have forgotten what a deferred word actually executes, you can retrieve the execution token by using 'DEFER@':

```
defer thisword                \ create a vector

: plus + ;                    \ define a word

' plus is thisword            \ assign the word to the vector
' thisword defer@             \ retrieve the execution token
2 3 rot execute               \ execute the deferred word
. cr                          \ display the result
```

As a matter of fact, this expression:

```
' thisword defer@ execute
```

Is equivalent to this one:

```
thisword
```

You can also reassign a vector without using 'IS'. 'IS' is a parsing version. That means the actual vector to which a certain behaviour is assigned is determined at *compiletime*. 'DEFER!' can be used to assign a certain behaviour at *runtime*. 'DEFER!' takes two execution tokens:

```
defer thisword                \ create a vector

: plus + ;                    \ define a word

' plus ' thisword defer!      \ assign it to a vector
```

This is equivalent to this:

```
defer thisword                \ create a vector

: plus + ;                    \ define a word

' plus is thisword            \ assign it to a vector
```

I guess you'll agree with me that this creates countless possibilities.

11.8 Recursion

Yes, but can she do recursion? Of course she can! It is even very natural and easy. Everybody knows how to calculate a factorial. In 4tH you can do this by:

```
: factorial                    ( n1 -- n2)
  dup 2 >
  if
    dup 1-
    factorial *
  then
;

10 factorial . cr
```

Which is exactly as one would expect. Unfortunately, this is not the way it is done in ANS-Forth. In order to let a colon-definition call itself, you have to use the word 'RECURSE'. 4tH supports this word too:

```
: factorial                    ( n1 -- n2)
  dup 2 >
  if
    dup 1-
    recurse *
  then
;

10 factorial . cr
```

It will even compile to the same code. If you use the word 'RECURSE' outside a colon-definition, the results are undefined. Note that recursion lays a heavy burden on the return stack. Sometimes it is wiser to implement such a routine differently:

```
: factorial
  dup
  begin
    dup 2 >
  while
    1- swap over * swap
  repeat

  drop
;

10 factorial . cr
```

So if you ever run into stack errors when you use recursion, keep this in mind.

11.9 Private declarations

Sometimes you want to hide some definitions from other programmers. This is especially true when you're writing libraries or templates. The Application Programmers Interface must be public of course, but you don't want anyone else to tinker with the internals of your library. And there is the problem of cluttering your name space.

Relax, 4tH has a way to get rid of these internal words. It's easy, just tell 4tH to hide them:

```
VARIABLE #emits                         \ private

: SHOW emit 1 #emits +! ;               \ public
: NL CR 0 #emits ! ;                    \ public

hide #emits
```

After that the name "#EMITS" is no longer recognized and can be reused if you want to, e.g. this is completely valid:

```
: dummy ;
hide dummy
: dummy ." I am no longer a dummy!" cr ;
```

As a matter of fact, the previous declaration of "DUMMY" has been turned into a ':NONAME' declaration by the use of 'HIDE'. Note that 'HIDE' is meant to make definitions *private*; not to optionally override previously defined definitions in order to achieve a Forth-like behavior:

```
[DEFINED] myoption [IF]
hide myoption
[THEN]

true CONSTANT myoption
```

If you want to achieve that, use the "first come, first served" rule:

```
[UNDEFINED] myoption [IF]
true CONSTANT myoption
[THEN]
```

The constant "MYOPTION" is only defined if it hasn't been defined before. You may have to change a few things here and there to make it work, but there are obvious advantages to this construction.: it saves memory and is much easier to understand and maintain. And you won't scratch your head why your perfectly valid 4tH program doesn't compile[2].

11.10 Aliases

Sometimes you want to make an alias for a word. Of course you can embed the word you want to alias in a new definition:

```
: noop ;
: nop noop ;
```

Although this approach works perfectly under all circumstances it has its disadvantages, because calling a word is relatively slow. Unless you're trying to make an alias for an internal word, you'd better use an 'ALIAS ':

```
: noop ;
' noop alias nop
```

[2]The tokenizer evaluates 'HIDE' in order to estimate the size of the symboltable while conditional compilation is first evaluated by the parser.

This is completely equivalent to:

```
defer nop
: noop ;
' noop is nop
```

Although the vector takes up a little space, it will save you from most of the calling overhead. Since you can only alias self-defined executable words, 'ALIAS' is quite limited. 'AKA' does not have that disadvantage:

```
: noop ;
aka noop nop
```

Both words are now completely equivalent and compile to exactly the same code. Even better, you can use 'AKA' with *every* self-defined word, including variables, vectors, files, values, fields and constants. 'AKA' even works with most built-in 4tH words, e.g. constants:

```
aka + plus
```

'AKA' will *not* work with defining, preprocessor, flow control words or inline macros[3]. If you're unsure, just try! If 4tH bombs out with an error message, you'll know. 'AKA' is also known as "also known as".

11.11 Changing behavior of data

One of the most ingenious things Forth can do, is change the behavior of data at runtime. With 4tH, you cannot do this for an entire datatype, but you can do it for individual 'VARIABLE's, 'CREATE's, 'STRING's, 'ARRAY's and 'CONSTANT's. Just use:

```
:REDO <name> <definition> ;
```

Where <name> is a previously defined 'VARIABLE', 'STRING', etc. The definition will behave as if the 'VARIABLE', etc. has just been thrown on the stack, e.g. to make a 'VARIABLE' behave as a 'CONSTANT 'you define:

```
variable me

10 me !
:redo me @ ;
```

The body of the definition will behave as if it said:

```
me @
```

Which boils down to a (rather slow) constant. You cannot change the contents of the variable anymore if you haven't taken precautions, because there is no way to address it. Here is another, more elaborate example:

[3] Rule of the thumb: any word that compiles to a single token. See the glossary for details.

```
create life                   \ create an array of string constants
  ," This is my life!"
  ," This is your life!"

0 constant my                 \ create two constants
1 constant your               \ to address the elements
                              \ now change the behaviour of LIFE
:redo life swap th @c count type cr ;

my life                       \ use it!
your life
```

At runtime, this will print:

```
This is my life!
This is your life!
```

There is even a shorter way to do this by using 'DOES>', e.g. we could also write the example above as:

```
create life                   \ create an array of string constants
  ," This is my life!"
  ," This is your life!"
does> swap th @c count type cr ;

0 constant my                 \ create two constants
1 constant your               \ to address the elements
                              \ now change the behaviour of LIFE
my life                       \ use it!
your life
```

'DOES>' always applies to the last defined 'VARIABLE', 'CREATE', 'STRING', 'ARRAY' or 'CONSTANT', so you don't have to repeat the name. So why ':REDO', you might ask. Well, sometimes you have to define some other words before you encapsulate the data. ':REDO' allows you to do just that.

If you just want to initialize the data, you can do so with 'LATEST'. 'LATEST' will compile to the last word you have defined, so if you want to change the name of your definition, that's the only thing you have to do. E.g. this is the 4tH translation of a double number[4] constant:

```
include lib/anscore.4th       \ for 2@, 2!
include lib/ansdbl.4th        \ for U>D

2 array mydouble              \ define a double number
max-n u>d latest 2!           \ initialize it
does> 2@ ;                    \ now make it a constant
```

Wording has always been very important to Forth. Using this technique, you can make your programs even more readable.

[4] See section 12.22.

11.12 Multidimensional arrays

We've seen two dimensional arrays with 'ARRAY' and 'STRING', but what about multi-dimensional arrays. Well, it's the same thing all over again. C doesn't actually have multi-dimensional arrays either. When you define one, just a chunk of memory is allocated.

In 4tH you can do the same thing, but now you have to do it yourself. E.g. when you want to define a matrix of cells of 4 rows by 5 elements, you have to multiply those and allocate an array of that size:

```
4 5 * array my_array
```

But what if you want to reference the fourth element of the third row? You *cannot* write something like:

```
2 3 my_array @
```

That's right. But you can change the behaviour of "MY_ARRAY" accordingly:

```
:redo my_array                ( n1 n2 -- a)
  rot 5 *                     \ calculate row offset
  rot +                       \ calculate element offset
  cells                       \ calculate number of cells
  +                           \ add to address of my_array
;
```

Or even better:

```
4 5 * array my_array does> rot 5 * rot + cells + ;
```

This word calculates the correct offset for you. Note that the third row is row number two (since we start counting from 0) and the fourth element is element number three:

```
2 3 my_array @
```

You can also use "MY_ARRAY" to initialize an array, since it simply calculates the correct address for you:

```
5 2 3 my_array !              \ sets 3rd row 4th element to 5
```

You can add more dimensions if you want. This works basically the same way: create an array of a size that equals the products of its dimensions and design a word that calculates the correct address.

11.13 Binary string constants

A binary string constant is an unterminated string that doesn't necessarily contain characters. Creating binary string constants is easy. Just compile them by their ASCII value into the String Segment with 'C,':

```
char H c, char i c, char ! c, 0 c,
```

The fun of it all is that 4tH doesn't allow you to access the String Segment directly, so you can *never* retrieve them. You need 'OFFSET' to define a word which does all the hard work for you. At runtime it takes an index and leaves the ASCII value of the character in question on the stack. 'OFFSET' is used just before you compile the ASCII values:

```
offset greet char H c, char i c, char ! c, 0 c,
```

Note that you have to terminate a binary string constant *manually* if you need to, although it is perfectly legal to create binary string constants with no termination at all. Retrieving characters is easy. This will print "Hi!":

```
0 greet emit
1 greet emit
2 greet emit cr
```

And so will this:

```
0 begin                         \ setup index
  dup greet dup                 \ retrieve character
while                           \ if not terminated
  emit 1+                       \ emit and increase index
repeat drop drop                \ clear stack
```

You can use binary string constants for compact tables, bitstrings or any other raw data as long as each element doesn't exceed the size of a single character. Although useful in itself, you'll get into trouble when you try to pack several binary strings in a single datastructure. Sure, you can give each binary string its own 'OFFSET' but binding their addresses into some form of array structure? No, because an 'OFFSET' is not a literal expression and hence cannot be compiled that way.

A solution is to "tag" individual strings. Tags are literal expressions and compilable, so you can do this:

```
OFFSET names
  TAG names Hans
    char H c, char a c, char n c, char s c, 0 c,
  TAG names Wil
    char W c, char i c, char l c, 0 c,
  TAG names Phil
    char P c, char h c, char i c, char l c, 0 c,
```

Or this, if you want to make it a little less "wordy":

```
OFFSET names
  TAG names Hans c" Hans" 0 c,
  TAG names Wil  c" Wil"  0 c,
  TAG names Phil c" Phil" 0 c,
```

Both expressions are completely equivalent. 'C"' simply offers you a much easier way to enter a *printable* string. Note that 'C|' is available too, if you happen to use embedded quotes. And yes, you can mix 'C,' and 'C"' as much as you like.

You can feed the 'TAG' as an index to an 'OFFSET', so this will return "W" (from "Wil"):

```
Wil names emit
```

And as promised, this will work as well:

```
CREATE developers
  Hans , Phil , Wil , NULL ,
```

Contrary to popular belief, this construct is as memory efficient as:

```
CREATE developers
  ," Hans" ," Phil" ," Wil"  NULL ,
```

It is just a bit more cumbersome to set up. You can even wrap up your binary string array into a ':REDO' definition, giving:

```
:redo developers
  swap cells + @c
  begin dup names dup while emit 1+ repeat
  drop drop cr
;
```

Which means you can write:

```
1 developers
```

Which will display "Phil". This way, embedding e.g. UTF-8 code not only becomes possible, but also very easy to set up.

11.14 Binary string variables

You can define raw chunks of memory with 'BUFFER:'. This will allocate a raw chunk of data of 1024 bytes:

```
1024 buffer: a
```

Finally, if you want to move raw chunks of data around, there is 'MOVE':

```
1024 buffer: a
1024 buffer: b

a b 512 move
```

This wil define two raw buffers of 1024 bytes and 'MOVE' will copy the first 512 bytes of buffer "a" to buffer "b".

11.15 Records and structures

The easiest way is to allocate a structure in the Character Segment. Just define the structure like this:

```
struct
  32 +field Name
  64 +field Address
  32 +field City
   4 +field Age
end-struct /Person
```

This might be a familiar example to you. We'll store information on a single person in this structure. Now we got the fields, the length of the fields and the length of the entire structure, stored in "/Person". Both the fields and the entire structure are nothing more than a set of constants, e.g. the offset of the field "Name" is stored in a CONSTANT named "Name". However, we still haven't allocated any memory. We can allocate room for the structure we've just defined by just using the word 'STRING'. Note that you can also create a cell-based structure. Then you need the word 'ARRAY' to allocate the memory required.

```
/Person string Person
```

Now we can define a word which initializes the fields:

```
: InitRecord                  \ initialize fields
  s" Hans Bezemer" Person -> Name    place
  s" Lagendijk 79" Person -> Address place
  s" Den Helder"   Person -> City    place
  s" 44"           Person -> Age     place
;
```

Of course, you can also use 'ACCEPT' to enter the contents of the fields. Fields act like ordinary strings. Note that numbers are stored as strings as well. This is not too much of a problem since 'NUMBER' can convert them back to numbers anyway.

This is a very simple use of structures. You can also use structures within structures:

```
struct
  64 +field Address
  32 +field City
end-struct /Location

struct
  32 +field Name
  /Location field Location
end-struct /Person

/Person string Person

s" Delft" Person -> Location -> City place
```

If you want to make an array of structures, that can be done as well:

```
struct                      \ create structure
  32 +field Name
  64 +field Address
  32 +field City
   4 +field Age
end-struct /Person

32 constant #Person         \ size of array of structs
                            \ now allocate the room
#Person /Person * string Persons
                            \ make it behave properly
:redo Persons swap /Person * + ;
```

```
                            \ initialize the first record
s" Hans Bezemer" 0 Persons -> Name    place
s" Lagendijk 79" 0 Persons -> Address place
s" Den Helder"   0 Persons -> City    place
s" 44"           0 Persons -> Age     place
```

You can also extend an already existing structure:

```
struct                      \ create structure
  32 +field Name
  64 +field Address
  32 +field City
  12 +field Age
end-struct /Person
                            \ now extend the structure
/Person
  32 +field Job
  16 +field Emp-number
end-struct /Employee
```

You now got two different structures, ”/Person” and ”/Employee”, that share the first four fields. Defining a structure within a structure is possible too:

```
struct                      \ create structure
  32 +field Firstname
  64 +field Lastname
  struct                    \ structure within a structure
    64 +field Address
     8 +field Zip
    32 +field City
  +field Location           \ let's give it a name
  4  +field Age
end-struct /Person
```

Let's use it:

```
/Person string Person       \ allocate some space
                            \ initialize it
s" Hans"         Person -> Firstname place
s" Bezemer"      Person -> Lastname  place
s" Lagendijk 79" Person -> Location -> Address place
s" Den Helder"   Person -> Location -> City    place
s" 44"           Person -> Age  place
```

And finally, if you want to define a structure and allocate the memory it occupies at the same time, you can do that one too. After all, '+FIELD' is literal expression[5]:

```
struct                      \ define the structure
  32 +field Firstname
  64 +field Lastname
   8 +field Initials
string Person               \ allocate the structure

s" Hans" Person -> Firstname place
s" JL"   Person -> Initials place
```

Well, if that isn't a complete implementation, I don't know what is..

[5] See section 7.13.

11.16 Unions

A union is a bunch of variables that share the same memory. Defining it in 4tH is quite easy:

```
include lib/ncoding.4th     \ include NCODING library

struct                      \ start defining the union
  32   /field               \ first field
  nell /field               \ second field
end-struct /union           \ end union definition

/union string myunion       \ allocate some space
                            \ now use it
s" Hans Bezemer" myunion place
1960 myunion n!
```

The union has the size of its *largest* field. You may wonder why the individual fields don't have their own names, but there is a reason for that. In other languages the field determines the type, so the compiler can figure out if you're using it properly. Since 4tH has no typechecking names are pretty useless, consequently there aren't any. You can use a union within another union or structure:

```
include lib/fp1.4th         \ include the FP library

struct                      \ define a union
  1 cells /field            \ sharing an integer
  float   /field            \ and a floating point
end-struct /number          \ number

struct                      \ define a structure
  field: type_tag           \ with some fields
  /number +field x          \ and the union concerned
array z                     \ allocate some space

23     z -> x !             \ use it as a number
23 s>f z -> x f!            \ use it as an FP number
```

You can also embed the union in the structure if you like:

```
include lib/fp1.4th         \ include the FP library

struct                      \ define a structure
  field: type_tag           \ with some fields
  struct                    \ define a union
    1 cells /field          \ sharing an integer
    float   /field          \ and an FP number
  +field x                  \ embed the union
array z                     \ allocate some space

23     z -> x !             \ use it as a number
23 s>f z -> x f!            \ use it as an FP number
```

If you really want to differentiate beween the floating point number and the integer, you can make an alias for field "X":

```
aka x y
```

And use it accordingly:

```
23     z -> x !            \ use it as a number
23 s>f z -> y f!           \ use it as an FP number
```

It is only cosmetic though, since it will compile to the same code and 4tH won't issue an error if you don't use it properly.

11.17 Complex control structures

Sometimes, the normal control structures of 4tH are not enough. Take this implementation of '-TRAILING':

```
: -trailing                   ( a n1 -- a n2)
  begin
    dup                       \ quit if length is zero
  while
    2dup 1- chars + c@ bl <>  \ is it still a space?
    if exit else 1- then      \ if not, quit
  repeat                      \ if so, decrement length
;
```

No one will tell you that this is elegant. You have to perform a test and quit the word. And this is still palatable. Imagine you have to test several conditions like this! It will become horrible pretty soon! Therefore, 4tH supports extended control structures. We've seen the basic control structures in sections 7.22, 7.23 and 7.24. Now we're expanding those into:

```
BEGIN .. WHILE .. WHILE .. AGAIN | REPEAT
BEGIN .. WHILE .. WHILE .. UNTIL
```

Yes, that's right: 'REPEAT' and 'AGAIN' are actually aliases. But what can we do with them? Well, take a look at our modified '-TRAILING' word:

```
: -trailing                   ( a n1 -- a n2)
  begin
    dup
  while                       \ quit if length is zero
    2dup 1- chars + c@ bl =
  while                       \ quit if it is not a space
    1-                        \ decrement length
  repeat
;
```

You have to admit that the latter version is much more elegant and readable. Although very elegant and usuable, in some circumstances you want to know which 'WHILE' terminated the loop and take additional action. This means you either have to set an additional flag or repeat the test, which is neither elegant nor efficient. For that purpose 4tH offers the DONE..DONE construct. In this example "-TRAILING" tells you why it terminated:

```
: -trailing                   ( a n1 -- a n2)
  begin
    dup
  while                       \ quit if length is zero
    done ." All spaces or NULL string" cr done
    2dup 1- chars + c@ bl =
  while                       \ quit if it is not a space
    1-                        \ decrement length
    done ." String trimmed" cr done
  repeat
;
```

A DONE..DONE clause *always* belongs to the last defined 'WHILE', so it doesn't really matter where you put it. What 'DONE' actually does is resolve the last defined 'WHILE' or 'DONE'. The last 'DONE' always jumps to 'REPEAT', so a single 'DONE' is a 'NOOP'. You can actually define something like this:

```
begin
  ( condition)
  while
    ( true)
    done ( false) done
    ( true)
    done ( false) done
    ( true)
    done
    ( true)
repeat
```

But you'll have to agree it's not very useful[6].

11.18 Optimization

The best way to optimize a program is to look out for certain patterns and obliterate them where possible, e.g.:

```
swap 2drop
```

Is a useless sequence of words that could easily be replaced by a single "2DROP". You don't need an optimizer to see that one. 4tH does very little optimization itself. One thing 4tH does for you is *tail call optimization*. That means that if the last instruction before the semicolon is a call to another word, it will change the 'CALL' instruction to a 'BRANCH' instruction. The advantage is twofold:

1. It will save you an expensive 'CALL' - 'EXIT' sequence;
2. It will save you Return Stack space.

Consequently, your programs will run a little bit faster[7]. If you want any further improvement, you will have to do that yourself. E.g. this will optimize a tail call optimization even more. Simply change:

```
: myfirstword ;
: mysecondword ;
: mythirdword if myfirstword else mysecondword then ;
```

Into this:

```
: myfirstword ;
: mysecondword ;
: mythirdword if myfirstword exit then mysecondword ;
```

If we decompile these programs we see the difference right away. The first compiles to:

[6] You can actually define "`if else else else then`" too, which is not very helpful neither.

[7] Special threading benchmarks show a speed increase of 40%, although real life programs will never see that much improvement.

```
[   0] branch    (1)
[   1] exit      (0)
[   2] branch    (3)
[   3] exit      (0)
[   4] branch    (9)
[   5] 0branch   (7)
[   6] call      (0)
[   7] branch    (8)
[   8] branch    (2)
[   9] exit      (0)
```

You can clearly see the tail call optimization at word 8. However, there is a 'BRANCH' at word 7 and an 'EXIT' at word 9 we can do without. The second version fixes this:

```
[   0] branch    (1)
[   1] exit      (0)
[   2] branch    (3)
[   3] exit      (0)
[   4] branch    (7)
[   5] 0branch   (6)
[   6] branch    (0)
[   7] branch    (2)
```

After the call at word 6 we immediately branch, which is correct because there is no code to execute after that. Note the tail optimizer was able to kick in *twice* here. Disadvantage of the second version is that it is less expressive, which means it is not completely clear why the programmer chose this construction.

In some rare circumstances, e.g. when you're calling user-defined words which manipulate the return stack[8], you *don't* want the tail optimizer to kick in. Suppressing it is very simple, just add the '[FORCE]' directive to the 'EXIT' or change the order of words slightly so that a reserved word is compiled at the end:

```
: return r> drop ;
: test dup if 1+ else return then [force] ;
0 test . cr
```

Another area of optimization is the calculation of constants. If we do it at compile time, we do not have to do it at runtime. Especially when it is in the middle of a loop or a word we frequently use. 4tH features a small peephole optimizer that tries to make the best of it. The rules are simple:

1. If you compile the word '+' or '-' and the previously compiled word was a LITERAL, a +LITERAL will be compiled *unless* rule 6 can be applied;
2. If you compile the word '*' and the previously compiled word was a LITERAL, a *LITERAL will be compiled *unless* rule 7 can be applied;
3. If you compile the word '/' and the previously compiled word was a LITERAL, a /LITERAL will be compiled *unless* rule 8 can be applied;
4. If you compile the word 'NEGATE' and the previously compiled word was a LITERAL, *LITERAL or /LITERAL, it will be negated;
5. If you compile the word '@' or '!' and the previously compiled word was a VARIABLE, a VALUE or TO will be compiled;

[8] E.g. NR> and N>R.

6. If you compile a +LITERAL and the previously compiled word was a LITERAL, +LITERAL or VARIABLE, the +LITERAL will added to it;
7. If you compile a *LITERAL and the previously compiled word was a LITERAL or *LITERAL, it will be multiplied;
8. If you compile a /LITERAL and the previously compiled word was a LITERAL it will be divided. If it was a /LITERAL, it will be multiplied;
9. If you compile a +LITERAL with the value 0 or a /LITERAL or *LITERAL with value 1, nothing is compiled at all;
10. If you compile a *LITERAL and the previously compiled word was a NEGATE, it will be replaced by a *LITERAL with a negative argument;
11. If you compile a /LITERAL and the previously compiled word was a NEGATE, it will be replaced by a /LITERAL with a negative argument;
12. If you compile a +LOOP with the value 1, LOOP is compiled;
13. If a 'THEN' or 'BEGIN' is compiled, the peephole optimizer is disabled[9] and previously compiled code will not be further optimized.

Let's see how that works in practice and examine this simple program:

```
-10 +constant 10-
100 begin 10- dup while 5 + dup . repeat
```

First we create a '+CONSTANT', which subtracts 10 from any number on the stack. Second, we set up a loop which starts at 100 and is subsequently decremented until it hits zero. 4tH will compile this code for you:

```
[   0] literal   (100)
[   1] +literal  (-10)
[   2] dup       (0)
[   3] 0branch   (7)
[   4] +literal  (5)
[   5] dup       (0)
[   6] .         (0)
[   7] branch    (0)
```

Note that although rule 2 seems to apply to the first two instructions, it is overruled by rule 4. With reason, because otherwise the LITERAL "90" would have been compiled which is certainly *not* what we meant. You can clearly see that the expression "5 +" has been condensed to a single +LITERAL, which saves an instruction. Now let's change it slightly and see what the peephole optimizer does:

```
-10 +constant 10-
100 begin 10- dup while 5 + 10- dup . repeat
```

We've added the '+CONSTANT' to the already optimized expression. This is what 4tH compiles:

[9] Although tail optimization will still partly work.

```
[   0] literal   (100)
[   1] +literal  (-10)
[   2] dup       (0)
[   3] 0branch   (7)
[   4] +literal  (-5)
[   5] dup       (0)
[   6] .         (0)
[   7] branch    (0)
```

At first it seems like it's identical, but it's not. The '+CONSTANT' has simply been subtracted from the expression! 4tH's peephole optimizer is not there to clean up your messy code, but to give you a helping hand where you really need it, e.g. consider this code:

```
struct
  1 +field operator
  2 +field operand
end-struct /instruction       \ define a simple structure
                              \ allocate some space
/instruction string instruction
                              \ now initialize it
1 instruction -> operator c!
5 instruction -> operand  c!
```

Which 4tH compiles to:

```
[   0] literal   (1)
[   1] literal   (768)
[   2] c!        (0)
[   3] literal   (5)
[   4] literal   (769)
[   5] c!        (0)
```

Without the peephole optimizer the addresses of the members of the structure would not have been calculated at compile time. Even more, optimizing this code by hand would not have been easy without writing some pretty murky source code. It is in these situations that the peephole optimizer excels.

11.19 Static variable pointers

The peephole optimizer is an integral part of 4tH and quickly and efficiently optimizes a whole range of expressions that can be resolved at runtime, including strings and variables. E.g. if you want to set a static pointer to a classic counted string, you could define it like this:

```
16 string str-len
str-len char+ constant str-string
```

That is because a 'STRING' expression can be optimized to a single literal. However, this is not the case with a variable and consequently this pig won't fly:

```
16 array array-len            \ won't compile!
array-len cell+ constant array-payload
```

It is for this kind of situation that we have 'EQUATES'. It takes a variable and an optional literal expression and create a static pointer that behaves like any other variable, yes, even including a ':REDO' or 'DOES>' definition[10]:

[10] See section 11.11.

```
variable <literal expression> EQUATES name
```

In its most basic form an 'EQUATES' expression is equal to 'AKA'[11]:

```
variable A
A equates B
```

Is equivalent to:

```
variable A
aka A B
```

Now we can rewrite our expression so it *can* be compiled properly:

```
16 array array-len            \ this will compile!
array-len cell+ equates array-payload
```

An 'EQUATES' expression will save you from writing this kind of code:

```
16 array array-len
: array-payload array-len cell+ ;
```

Which not only requires extra code, but will definitely run much slower.

11.20 Assertions

You have probably seen this before: you've made a program, compiled it and it doesn't work. Then you start putting code at strategic places, trying to pinpoint the error. And when you're finally done, you've got to revisit all of these places to remove that code. And you probably forget a few..

4tH has a built-in facility which allows to put that code there, debug your program and remove the debugging code from your program by changing a single line.

It is called "assertion" and those of you who have ever worked with C probably know what we're talking about.

An assertion is a line of code that will evaluate an expression. If the expression evaluates to false, it will exit the program with an error message. Let's take a look at this simple colon-definition:

```
: add                         \ expects two numbers on the stack
  +
;
```

If we call add by writing:

```
1 add
```

it will fail. Now we add this assertion:

[11] See section 11.10.

```
assert( depth 2 >= )
```

It will evaluate to false when there are less than two items on the stack. The program will be terminated and the appropriate error message will be issued. You may think that this is nice, but you still have to remove all assertion manually.

Not true! If you tried this out already you will see that you won't find an assertion anywhere. It's gone! True, if you want to use assertions you have to enable them. You do that with the word '[ASSERT]':

```
[assert]
: add
  assert( depth 2 >= )
  +
;

1 add
```

Now assertions will compile and work. If you remove the word '[ASSERT]' all assertions will disappear like they were comment. '[ASSERT]' works just like '[DECIMAL]', '[HEX]', etc. They work linear and do not follow the program flow. If you put '[ASSERT]' halfway your source-file you will notice that assertions work from that point:

```
: add
  assert( depth 2 >= )        \ assertions disabled
  +
;

[assert]                      \ enable assertions

: print-hex
  base @ >r hex
  assert( depth 1 >= )        \ assertions enabled
  . cr r> base !
;
```

Assertions are only enabled in "PRINT-HEX". The assertion inside "ADD" will be removed and thus be disabled. But there is more to '[ASSERT]' than the eye meets. It doesn't enable assertions, it toggles them. When the 4tH compiler starts, assertions are disabled. The first '[ASSERT]' enables them. A second '[ASSERT]' will disable them again:

```
[assert]                      \ enable assertions

: add
  assert( depth 2 >= )        \ assertions enabled
  +
;

[assert]                      \ disable assertions

: print-hex
  base @ >r hex
  assert( depth 1 >= )        \ assertions disabled
  . cr r> base !
;
```

There are many possibilities:

- You can start testing low level colon-definitions and move your way up to the high level definitions by moving '[ASSERT]' down.
- You can enable assertions on certain parts of your code by enclosing them with an '[ASSERT]' pair.
- You can switch the entire context of '[ASSERT]'s by adding a single '[ASSERT]' to the top of your source.

You are not limited to range-checking when using 'ASSERT('. Any expression that evaluates to TRUE is allowed:

```
[assert]
: add
  assert( ." ADD starts at " here . cr true )
  assert( depth 2 >= )
  assert( ." Values: " over over . . cr true )
  +
;
```

We're sure you can come up with more useful ideas. We did too.

11.21 Breakpoints

4tH also offers you the possibility to set breakpoints. It's quite easy to enable this facility. Just add this to the very beginning of your source:

```
[needs lib/debug.4th]
```

Setting a breakpoint is quite easy too, e.g. this piece of code malfunctions:

```
32 string argument
1 args argument place
```

Change it to:

```
32 string argument
1 args argument ~~ place
```

Now the breakpoint is enabled. It will enter a Forth-like shell just before 'PLACE' is executed. Now a host of words are at your disposal. You can examine any region of the Character Segment with "DUMP" or print any string variable with "TYPE". 4tHs internal variables and regions are known by name, like "PAD", "TIB", ">IN", "BASE" and "OUT". You can examine them or any other variable by using "?", "@" and ".".

You have a small calculator, that you can use to multiply, substract, add. You can change 'BASE' by executing "OCTAL", "HEX", "BINARY" or "DECIMAL". It also has a host of binary operators like "OR", "AND", "XOR", "INVERT", "LSHIFT" and "RSHIFT". It also has stack operators like "DUP", "DROP", "OVER" and "SWAP". "CLEAR" will clear the stack for you.

You can examine both stacks. ".S" will show you the data stack (including any rubbish you put there yourself during the debugging session) and "R.S" will show you the return stack. "DEPTH" and "RDEPTH" will tell you how many items there are on the stack. When

you're done, you may leave the debugger by typing "BYE". Your program will continue as usual.

A word of caution: since the debugger is a 4tH program itself, it doesn't actually freeze the virtual machine. It just *seems* like it is frozen. The contents of PAD may be slightly different than you expected. If you *really* need to examine the PAD as it was, don't examine it directly, but use "SPAD" to examine the "shadow PAD". "SPAD" leaves the address for the "shadow PAD" on the stack. The same goes for ">IN", "BASE" and "OUT": never examine these by address, but always by name. Although every effort has been made to catch any errors, some extreme stress tests might fail. It is not recommended to use the debugger when stack space is very tight.

11.22 Debugging

We've all been there. You've written a non-trivial program and when you try to run it, it bombs out with an error message. Then the question arises: "Where did I go wrong?" This proces is called "debugging". Debugging is very hard without any dedicated tools. Modern development systems include a debugger, featuring breakpoints, single stepping and variable tracking.

4tH features the following tools:

- Stack examination
- Assertions[12]
- Breakpoints[13]
- Decompilation
- Preprocessor
- Compiler directives

First of all, you need to understand how the 4tH compiler works. Initially the tokenizer collects all 4tH sources and breaks them down to tokens. A token is a word or a string. This makes it much easier for the compiler to parse the source[14]. 4tH has a single pass compiler, so after the tokenizer has finished successfully, it will start compiling right away. Even if the compiler encounters an error, it has actually compiled *something*. The point where the compiler stops gives you clues on where it went wrong. 4tH doesn't discard this partial compilant, but keeps it in memory so you can examine it. It won't allow any other action.

If you managed to successfully compile the source, it will pass the compilant to the virtual machine to execute it. Again, the point where the virtual machine stops is retained. Having one external editor window open with the 4tH source and another with 4tH itself is one the most comfortable ways to debug 4tH code.

If you get the dreaded I/O error 4tH is probably unable to locate an include file. You can use the preprocessor to find out where you went wrong. The preprocessor mimics the

[12] See section 11.20.

[13] See section 11.21.

[14] The tokenizer ignores the source layout, which means 4tH is able to read almost any ASCII file, regardless format. The downside of it all is that 4tH can't tell you on which line an error occurred - simply because it doesn't tokenize or compile on a line-by-line basis.

include sequence and leaves a 4tH file you can examine. You can also place an '[ABORT]' directive after the inclusion of each file to see when the I/O error pops up.

Tokenizer errors can also be produced by improperly formatted strings, like empty strings or missing delimiters. Highlighting editors like SciTE[15] allow you to find most of these pretty quickly. If not, you have to resort to the '[ABORT]' directive again because 4tH leaves no indication where you went wrong. The easiest way to track compilation errors is by decompiling the partial compilant, e.g.:

```
: one dup ." This is the address of TWO: " . cr drop ;
: two here one ;
: three 10 0 do [char] * emit loop cr ;
: four tree two tree ;
four
```

Doesn't compile. This will tell you where it went wrong:

```
[  11] literal   (10)
[  12] literal   (0)
[  13] do        (16)
[  14] literal   (42)
[  15] emit      (0)
[  16] loop      (13)
[  17] cr        (0)
[  18] exit      (0)
[  19] branch    (0)
```

It may be a bit hard to read, but you can clearly see it went wrong shortly after the 'LOOP'. Yes, "TREE" is misspelled. If you are unable to locate the error, again '[ABORT]' can help you out - but most of the time these drastic measures are not required.

If you suspect that runtime errors may occur in a certain piece of code, you can use assertions[16] to detect them. Although a well placed breakpoint[17] may help you out, it is annoying to stop and restart the program with each interation. For that reason a simple stack dump using ".S" may be more efficient:

```
**********
37 37 (TOS) This is the address of TWO: 37
(TOS) **********
```

".S" can be found in `anstools.4th`. Of course, you can always fallback to the ancient `printf()` practice. Note that you are still required to keep the stack balanced. If you don't you're even worse off.

11.23 Running 4tH programs from the Unix shell

If you're using Unix (which we highly recommend), you can run 4tH programs right from the Unix shell. All you have to do is to add one single line at the top:

```
#! /usr/lbin/4th cxq
." Hello world!" cr
```

[15] http://www.scintilla.org/SciTE.html
[16] See section 11.20.
[17] See section 11.21.

It indicates the way you normally compile and run a 4tH program, but without the filename, e.g.:

```
/usr/lbin/4th cxq hello.4th
```

In this case, you're using the classic 4tH compiler, which is located in the /usr/lbin directory. Note that you can add options if you want. The 'cxq' options tell the compiler to silently compile and execute a program.

Note this trick only works with 4tH sources, not compiled programs. You also have to flag the 4tH source as 'executable'. You can do that by issuing this command:

```
chmod 555 hello.4th
```

Now you can simply enter:

```
hello.4th
```

at the Unix prompt and your program will be compiled and executed. Don't worry about compromising the portability of your program. It will still compile and run happily under other Operating Systems, since '#!' is an alias for '\'. It only has a special meaning to the Unix shell.

11.24 Embedding 4tH programs in a batch file

If you're running a Microsoft Operating System like Windows or DOS[18], you can embed 4tH source code in an ordinary batch file[19]. All you have to do is to make the shell ignore the 4tH code, e.g.:

```
@goto exec
." Hello world!" cr
(
:exec
@4th cxq %0.bat %1 %2 %3 %4 %5 %6 %7 %8 %9
@rem )
```

Now save your file as "EXAMPLE.BAT" in the current working directory[20] and run it:

```
example
```

Don't add the ".BAT" extension or the whole thing won't work. 4tH will now automatically pick up the batch file and execute it. Well, how does it work?

It's simple: the shell silently jumps to the "EXEC" label and executes 4tH. 4tH will compile the batch file. It ignores the line that starts with '@GOTO', since '@GOTO' is an alias for '\'. It compiles anything up to the opening parenthesis, since that is the start of a multiline comment. The shell on its turn ignores the closing parenthesis, since that has been commented out by "@REM".

[18] DOS version 3.3 or higher.

[19] This method was taken from CSL, the "C Scripting Language". You can learn more about CSL at "http://csl.sourceforge.net".

[20] If you want to store it permanently in another directory, you may have to add additional path information.

Chapter 12

Standard libraries

12.1 Adding your own library

This is a lot easier than you might think! As a matter of fact, almost any program can be turned into a specialized library. A well-written program contains a lot of definitions and only *one* executable word. Take that word away and you've got a library!

A library may contain word defintions, variables, constants, almost anything you like. And a program that includes that library will have all these definitions at its disposal. As a matter of fact, the resulting program will behave like you entered the contents of the entire library file at the position of the '[NEEDS' directive, e.g. this is an except of `constant.4th`:

```
-1 constant NULL
```

When it is included in this file:

```
\ This is a sample table using NULL

[needs lib/constant.4th]

create sample
," First entry"
," Second entry"
," Third entry"
NULL ,
```

It will compile to the same code as this:

```
\ This is a sample table using NULL

-1 constant NULL

create sample
," First entry"
," Second entry"
," Third entry"
NULL ,
```

So it is *not* a good idea to make your library files too big, since there will be a lot of superfluous code included in the compilant which 4tH will *not* dispose of automatically.

You can nest '[NEEDS' directives, so one library file may include other library files. This helps to prevent duplicate code, which can be a serious maintenance problem. You can nest them as deep as you want, available memory being the only restriction.

However, when nesting inclusions you always have the problem of multiple inclusions. Don't think that all 4tH users know by heart which library files calls which. Multiple inclusions will lead to errors, unless you take precautions. We have '[DEFINED]' and '[UNDEFINED]' to prevent that:

```
[UNDEFINED] 2drop [IF]
: 2drop drop drop ;
: 2dup over over ;
: 2swap rot >r rot r> ;
[THEN]
```

If you have included this file before, '2DROP' is already defined, so in fact all definitions are skipped when the file is included for the second time. Of course, it will take up some extra memory, but at least it won't generate any errors.

You should *not* automatically resolve library dependencies when the file included:

- Is fairly large[1];
- Is required by many other include files[2];
- Has lots of (nested) dependencies itself.

In that case you should add a check whether a certain file has been loaded and if not, abort compilation, e.g.:

```
[UNDEFINED] F+ [IF] [ABORT] [THEN]
```

If you want to port your library file, it might be a good idea to hide specific 4tH constructions, e.g.:

```
[DEFINED] 4TH# [IF]           ( a n --)
  : string! chars + 0 swap c! ;
[ELSE]                        \ make an ASCIIZ string
  : string! swap 1- c! ;      ( a n --)
[THEN]                        \ make a counted string
```

Since '4TH#' is a 4tH specific constant, it will not be defined in other Forth compilers. This way the compiler will automatically select the correct definition. Finally, you may want to hide all words which you don't want exposed to the outside program:

```
[DEFINED] 4TH# [IF]
  hide buffer
  hide /buffer
[THEN]
```

The reason for that is twofold. First, you don't want the user of your library to meddle with the internal code or variables of your library. Second, if you leave all these words in, you're cluttering the symboltable, which may lead to compilation errors if the user happens to choose a name you already used inside your library.

[1] See sections 12.24 and 13.22.
[2] See sections 12.23 and 12.24.

Finally, you sometimes find yourself in a situation where you're faced with the choice to either make two library files with *slightly* different code or resort to conditonal compilation. If you choose the latter the question remains how the user of your library can switch between both versions. For those situations 4tH offers '[PRAGMA]'. '[PRAGMA]' simply compiles a constant with a TRUE value. E.g. `compare.4th` can be compiled in a case sensitive mode by issuing the `casesensitive` pragma:

```
[pragma] casesensitive
[needs lib/compare.4th]
```

In the library file itself you can treat the pragma just like any other constant:

```
[UNDEFINED] casesensitive [IF]
    dup [char] A - max-n and 26 < if bl or then
[THEN]
```

Where you place your library files is up to you. You can add them to the library files that come with 4tH, you can put them in another directory, whatever pleases you.

12.2 Adding templates

When you include a library file you add some words to your program. When you include a template you add some words to an existing program. That is the major difference between a library file and a template file. We've included a template with 4tH which allows you to create conversion program pretty quickly. The template is called "convert.4th" and it allows you to create a conversion program by defining just three words.

A standard conversion program takes an input file and creates an output file in a different format. When it can't open a file it will issue an error message, e.g.

```
Cant open input.txt
```

When you don't supply an input file and an output file, it will issue an error message e.g.:

```
Usage: myconversion input output
```

And of course, it will read and process the input file. And that's all you have to tell 4tH:

- The usage message
- How to read the file
- How to process the file

So, let's create a program that will convert a block file to a regular text file. How do we do that? First of all we've got to issue a usage message, like:

```
Usage: blk2txt blockfile textfile
```

Well, that is easy. If it comes to that we've got to abort the program, so this will do:

```
: Usage abort" Usage: blk2txt blockfile textfile" ;
```

Then we've got to read the file. A block file contains lines of 64 characters, always. So, we've got to create a buffer and read 64 characters. This will do:

```
64 string buffer
: Read-file buffer 64 accept ;
```

Finally, we've got to write the output file. Adding a 'CR' after typing the line will do, but we don't want any trailing spaces, so we need to strip those trailing spaces:

```
: Process buffer 64 -trailing type cr ;
```

Now we need to include the template and we're done:

```
[needs lib/convert.4th]
```

Wow! Do you know how much coding we need to do when we try to do this in C? This source code takes less than 256 bytes! Compile it and we're done! So how does it work? Well, the template expects us to define "Usage" and "Process". If you don't it will abort compilation:

```
\ Has Usage been defined? If not, abort!
[UNDEFINED] Usage [IF]
[ABORT] [THEN]

\ Has Process been defined?
[UNDEFINED] Process [IF]
[ABORT] [THEN]
```

If you don't define "Read-file" the template assumes that "Process" is completely self-contained, which means it will only open the files. Finally, you can optionally define "PreProcess" and "PostProcess" if you need any special actions at the beginning or the end:

```
: ProcessFile                          \ process the input file line by line
[DEFINED] PreProcess  [IF]
  PreProcess                           \ do any preprocessing
[THEN]
[DEFINED] Read-file   [IF]
  begin Read-file while Process repeat \ read file and process line or buffer
[ELSE]
  Process                              \ self contained processing
[THEN]
[DEFINED] PostProcess [IF]
  PostProcess                          \ do any postprocessing
[THEN]
;
```

If you don't define it, it won't include it. You can use such templates for many different programs, e.g. this will convert a Unix text file to an MS-DOS text file:

```
: Usage abort" Usage: udc infile outfile " ;
: Read-file refill ;
: Process 0 parse-word type 13 emit 10 emit ;
```

You can make them as sophisticated or as simple as you like. You can create other words as well, as long as those three words have been defined. Templates can be handled like any other library file. You can place them where you want, they can hold anything you want. Amaze your collegues by writing programs in a fraction of the time they should need!

12.3 Parsing the command line

The first thing that comes to mind is, of course, `getopts()`. 4tH offers a perfect equivalent of this C-function without copying it. The first thing you have to do is to include it:

```
include lib/getopts.4th
```

After that, you have to create a decision table containing the option characters and the actions that should be executed:

```
create myoptions
  char p , ' _print ,                  \ associate 'o' with _print
  char q , ' _quiet ,                  \ associate 'q' with _quiet
  char v , ' _verbose ,                \ associate 'v' with _verbose
  char f , ' _file ,                   \ associate 'f' with _file
  NULL ,
```

The stack diagram for an "option word" is very simple: assume nothing is on the data stack and leave nothing behind. If an option requires an argument, you can retrieve it by issuing "GET-ARGUMENT". Note these words have to be defined *before* the decision table:

```
256 string myfile                      \ allocate space for filename

false value (print)                    \ 'print' is off
false value (quiet)                    \ 'quiet' is off
false value (verbose)                  \ 'verbose' is off

: _print true to (print) ;             \ enable 'print'
: _quiet true to (quiet);              \ enable 'quiet'
: _verbose true to (verbose) ;         \ enable 'verbose'
: _file get-argument myfile place ;    \ set the filename
```

"GET-ARGUMENT" leaves an address/count string on the stack, which *has* to be consumed before returning. Finally, the decision table has to be passed to "GET-OPTIONS":

```
myoptions get-options
```

When "GET-OPTIONS" returns, it has also set a value called "OPTION-INDEX". This value is an index to the first non-option command line argument. Note that the 4tH program itself has an index of 0, so the first "real" command line argument has index 1. If "OPTION-INDEX" equals 'ARGN', there are no non-option command line arguments.

You can pass "OPTION-INDEX" to "ARG-OPEN", provided you've included it, e.g.:

```
include lib/getopts.4th
include lib/argopen.4th
( options table and option words)

myoptions get-options
option-index dup 2 + argn >
abort" Missing filename"

input over arg-open
output swap 1+ arg-open
```

You can also use `convert.4th` to handle all the details for you, but you *have to* name your decision table "OPTIONS" in order to make it work.

12.4 Mixing character and number data

Sometimes you have to mix character and number data, e.g. when you're porting a Forth program or when the need complex datastructures arises. Since 4tH gives each datatype its own segment this is not easy. However, there is a library that can help you. Let's have a look at this program:

```
16 constant /my               \ size of array
/my array my                  \ define array
0                             \ set up counter
begin
  dup dup                     \ duplicate counter
  cells my + !                \ store counter in array
  1+                          \ increment counter
  dup /my =                   \ limit reached?
until drop                    \ drop the counter

my                            \ set up index
begin
  dup @ . cr                  \ print the value
  cell+                       \ next element
  dup my /my cells + =        \ limit reached
until drop                    \ drop the index
```

This simple program defines a small array, fills and displays it. Now, this little thing does the same thing, but is located in the Character Segment:

```
include lib/ncoding.4th
                              \ load the library
16 constant /my               \ size of array
/my nells string my           \ define array
0                             \ set up counter
begin
  dup dup                     \ duplicate counter
  nells my + n!               \ store counter in array
  1+                          \ increment counter
  dup /my =                   \ limit reached?
until drop                    \ drop the counter

my                            \ set up index
begin
  dup n@ . cr                 \ print the value
  nell+                       \ next element
  dup my /my nells + =        \ limit reached
until drop                    \ drop the index
```

You see that the code is *very* similar. The 'STRING' declaration clearly indicates that the array is allocated in the Character Segment. But as you can see it is not an array of *cells*, but an array of *nells*. 'NELL' holds the size of a single nell, so we multiply it by the number of nells we want to get the proper size of the array. After that, it is just replacing the Integer Segment words with nell equivalents:

Note that although you can replace every cell with a nell, you *do* pay a penalty in execution speed, so use with caution.

NELL	CELL
/nell	/cell
nells	cells
n@	@
n!	!
nell+	cell+
nell-	cell-

Table 12.1: NELL equivalents

12.5 Dynamic memory allocation

If you don't know what this is, you probably shouldn't bother. Sometimes you don't know how much memory you will actually need, sometimes you know how much you need, but you won't need it during the entire execution of the program. In these cases, you can temporarily allocate a chunk of memory and release it when you no longer need it.

4tH has similar facilities. E.g. if you want to allocate 600 bytes, you simply include `ansmem.4th` and allocate it:

```
[needs lib/ansmem.4th]
600 allocate
```

'ALLOCATE' leaves two items on the stack. The first one is a flag. If it is true, memory allocation has failed, so we can easily add some error checking to our little program:

```
[needs lib/ansmem.4th]
600 allocate abort" Out of memory"
```

It it returns false, memory has been allocated. Its address is the second item on the stack. You can pretty much do what you want with it, but remember that memory is usually allocated in the Character Segment, so if you want to store numbers as well over there, read section 12.4 again. Anyway, this is completely valid:

```
[needs lib/ansmem.4th]
600 allocate abort" Out of memory"
s" Hello temporary world!" rot place
```

Let's change that one a little bit to prove we've actually stored anything:

```
[needs lib/ansmem.4th]
600 allocate abort" Out of memory"
>r s" Hello temporary world!" \ Let's save the address
r@ place                      \ Now store the string
r> count type cr              \ Let's print the string
```

Let's allocate another 100 bytes and free all memory afterwards:

```
[needs lib/ansmem.4th]
600 allocate
abort" Out of memory" >r      \ First allocation
s" Hello temporary world!"
r@ place                      \ Now store the string
r@ count type cr              \ Let's print the string
```

```
100 allocate
abort" Out of memory" >r      \ Second allocation
s" I'm a little crammed!"
r@ place                      \ Store another string
r@ count type cr              \ Let's print the string

r> free
abort" Cannot free memory"    \ Now free the first block
r> free
abort" Cannot free memory"    \ Now free the second block
```

Yes, that's right: you feed 'FREE' the address that 'ALLOCATE' returned and it returns a flag. If it is a true flag, an error occurred; if not, everything is hunky dory. Let's try to free it twice:

```
[needs lib/ansmem.4th]
600 allocate
abort" Out of memory" >r      \ First allocation
s" Hello temporary world!"
r@ place                      \ Now store the string
r@ count type cr              \ Let's print the string

r@ free
abort" First attempt"         \ Now let's free the block
r> free
abort" Second attempt"        \ And try to free it again..
```

Yes, now 4tH terminated with the error message "Second attempt". You can not free a block twice..! But you can reallocate it if you happen to change your mind. You can increase or decrease its size, without losing any data. When the new block is too small to hold all the data, the data is truncated. Let's see it in action:

```
[needs lib/ansmem.4th]
50 allocate
abort" Out of memory" >r      \ First allocation
s" Hello temporary world!"
r@ place                      \ Now store the string
r@ count type cr              \ Let's print the string

r> 100 resize
abort" Out of memory" >r      \ Now resize the block
r@ count type cr              \ Here is your string again

r> free
abort" Cannot free memory"    \ Now free it
```

You'll see that your precious string is still alright. Apart from a flag, 'RESIZE' also returns the address of the reallocated block. If 'RESIZE' fails, *your original data is still alright*, so in some circumstances you might want to save the old address.

12.6 Tweaking dynamic memory

You might find that 4tH doesn't reserve much memory for dynamic allocation. Dynamic memory is allocated on the heap, which is 16 kB. You can increase it, but first you have to know how dynamic memory works. You can determine how much memory has been allocated by using the word 'ALLOCATED':

```
[needs lib/ansmem.4th]
50 allocate
abort" Out of memory" >r      \ First allocation

r@ . ." allocates "
r@ allocated . ." bytes." cr

r> free
abort" Cannot free memory"    \ Now free it
```

And it will print something like:

```
768 allocates 64 bytes.
```

64 bytes? I thought we allocated 50 bytes! Let's try another one:

```
[needs lib/ansmem.4th]
500 allocate
abort" Out of memory" >r      \ First allocation

r@ . ." allocates "
r@ allocated . ." bytes." cr

r> free
abort" Cannot free memory"    \ Now free it
```

This time it prints something like:

```
768 allocates 512 bytes.
```

As a matter of fact, 'ALLOCATED' will *always* return multiples of 32 bytes. That is a consequence of how 4tH handles dynamic memory. 4tH divides dynamic memory into *paragraphs*. When you allocate memory, 4tH allocates as much paragraphs as it needs to provide you with the memory you requested. Then these paragraphs are marked as 'taken'. This marking is done in the Heap Allocation Table, which is located in the Integer Segment. Every paragraph is represented by a cell in the HAT.

You can fine-tune this mechanism by defining some constants before including `ansmem.4th`. This will create a heap with 512 paragraphs of 256 bytes, which is 128 kB:

```
512 constant #paragraph       \ 512 paragraphs
256 constant /paragraph       \ each paragraph is 256 bytes
[needs lib/ansmem.4th]
500 allocate
abort" Out of memory" >r      \ First allocation

r@ . ." allocates "
r@ allocated . ." bytes." cr

r> free
abort" Cannot free memory"    \ Now free it
```

Try to keep the number of paragraphs *low*. 1024 seems like a nice upper limit. If you need that much memory, it is *much* better to handle it in larger chunks. This avoids fragmentation and keeps the time to search the HAT within acceptable limits.

If you're using dynamic memory to manage e.g. linked lists, you can force a heap of cells by defining the pragma `forcecellheap` *before* you include `ansmem.4th`. You can even use the node structure definition to make it match a single paragraph, e.g.:

```
[pragma] forcecellheap        \ force a CELL heap

struct                        \ define the node
  field: kind
  field: o1
  field: o2
  field: o3
  field: val
end-struct /paragraph         \ name it "/paragraph"

include lib/ansmem.4th        \ leave the library to it
```

You can now be assured that the size of a paragraph equals a single node. Neat, huh?

12.7 Using two heaps

Due to 4tH's Harvard architecture[3], you cannot mix cells and characters. Of course, you can use a library to store numeric information[4], but sometimes that is not sufficient. Fortunately, 4tH offers another library that allows you to have *two* heaps, one for your cells and one for your strings. Well, your first question probably is: how can I differentiate between the two? It's easy: if you combine the two you allocate cells with "ALLOCATE", "FREE", "RESIZE", just as usual. If you want to allocate strings, you use "CALLOCATE", "CFREE" and "CRESIZE".

In order to achieve that you have to include the cell library *before* the string library:

```
include lib/memcell.4th
include lib/memchar.4th
```

If you don't, `memchar.4th` will default to your usual "ALLOCATE", "FREE" and "RESIZE" words. You can also determine the size of both heaps by setting a constant before including them. The size is in address units:

```
2048 constant /heap           \ a 2K cell heap
include lib/memcell.4th
8192 constant /heap           \ an 8K string heap
include lib/memchar.4th
```

Note the pragma `forcecellheap` is automatically set when you include `memcell.4th`, so you can use that one in your programs as well. Both libraries are drop-in replacements of `ansmem.4th` (for their respective datatype). So why two different implementations? It largely depends what your program needs. If you require:

- A small implementation;
- Fairly robust;
- A few large allocations or a limited number of allocations of a similar size.

You'll be well off with `ansmem.4th`. However, if you require:

- An unknown number of allocations of different sizes;
- A combination of a cell heap and a string heap.

You'll be better off with `memcell.4th` and `memchar.4th`.

[3] http://en.wikipedia.org/wiki/Harvard_architecture
[4] See section 12.4.

12.8 Garbage collection

4tH features a simple form of garbage collection. The only thing between ”real” garbage collection and this implementation is that you remain in control at all times. First you have to decide what kind of datatype you want to put under control of the garbage collector by defining a pragma. This will enable cell garbage collection:

```
[pragma] forcecellgc
include lib/gc.4th
```

If you don’t define this pragma, garbage collection will default to characters. Since the garbage collector uses the same libraries we discussed in section 12.7, you can size them with ”/HEAP”. Next, you have to start garbage collection:

```
start-gc
```

It’s time to start a ”scope”:

```
start-scope >r
```

This will return a single cell scope token which is used to keep track of when to do garbage collection. Now the fun starts: allocate and resize away all you like, but instead of using the usual allocation words, you use these:

```
50 gc-alloc >r
s" Hello temporary world!"
r@ place                        \ Now store the string
r@ count type cr                \ Let's print the string

r> 100 gc-resize
count type cr                   \ Here is your string again
```

Wow, did you see that?! We lost a pointer in the last line! Well, that isn’t too bad in this case - we’re using a garbage collector after all. Note you may open up new scopes if you like, but you have to close them in the right order, so you may want to use the data stack to keep track of your scope tokens, like we did.

Closing a scope is easy. Just pass the scope token to ”END-SCOPE” and *all* memory you allocated since the last ”START-SCOPE” will be automatically freed:

```
r> end-scope
```

When you’ve ended a scope, the garbage collection will automatically revert to the previous scope, so any allocation you do *after* closing a scope will be handled by the *previous* scope. Finally, when you’re done you can shutdown the entire garbage collector by issuing:

```
end-gc
```

This will free up all garbage collection support structures, which are also dynamically allocated. Famous last words: do *not* free or resize memory allocated with ”GC- ALLOC” or ”GC-RESIZE” with the standard words ”FREE” or ”RESIZE”. Ignore this warning at your own peril.

12.9 Application stacks

Did you ever feel like a second return stack would be nice? Well, you can. As a matter of fact you can have several dedicated stacks. It's quite easy to use:

```
[needs lib/stack.4th]

16 array mystack              \ allocate some space
mystack stack                 \ convert it into a stack

234 mystack >a                \ push 234 on the stack
456 mystack >a                \ push 456 on the stack
mystack a@ . cr               \ examine top of stack
mystack a>                    \ pop 456 from the stack
mystack a>                    \ pop 234 from the stack
. . cr                        \ show the values
```

Note there is a floating point[5] version as well, called `fstack.4th`. It works in a similar way. Wouldn't it be nice to have a string stack too? Yes, 4tH provides that one too! It works the same way:

```
[needs lib/stsstack.4th]

1024 constant /mystack
string mystack                \ allocate some space
mystack /mystack string-stack \ convert it to a string stack

s" Hello" mystack >s          \ push string 'Hello' on stack
s" World" mystack >s          \ push string 'World' on stack
mystack s@ type cr            \ examine top of stack
mystack s>                    \ pop 'World' from the stack
mystack s>                    \ pop 'Hello' from the stack
type cr type cr               \ show the values
```

But if you only need a single stack - but one you can monitor - there is another library member:

```
2048 constant /cstack
[needs lib/stmstack.4th]

s.clear                       \ clear the string stack

s" Hello" >s                  \ push string 'Hello' on stack
s" World" >s                  \ push string 'World' on stack
s@ type cr                    \ examine top of stack
s>                            \ pop 'World' from the stack
s>                            \ pop 'Hello' from the stack
type cr type cr               \ show the values
s.error CSE.NOERRS = . cr     \ show the exit status
```

You can set the size of the string stack by defining "/CSTACK" before including it. "S.CLEAR" clears the stack and resets "S.ERROR". "S.ERROR" is simply a value you can query or set according to your needs. If you cause an underflow or an overflow, the integrity of the stack is maintained.

Note there is a catch: when you've popped a string from the string stack, the string itself is untouched, so the address-count pair is still valid. However, if you push another string onto the same stack, the popped string is clobbered - so a kind of 'SWAP' is out of the question. There is another way to create one (or more) string stack without these disadvantages, called `strstack.4th`, but it is larger and slower. It works the same way as the first library member we presented.

[5] See sections 12.24 and 12.23.

12.10 Local variables

I have to admit something: I *really* don't like local variables. They offer a kind of "random access" that goes totally against everything the stack-based Forth language stands for. And they don't come for free - you pay a performance penalty for setting them up and setting them free. That's why 4tH doesn't offer any native support.

However, sometimes you just don't want to rewrite a routine, but just use it. And then it's handy when *some* support is there. So here we are. Let's start with an example from gForth:

```
: my.max { n1 n2 -- n3 }
  n1 n2 > if
    n1
  else
    n2
  endif ;
```

So, what's happening here? First you see something that *resembles* a stack diagram - which it isn't: it's executable code. Note it uses accolades, *not* parentheses. Now, what does it do? It assigns the top of the stack to "n2" and the second on stack to "n1". That's pretty neat, because if it wouldn't you'd have to read the whole thing backwards - this *really* follows the stack diagram. Everything after "–" is comment, though.

When the first accolade is executed it builds a frame of local variables, so yes, it also works with recursion. The rest you can read as if this were executed:

```
0 value n1                    \ define a value n1
0 value n2                    \ define a value n2

: my.max ( n1 n2 -- n3 )      \ no accolades here
  to n2 to n1                 \ assign TOS to n2
  n1 n2 > if                  \ assign 2OS to n1
    n1                        \ compare them
  else                        \ and then select
    n2                        \ either n1 or n2
  then ;
```

We already told you that a local frame has to be released - and it doesn't. So what's happening here? Well, gForth *modified* the execution semantics of ';', so it *secretly* checks whether locals were used and released them if they did. Every locals scheme I've seen does it. In 4tH, you can't do that, because the compiler won't let you. The consequence is, you'll have to do it yourself. Now let's make a 4tH version out of this:

```
include lib/locals.4th

: my.max   ( n1:A n2:B -- n)
  2 locals
    :A @ :B @ > if
      :A
    else
      :B
    then @
  end-locals ;
```

This does the very same thing. First, there are no accolades. Second, you only define there are two locals - and that's it. 4tH locals *always* bear the same name. You can't choose them. Third, they are variables, not values. Fourth, you are the master of the frame - not the compiler.

When you issue ”LOCALS”, it takes a value from the stack and allocates a locals frame of that size (in this case a frame of 2 local variables). Then it takes *that same number* of values from the stack and places them into the frame. So let’s call it:

```
1 10 my.max . cr
```

It works just like the gForth version, so ”1” ends up in ”:A” and ”10” ends up in ”:B” (hint: that’s also what the stack diagram tells you). The rest is pretty much like the gForth version - except that ”:A” and ”:B” are *variables* of course. And what if you need more local variables? Well, you can go up to *eight*, from ”:A” to ”:H”.

Finally, we reach the end of our word and there it is, the word that releases the frame, ”END-LOCALS”. If you fail to use it, the local variables stay *in scope*, so they remain valid in the caller - and believe me: that’s not what you want. If you leave a word prematurely by using ’EXIT’, you’ll have to terminate the scope manually as well, so instead of e.g.:

```
if exit then
```

You’ll have to write:

```
if end-locals exit then
```

Don’t forget! This way you can do almost anything other local variable schemes can do, e.g. take this one, also from gForth:

```
: strcmp { addr1 u1 addr2 u2 -- n }
  addr1 addr2
  u1 u2 min 0
  ?do { s1 s2 }
    s1 c@ s2 c@ -
    ?dup-if
      unloop exit
    then
    s1 char+ s2 char+
  loop
  2drop
  u1 u2 - ;
```

This one introduces a new scope *within* a **?DO..LOOP**. So how do we tackle this one? Well, like this:

```
: strcmp   ( a1:A n1:B a2:C n2:D -- n )
  4 locals
    :A @ :C @
    :B @ :D @ min 0
    ?do
      2 locals
        :A @ c@ :B @ c@ - dup
        if                      \ we don’t have ?DUP
          end-locals            \ discard LOOP locals
          unloop                \ discard LOOP parameters
          end-locals            \ discard strcmp locals
          exit                  \ exit the routine
        else                    \ we don’t have ?DUP
          drop                  \ so we have to drop it
        then
        :A @ char+ :B @ char+
      end-locals                \ free LOOP locals
    loop
    2drop
    :B @ :D @ -
  end-locals ;
```

True, it's much more "wordy", but it does the same thing word for word. It's just not that sneaky. First both strings (in ANS Forth fashion) are assigned to locals variables. Since both strings need two values on the stack, we need 4 local variables. The top of the stack (the count of the second string) is put into ":A", the address of the second string ends up in ":B", the count of the first string ends up in ":C" and finally the address of the first string ends up in ":D". No surprises here.

You may also follow how they are used. The length of the smallest string is the loop parameter of the **?DO..LOOP**. After that follow the addresses of both strings. When the **?DO..LOOP** is entered, a new stack frame is created for these addresses. Consequently, when the loop is terminated, the scope has to be terminated as well. That's why you find an "END-LOCALS" just before 'LOOP'. Of course, gForth does this automatically.

And here is the catch: in the middle of the loop we leave the word *entirely* and return to the caller. At that moment we not only have to deal with the scope of the word, but the scope of the loop as well. So first we release the locals frame of the loop (just before 'UNLOOP') and then the locals frame of the word (just before 'EXIT'). Like I said, gForth does this as well - you just don't see it.

There is just one more catch: locals are not taken into consideration when you use 'CATCH'[6]. The fix is easy, though. There is a word called "LFRAME", which is a pointer to the current locals frame. So if you want to preserve the sanity of your locals subsystem, you simply do this:

```
lframe >r catch r> over       \ duplicate CATCH value
if to lframe else drop then   \ CATCH value still on stack
```

What this does is save the address of your locals frame on the return stack (where it is supposed to be save) and restore it after an exception has been caught. If there was no exception at all we can simply discard it. You can even wrap the whole thing in a definition if you want - and be done with it.

It is the most "userfriendly" locals scheme? Probably not, but it is just as versatile and a real life saver if you need it. And - believe it or not - completely portable to gForth and other Forth compilers.

Note that there is a floating point[7] version as well, called `flocals.4th`. It works in a similar way.

12.11 Random numbers

If you want to program a game or a simulation, you'll probably need random number generation. Of course, you can do that too with 4tH. Just include the proper file:

```
include lib/random.4th
```

And initialize it:

```
randomize
```

[6] See section 11.5.
[7] See sections 12.24 and 12.23.

It generates a number between 0 and "MAX-RAND", but we'll teach you how to generate a number for virtually any range below that.

There are two things important when you want to do that: the range limit and the lower limit. Say, we want to simulate a dice with numbers in a range from 1 to 6. The lower limit is 1. We subtract that from the upper limit to get the range limit.

So: upper limit minus lower limit gives a range limit of five (6 - 1 = 5). The general formula looks like this:

*<range-limit> 1+ random * max-rand 1+ / <lower limit> +*

When we apply this to the dice-example, the complete formula is:

```
5 1+ random * max-rand 1+ / 1 +
```

This will give you a dice-simulation, that produces random numbers between 1 and 6. If you don't like to write this code yourself, you can always include `choose.4th`. After including that you can simply get a predetermined range of number by simply writing:

```
include lib/choose.4th
6 choose 1+
```

Which is in fact your perfect dice simulation, because it returns a number between 1 and 6. "CHOOSE" in itself provides a number between 0 and 5 in this example.

Sometimes you want the randomizer to provide a preset range of numbers. There is an variable called "SEED" which determines what range of numbers is provided. "RANDOMIZE" initializes it with the setting of the internal clock, which is the same value that 'TIME' returns. Yes, that means that if you call "RANDOMIZE" twice within a second, the randomizer will return the same range of numbers. Presetting the randomizer is pretty trivial:

```
1234567890 seed !
```

Finally, under the hood it is not a single randomizer, but two! Default a modified[8] BSD algorithm is used, but you can also default to the standard Micrsoft algorithm by defining the appropriate pragma:

```
[pragma] MS-random
include choose.4th
```

You can also select the required version by calling it by name:

```
MSRandom . BSDRandom . cr
```

Are these randomizers very good? No, as a matter of fact, they are not. They are sufficient to power your occasional game, but for more serious applications they simply fall short. That's why 4tH also provides some very heavy duty randomizers, designed by randomizing expert George Marsaglia[9]. The first is `gmkiss.4th`. Note that although it provides a similar interface as `random.4th`, it is *not* a drop-in replacement, most significantly because it can return the full range of a cell value, including negative numbers. Another difference is that it takes *four* cell size values to initialize:

RANDOMIZER	REGISTERS	TYPE
CONG	x	congruential
SHR3	y	3-shift shift register
MWC	z, w	multiply with carry

Table 12.2: gmkiss randomizers and their registers

```
1234567890 987654321 12893467 1029384756 seed4!
```

As a matter of fact, it is built out of *three different* randomizers, which feed a fourth one. You can call each and every one of these randomizers and set their registers individually.

All those randomizers combined results in the KISS random number generator, which is the randomizer that "RANDOM" calls[10].

If that still isn't enough we got even a better one, `gmskiss.4th`, which is arguably one of the best randomizers in the world. This one uses a large amount of memory, so don't use it for trivial tasks. It can return the full range of a cell value (including negative numbers) and takes three cell size values to initialize:

```
include lib/gmskiss.4th
987654321 12893467 1029384756 seed3!
." Random number: " kiss . cr
```

If you don't want to set the seed yourself, you can always call "RANDOMIZE" instead. This "super KISS" implementation differs from the previous one because the "multiply with carry" generator has now superlong periods ($54767 * 2^{1337279}$).

But hold, that is not everything. There are a *ton* of different randomizers in 4tH. Table 12.3 gives you an overview. Some of them comply by the standard API[11], some of them are so specialized that they don't. So whatever your needs are, there is always a randomizer that fits them.

12.12 Sorting

Yes, 4tH can do that too. You just have to include `qsort.4th` (a "quick sort" implementation) to make it all possible. It works pretty much like the sort routines you've seen in C, which means you have to devise a word to compare two values. Note that `qsort.4th` can only sort *cell arrays*. Setting it up is pretty simple. First you have to include it:

```
include lib/qsort.4th
include lib/random.4th
```

Then you have to create a word that returns a true flag when the second value on the stack is smaller than the top of the stack. In this example we will just compare two integers, so that is pretty easy:

```
: MyPrecedes < ;
```

[8] The standard BSD algorithm is notoriously flawed, especially the lower bytes of the returned random number.
[9] http://en.wikipedia.org/wiki/George_Marsaglia
[10] Unless you've included `random.4th` as well.
[11] They provide `random`, `randomize` and `MAX-RAND`.

`qsort.4th` creates a deferred word[12] called "PRECEDES". Now we have to assign our word to "PRECEDES", so that it is executed when "PRECEDES" is called:

```
' MyPrecedes is Precedes
```

That's it! We're ready to rock 'n roll now. Let's set up a simple testing environment:

```
10 constant #elements
#elements array elements

randomize

: InitElements #elements 0 do random elements i th ! loop ;
: ShowElements #elements 0 do elements i th @ . loop cr ;
```

This creates an array of ten elements, which is filled with random values by "InitElements". "ShowElements" will show on screen what is stored there. The actual sort is straightforward: tell "SORT" which array and how many elements there are to sort and you're done:

```
: SortElements elements #elements sort ;
```

Now let's put it all together:

```
InitElements
ShowElements
SortElements
ShowElements
```

It will initialize the array, show its contents, sort it and show it again. It will output something like this:

```
12717 6028 1389 31870 14234 15884 31062 14788 18186 149
149 1389 6028 12717 14234 14788 15884 18186 31062 31870
```

And what if these were string addresses? Well, "SORT" would have sorted them too, from the lowest addresses up to the highest addresses, but that's probably not what you meant. You wanted to sort the actual strings, not just their addresses. Can 4tH do that too? Sure, you just got to create another "PRECEDES" word. Something like this:

```
: SPrecedes >R COUNT R> COUNT COMPARE 0< ;
```

This will take the two values and treat them as strings. Now the strings are sorted, not just the addresses. Note that the strings themselves will not move in memory. The *pointers* move, the strings themselves don't. This is a so-called "address based sort".

The 4tH library contains several sorting algorithms. There are two families: "index based" and "address based". The first family can sort *any* array[13], the second one only cell arrays, because fetching and exchanging is done within the library member itself.

Another difference is that the "index based" family allows you to combine several different sorting algorithms within the same program, but also *requires* you to define your own element exchange word. You can also combine *one* member of the "address based" family with the "index based" family, as long as you handle your "PRECEDES" definitions properly since they are not compatible.

Apart from their ”family” characteristics each and every algorithm has its own strengths and weaknesses. The larger library members are usually faster, but the table gives you more accurate timings. The factor ”Speed” gives you an indication of how fast they are. 1.0 is the reference speed, higher numbers are proportionally slower[14]. The same for the ”Size” factor - higher numbers are proportionally bigger.

Which one is optimal for your specific problem depends on a lot of things, e.g. the state and condition of the data (like partially sorted sequences, duplicates) are important factors in choosing the right sorting algorithm. There more than adequate resources for you on the web to find out, but here are a few ”rules of the thumb”:

- If you have only a few records to sort, take Simple sort (improved) or Selection sort;
- If you have a significant number of records, need a reasonable, consistent performance and a compact routine, take Circle sort;
- If you have a significant number of records and need a good performance, take Shell sort or Comb sort;
- If you have a large number of records and absolutely need the fastest performance, take Quick sort or Intro sort;
- If you have a large number of records and need a very consistent, good performance, take Intro sort or Heap sort.

If you’re still unsure and size or speed isn’t a major issue, you can’t go wrong with Heap sort.

12.13 Bitfields

Bitfields are always a bit of a controversial subject, since many complain that they are not portable or simply not worth the effort. Anyway, there is a library included if you should ever need them. In its most basic form they are quite simple: you provide the number and a certain number of bits are extracted from a certain position:

```
include lib/bitfield.4th      \ include the library

[binary] 00011000 [decimal]   \ put a number on the stack
3 2 bn@ .                     \ start at bit 3, 2 bits long
```

This will return ”3” - or ”11” binary, if you prefer. Counting starts at the rightmost bit (which is bit 0) so this expression returns bits 3 and 4. Saving it is very easy as well, just add a new value:

```
include lib/bitfield.4th      \ include the library
                              \ store ”2” into bit 3 and 4
2 [binary] 00011000 [decimal] 3 2 bn!
```

Of course, you can also store a pattern into a string or cell variable:

[12] See section 11.7.
[13] Since it works with indexes, not addresses. Consequently, it is a bit slower in general.
[14] For a 100,000 element cell array

```
include lib/bitfield.4th      \ include the library

variable myvar                \ define a variable

0 myvar !                     \ initialize it
3 myvar 3 2 b!                \ store "3" into bit 3 and 4
myvar 3 2 b@ . cr             \ print the bitfield
```

Its character counterparts are named "BC@" and "BC!", which may not be all that surprising.

Some may argue that these are not true bitfields, because you always have to provide the starting bit and the number of bits. With a bit of effort though we can make these bitfields behave like the real thing. First, include `constant.4th` in addition to the bitfield library:

```
include lib/bitfield.4th      \ include the library
include lib/constant.4th      \ include "cell-bits"
```

Let's define the fields in the usual "starting bit, length in bits" form and add the expression "CELL-BITS * +". After that, turn the whole thing into a constant. Just make sure the bitfields don't overlap:

```
0 3 cell-bits * + constant ink    does> cell-bits /mod ;
3 3 cell-bits * + constant paper  does> cell-bits /mod ;
6 1 cell-bits * + constant bright does> cell-bits /mod ;
7 1 cell-bits * + constant flash  does> cell-bits /mod ;
```

This will scale the bitfield "length" parameter in such a way, that it stays arithmetically separated from the "starting bit" parameter. Finally, wrap it into a simple 'DOES>' definition. This will split the constant into the two components again when it's used. Now let's use it in a real world example:

```
0 enum black  enum     blue   \ enumerate all colors
  enum red    enum     magenta
  enum green  enum     cyan
  enum yellow constant white

24 constant /lines            \ length of the display
32 constant /rows             \ width of the display
                              \ allocate display buffer
/lines /rows * buffer: attributes
  does> swap rot /rows * + chars + ;
                              \ make it a two dim. array
white 5 6 attributes -> ink    bc!
  red 5 6 attributes -> paper  bc!
 true 5 6 attributes -> bright bc!
false 5 6 attributes -> flash  bc!
                              \ set colors at line 5, row 6
```

This program will set the colors in the display buffer of a Sinclair ZX Spectrum. As you can see for yourself, the bitfields behave as you would expect. Only defining them is a little clumsier - nothing a preprocessor can't fix.

12.14 Bit arrays

A bit array (also known as a bitmap, a bitset, or a bitstring) is an array data structure that compactly stores individual bits (boolean values). Setting it up is a bit awkward, but using it is very simple. First we have to include the libary:

```
include lib/bitarray.4th
```

Then we have to set up the bit array itself. The *minimal* size of a bit array is one cell. The *total* size of a bit array has to be a multiple of cells. Although you can use cells to size the array, the most portable way is to use bits (128 in this example, to be exact):

```
128 cell-bits / array mybits
```

Then we have to declare the execution semantics. Note from that moment on, the array is no longer accessable as a cell array. So if you want to initialize the array, the easiest way is to do it *before* that, otherwise you'll have to do it bit by bit:

```
:redo mybits bit-array ;
```

Or even shorter:

```
128 cell-bits / array mybits does> bit-array ;
```

Believe it or not, but that was the hardest part! Manipulating a bit array is dead easy. Bits are numbered zero and up. To set a bit simply issue:

```
0 mybits bit-on
```

Resetting or toggling a bit is no rocket science either:

```
34 mybits bit-off
63 mybits bit-toggle
```

Of course, you can query the status of a bit. If the bit is set it returns true, otherwise false:

```
34 mybits bit?
```

Because of their compactness, bit arrays have a number of applications in areas where space or efficiency is at a premium. Most commonly, they are used to represent a simple group of boolean flags or an ordered sequence of boolean values. However, bit arrays can also be used for the allocation of memory pages, inodes, disk sectors, etc. In such cases, the term "bitmap" may be used[15].

12.15 Associative arrays (using hash tables)

Associative arrays have been made popular by languages such as Python and PHP. In general they are not very efficient, but make it easy to associate keys with their respective values. 4tH does not have native support for them, but if you want them, they are there.

The first one is small, but limited. First you need to include `hash.4th`, because this implementation uses hash tables:

```
include lib/hash.4th
```

[15] Not to be confused with raster images, which is something entirely different.

Second, you allocate an integer array in which you want to store the keys and initialize it:

```
16 constant /myhash
/myhash array myhash
myhash /myhash hashtable
```

Third, you need to select the hashing algorithm you want to use. ”SDBM” is a decent one, but you can use another one if you want or even write your own:

```
' sdbm is hash
```

Now we’re ready to rock ’n roll! Before we save a value, it is wise to see if its key is free:

```
s" beta" myhash get?
```

”GET?” returns two values. First a flag. If it is non-zero, the key is free. The second one is the value associated with the key. When you’re testing (like now) it is useless, so we can drop it:

```
0= abort" Key taken!" drop
```

Now we can do some business:

```
512 s" beta" myhash put
```

And if we want to retrieve the value we simply write:

```
s" beta" myhash get . cr
```

Well, it’s small, simple and painless, but there is one little catch. What will you do when a collision[16] occurs? Well, frankly that’s up to you. The program does not define any alternative action. You could try to increase the size of the hash table - but not while the program is still running of course. You could also define an alternative hash table, but does that really make life easier?

At least you’ve learned two lessons:

1. Hash tables are not very memory efficient;
2. Hash tables need to take alternative action when a collision occurs.

Of course, there is a version which addresses the latter problem - and uses even *more* memory. In order to resolve a collision, memory is divided into buckets. A ”bucket” is a certain number of associated memory locations where the values of keys with the same hashes are stored. E.g. of you make buckets of eight, you can survive seven collisions before the program gives up. Buckets are four entries deep by default, but you can override that:

```
3 constant /bucket
include lib/hashbuck.4th
```

[16] A ”collision” means a key is already taken.

Now we have to allocate the hash table. The best strategy is to provide space for *many* keys with *small* buckets. In this example we provide space for fifteen keys:

```
include lib/hash.4th
15 /bucket 2 * * 1+ constant /myhash
/myhash array myhash
```

Note that every entry in a bucket takes *two* cells and *one* extra cell is needed for the hash table itself. In this example, we have room for fifteen keys, each having buckets of three entries, each entry taking two cells - and one cell overhead. The rest looks familiar:

```
myhash /myhash hashtable
' sdbm is hash
512 s" beta" myhash put
s" beta" myhash get . cr
```

Note we don't have to test the key ourselves, the program itself transparently handles any collisions. And what when we are out of buckets? Well, the program will issue an error message and quit. If you want to see for yourself if a bucket was full, you need to use "PUT? which returns a non-zero flag if a bucket was full:

```
512 s" beta" myhash put? abort" I want to issue my own!"
```

Finally, there is a version that has *none* of these drawbacks. It is more memory efficient and you don't have to worry about collisions or buckets. But as always, there is a price to pay: it is slower, since it looks up the keys with a binary search. The overhead is negligible, just two cells. So this:

```
include lib/hashkey.4th
include lib/hash.4th

32 constant /myhash
/myhash array myhash
```

Will get you 15 solid key and associated value pairs. Again, the rest looks familiar:

```
myhash /myhash hashtable
' sdbm is hash
512 s" beta" myhash put
s" beta" myhash get . cr
```

You want to store strings instead of numbers? You can do that too if you use `userpad.4th`. This library member stores strings in a circular buffer, much like the internal PAD[17]. After that, storing strings is easy. You only need to define these words:

```
: sput >r 2>r >pad drop 2r> r> put ;
: sget get error? abort" Bad bucket" count ;
```

And this is where it all comes together:

```
s" This is the end" s" beta" myhash sput
s" beta" myhash sget type cr
```

But is it worth all the trouble? In some circumstances where quick access to large tables is required, may be, but you be the judge of that.

[17] Which means they *will* get overwritten at some point.

12.16 Associative arrays (using binary trees)

Sure, it takes much more code and requires dynamic memory, but it has none of the drawbacks associated with hashtable based implementations. Just include the library, define a variable and initialize it:

```
include lib/bintrkv.4th
variable kv
NULL kv !
```

And off you go!

```
15 5 kv put
100 6 kv put
5 kv get .
6 kv get . cr
```

Of course you can define additional arrays or reassign a value:

```
variable mykv
NULL mykv !

15 5 mykv put
16 5 kv put
```

You can also use strings as key values:

```
25 s" alpha" kv put$
35 s" beta" kv put$

s" alpha" kv get$ .
s" beta" kv get$ . cr
```

Don't worry about cleaning up dynamic memory:

```
kv @ destroy-tree
```

That will take care of all that, but of course you have to do it for each and every associative array you created.

12.17 Lookup tables with integer keys

No CASE construct, huh? Now how are we supposed to make those complex decisions? Well, do it the proper way. Leo Brodie wrote: "I consider the case statement an elegant solution to a misguided problem: attempting an algorithmic expression of what is more aptly described in a decision table". And that is exactly what we are going to teach you.

Let's say we want a routine that takes a number and then prints the appropriate month. In ANS-Forth, you could do that this way:

```
: Get-Month
  case
     1 of ."  January " endof
     2 of ." February " endof
     3 of ."   March  " endof
     4 of ."   April  " endof
     5 of ."    May   " endof
     6 of ."   June   " endof
     7 of ."   July   " endof
     8 of ."  August  " endof
     9 of ." September" endof
    10 of ."  October " endof
    11 of ." November " endof
    12 of ." December " endof
  endcase
  cr
;
```

This takes a lot of code and a lot of comparing. In this case (little wordplay) you would be better of with an indexed table, like this:

```
create MonthTable
  ,"  January "
  ," February "
  ,"   March  "
  ,"   April  "
  ,"    May   "
  ,"   June   "
  ,"   July   "
  ,"  August  "
  ," September"
  ,"  October "
  ," November "
  ," December "

: Get-Month                    ( n -- )
  12 min 1- cells MonthTable + @c count type cr
;
```

Which does the very same thing and will certainly work faster. True, you can't do that this easily in ANS-Forth, but in 4tH you can, so use it! The word ',"' compiles a string, whose address can be retrieved by '@C' as if it were a numeric constant. Note that '@C' just returns the *address* of the string, so you have to use 'COUNT' to obtain an address/count pair. Of course, there is also an equivalent to ',"' called ',|'. The latter is delimited by a bar instead of a quote, but essentially works the same way.

But can you use the same method when you're working with a random set of values like "2, 1, 3, 12, 5, 6, 4, 7, 11, 8, 10, 9". Yes, you can. But you need a special routine to access such a table. Of course we designed one for you. It is called "ROW" and you can use it by adding this directive:

```
[needs lib/row.4th]
```

This routine takes four values. The first one is the value you want to search. The second is the address of the table you want to search. The third is the number of fields this table has. And on top of the stack you'll find the excecution token of the word that searches the table. Two have been predefined, "NUM-KEY" searches numeric values and "STRING-KEY" searches string values.

This routine can search zero-terminated tables. That means the last value in the index field must be zero. Finally, it can only lookup positive values. It returns the value you searched,

the address of the row where it was found and a flag. If the flag is false, the value was not found.

Now, how do we apply this to our month table? First, we have to redefine it:

```
create MonthTable
  1 ,  ,"  January "
  2 ,  ," February "
  3 ,  ,"   March  "
  4 ,  ,"   April  "
  5 ,  ,"    May   "
  6 ,  ,"   June   "
  7 ,  ,"   July   "
  8 ,  ,"  August  "
  9 ,  ," September"
  10 , ,"  October "
  11 , ," November "
  12 , ," December "
  NULL ,
```

The first field must be the "index" field. It contains the values which have to be compared. That field has number zero. Note that this table is sorted, but that doesn't matter. It would work just as well when it was unsorted. Let's get our stuff together: the address of the table is "MonthTable", it has two fields and we want to return the address of the string, which is located in field 1. Field 0 contains the values we want to compare. We can now define a routine which searches our table:

```
: Search-Month                ( n1 -- n2 f)
  MonthTable 2 num-key row    \ search the table
  dup >r                      \ save flag
  if nip cell+ @c else drop then
  r>                          \ if found get value
;                             \ if not drop address
```

Because "ROW" is able to search integer tables and string tables, you have to define which one it is by using either `num-key` or `string-key`. Now, we define a new "Get-Month" routine:

```
: Get-Month                   ( n --)
  Search-Month                \ search table
  if                          \ if month is found
    count type                \ print its name
  else                        \ if month is not found
    drop ." Not found"        \ drop value
  then                        \ and show message
  cr
;
```

Is this flexible? Oh, you bet! We can extend the table with ease:

```
3 Constant #MonthFields

create MonthTable
  1 ,  ,"  January " 31 ,
  2 ,  ," February " 28 ,
  3 ,  ,"   March  " 31 ,
  4 ,  ,"   April  " 30 ,
  5 ,  ,"    May   " 31 ,
  6 ,  ,"   June   " 30 ,
  7 ,  ,"   July   " 31 ,
  8 ,  ,"  August  " 31 ,
```

```
  9 ,  ," September" 30 ,
  10 , ,"  October " 31 ,
  11 , ," November " 30 ,
  12 , ," December " 31 ,
  NULL ,
```

Now we make a slight modification to "Search-Month":

```
: Search-Month                ( n1 -- n2 f)
  MonthTable #MonthFields num-key row
  dup >r                      \ search table, save flag
  if nip cell+ @c else drop then
  r>                          \ if found get value
;                             \ if not drop address
```

This enables us to add more fields without ever having to modify "SearchMonth" again. If we add another field, we just have to modify "#MonthFields". We can now even add another routine, which enables us to retrieve the number of days in a month:

```
: Search-#Days                ( n1 -- n2 f)
  MonthTable #MonthFields num-key row
  dup >r                      \ search table, save flag
  if nip cell+ cell+ @c else drop then
  r>                          \ if found get value
;                             \ if not drop address
```

Of course, there is room for even more optimization, but for now we leave it at that. Do you now understand why 4tH doesn't have a CASE construct?

12.18 Lookup tables with string keys

But what if the table we're using looks like this:

```
create MonthTable
  ," January" 31 ,
  ," February" 28 ,
  ," March" 31 ,
  ," April" 30 ,
  ," May" 31 ,
  ," June" 30 ,
  ," July" 31 ,
  ," August" 31 ,
  ," September" 30 ,
  ," October" 31 ,
  ," November" 30 ,
  ," December" 31 ,
  NULL ,
```

Sure, 4tH compiled some kind of integer value there, but an address to a string is less than helpful. We have to compare *strings* in order to find the correct entry, not *addresses*. So, we need a word that searches the table and returns the contents of the field that follows the appropriate string. Well, of course there is such a word. It is "ROW" again. You can use it by entering:

```
[needs lib/row.4th]
```

At the *beginning* of your program. "ROW" takes an address/count pair of the string that has to be found, the address of the table it has to search for that string and the number of fields the table has. It returns the original address/count pair of the string, the address of the row where the search stopped and a flag. That makes it quite a useful word, e.g. how many days has June:

```
: GetDays                      ( a n --)
  MonthTable 2 string-key row \ search the table
  if
    cell+ @c . drop drop       \ if found, display the number of days
  else                         \ else an error message
    drop type ."  is not a month!"
  then
  cr
;

s" June" GetDays
```

If "ROW" returns true, the value was found. If it returns false, it wasn't. Note you have to indicate which datatype "ROW" has to deal with. "ROW" is quite versatile, but that is not the only merit of "ROW" as we will see in the next sections.

12.19 Lookup tables with multiple keys

Some tables have multiple keys to search them, e.g. by name or by number. So far all tables we've seen dealt with a single key in the first column. It would be a shame if you had to split a table into two tables, simply because you had two different ways to access it. Fortunately, "ROW" can handle this kind of tables as well as long as you put the key columns up front and add a NULL at the end of the table for every key, e.g.

```
create mytable
  ," Monday"      1 ,
  ," Tuesday"     2 ,
  ," Wednesday"   3 ,
  ," Thursday"    4 ,
  ," Friday"      5 ,
  ," Saturday"    6 ,
  ," Sunday"      7 ,
  NULL , NULL ,
```

This table consists only of key fields. You can search for the name and get a number or search for the number and get the equivalent name. The trick is to keep in mind what the key field is and the relative position of the datafield. In this case we want to search on number, so the corresponding name is the field *before* the key field. When you start the search the pointer you pass to "ROW" has to point to the key field you want to search. In this case that is equivalent to:

```
mytable cell+
```

Let's assume we want to search this table both ways:

```
: day>num                      ( a1 n1 -- a1 n1 -f | n2 f)
  mytable 2 string-key row dup >r
  if nip nip cell+ @c else drop then r>
;
```

```
: num>day                     ( n1 -- n1 -f | a1 n2 f)
  mytable cell+ 2 num-key row dup >r
  if nip cell- @c count else drop then r>
;
```

The first word doesn't hold any surprises. It is a vanilla search word. The second one passes a slightly modified pointer to "ROW" and decrements the address it returns, so it now points to the name field. We can use both words quite easily and transparently:

```
s" Friday" day>num if . else type ."  not found" then cr
s" New yearsday" day>num if . else type ."  not found" then cr

5 num>day if type else . ." not found" then cr
8 num>day if type else . ." not found" then cr
```

You will see they work as expected.

12.20 Lookup tables with duplicate keys

Although most tables come with unique keys you may find yourself in a situation where you have to resume a search. "ROW" can handle that situation as well. Let's examine this table:

```
create people
  ," Ritchie"  ," Lionel"
  ," Dijkstra" ," Edsger"
  ," Moore"    ," Henri"
  ," Ritchie"  ," Dennis"
  ," Wirth"    ," Nick"
  ," Hopper"   ," Grace"
  ," Moore"    ," Chuck"
  ," Hopper"   ," Dennis"
  NULL ,
```

There is one key field, since the table is terminated with only one NULL. We also find multiple Hoppers and Moores, so we can't be sure we've found the right one right away. In order to get that one we might have to continue our search. That is exactly what this program does:

```
: >surname 2 string-key row ; ( a n x1 -- a n x2 f)
: first? rot cell+ @c count compare 0= ;
: >next cell+ cell+ >surname ;
: .name type space type ;     ( a1 n1 a2 n2 --)

: >first                      ( a1 n1 x a2 n2 --)
  2>r                         ( a n x)
  if                          ( a n x)
    dup 2r@ first?            ( a n x f)
    if                        ( a n x)
      drop ." Found " 2r> .name cr
    else                      ( a n x)
      >next 2r> recurse       ( a n x a n)
    then                      ( --)
  else                        ( a n x)
    drop 2r> .name ."  not found" cr
  then                        ( --)
;
```

```
: >name -rot 2>r >surname 2r> >first ;

: demo
  s" Ritchie" s" Dennis" people >name
  s" Moore" s" Chuck" people >name
  s" Lovelace" s" Ada" people >name
;

demo                          \ run the demo
```

">NAME" is a wrapper around this programs most important words ">SURNAME" and ">FIRST". ">SURNAME" simply searches for a given surname in the table and returns its address. ">FIRST" takes over and compares the first name. If it checks out we're done, if not it calls ">NEXT". ">NEXT" increments the pointer, so it now points to the next row. Then it calls ">SURNAME" again, effectively continuing the search. Finally ">FIRST" calls itself to check the first name again. Depending on the contents of the table, this process can be repeated several times.

12.21 Fixed point calculation

We already learned that if we can't calculate it in dollars, we can calculate it in cents. And still present the result in dollars using pictured numeric output:

```
: currency <# # # [char] . hold #s [char] $ hold #> type cr ;
```

In this case, this:

```
200012 currency
```

Will print this:

```
$2000.12
```

Well, that may be a relief for the bookkeepers, but what about us scientists? You can do the very same trick. We have converted some Forth code for you that gives you very accurate results. You can use routines like SIN, COS and SQRT. A small example:

```
[needs lib/math.4th]

45 sin . cr
```

You will get "7071", because the result is multiplied by 10000. You can correct this the same way you did with the dollars: just print the number in the right format. You can also use a delightful little library created by Leo Brodie called "fraction.4th". This library allows you to do arithmetic in the range of -13.1072 and 13.1071 with a precision of 0.0001! It seamlessly integrates with the "math.4th" library:

```
[needs lib/math.4th]
[needs lib/fraction.4th]

45 sin s>v v. cr
```

This one will actually print:

```
0.7071
```

Of course, you can also use it with other math words, as long as they are properly scaled. The scaling constant is called '10K' and is available from "constant.4th". The nice thing about fractions is that you can do all basic math (within its range, of course) without bothering about the decimal point. To convert a fixed point number to a fraction, you have to multiply it by '10K', which is 10,000 and call 'S>V', e.g. in order to convert '0.7071' to a fraction you need to do this:

```
7071 s>v
```

You can also define a fraction yourself, e.g. this converts two-thirds to a fraction:

```
2 3 v/
```

You can add or subtract fractions, e.g.:

```
7071 s>v 2 3 v/ - v. cr
```

This is equivalent to:

```
PRINT 0.7071 - 0.6666
```

You can multiply or divide fractions, e.g.:

```
7071 s>v 2 3 v/ v* v. cr
```

This is equivalent to:

```
PRINT 0.7071 * 0.6666
```

Since a fraction is still a single-cell number, you can manipulate the stack in the usual way. In some cases you can even mix cells and fractions, as the following table will show you:

With the word 'V>S' you can convert a fraction back to a '10K' scaled fixed point number. You can print a fraction using the word 'V.'.

Another example is SQRT from "math.4th". If you enter a number of which the root is an integer, you will get a correct answer. You don't even need a special formatting routine. If you enter any other number, it will return only the integer part. You can fix this by scaling the number.

However, scaling it by 10 will get you nowhere, since "3" is the square root of "9", but "30" is not the square root of "90". In that case, we have to square the scale itself; we need 100, 10,000 or even 1,000,000 to get a correct answer. In order to retrieve the next digit of the square root of "650", we have to multiply it by 100, which is the square of 10:

```
[needs lib/math.4th]

: .fp <# # [char] . hold #S #> type cr ;
650 100 * sqrt .fp
```

Which will print:

```
25.4
```

To acquire greater precision we have to scale it up even further, like 10,000. This will show us, that "25.49" brings us even closer to the correct answer. If you want to use the ”fraction.4th” library, you will have to scale it to the square of ’10K’, which means you’re restricted to numbers upto ”21.475”:

```
include lib/math.4th
include lib/fraction.4th

10K 10K * constant 100M

21 100M * sqrt s>v v. cr
```

All these words and more are included in the library file ”math.4th”. You might not find the word you actually need, because we are not much of a mathematician. When we encounter new routines they will be added to 4tH. We would appreciate your input!

12.22 Double numbers

C’mon, indulge me, run this program:

```
max-n . cr
```

You will probably see some fairly large number displayed on your screen. What is it? Well, it is the largest number that can fit in a cell. Larger numbers and 4tH will start to behave erratically:

```
max-n 1+ . cr
```

Still, it is large enough to do the accounting for a reasonably sized enterprise. But it hasn’t been always like that. Early Forths could barely handle the accounting of an average schoolboy. In order to get some real work done they had to expand the range somehow. And if one cell isn’t enough you simply take two cells. That is what double numbers are all about: they are numbers that are composed of two cells.

The problem is that Forths operators aren’t overloaded. If you try to add up two double numbers with ’+’ you will end up with one double number and the addition of the two parts of the first double number. So in order to add up two double numbers, a separate word had to be defined. If you need a special word for addition, you will also need one for multiplication, subtraction, division and negation.

It is no secret that Charles Moore, the inventor of Forth, thought that double numbers had become superfluous after the introduction of modern processors and modern Forth compilers. That is one of the reasons that 4tH doesn’t have a native double word implementation. But should you need this vastly expanded range for one reason or another 4tH allows you to enter the murky world of double, unsigned and mixed numbers.

So, how does it work. First of all, if you need a full implementation you have to include these two libraries:

```
include lib/todbl.4th          \ double number input
include lib/dbldot.4th         \ double number output
```

Then you probably need some variables. But hey, if double numbers take up two cells, you can't use ordinary variables. That's true. You will need small arrays:

```
2 array dVar1                  \ double variable one
2 array dVar2                  \ double variable two
```

And here comes the next problem. How do you enter double numbers? Depending on the size of the number, you can use two approaches. The easiest one is to convert a single number to a double number with "U>D". The only catch is this only works for *positive* numbers. If you want to enter a *negative* number, you have to negate it afterwards:

```
500 u>d 60000 u>d d+           \ add 500 and 60000
2dup d. cr                     \ print the double number
dVar1 2!                       \ store it in variable one
```

Yes, every single operation has a *double* counterpart:

SINGLE	DOUBLE
+	d+
negate	dnegate
.	d.
2/	d2/
max	dmax
min	dmin
dup	2dup
@	2@
!	2!

Table 12.7: Examples of single and double number counterparts

But what if you want to enter a very large number right away? In that case you will have to convert a string to a double number with "S>DOUBLE[18]".

```
s" 5000000000" s>double        \ convert a string to double
2drop 2dup d. cr               \ print the double number
dVar1 2@ dmax dVar2 2!         \ save the largest in variable two
```

Finally, when you have done all you needed to do and you're left with a number that is small enough you can convert it back to a single number with "D>U". Note this will only work for positive numbers:

```
dVar2 2@ dVar1 2@ d-           \ subtract both double variables
2dup d. cr                     \ print the double number
d2/ d2/ d>u . cr               \ divide by 4 and convert to unsigned
```

Rule of the thumb is: stay away from double numbers if you can. It is slow, cumbersome and error-prone. If you can't, goodnight and good luck!

[18]Note it will bomb out when the string isn't a double number. If you want to play safe, use ">DOUBLE". There is also a single number ">NUMBER" word. In that case you can select the double number version by using its alias ">DNUMBER".

12.23 Floating point numbers (unified stack)

Warning!
This is *really* complex stuff, I *cannot* guarantee that it functions flawlessly. You may *lose accuracy* or get the *wrong result*. **Don't use any of this for any real life applications.**

Ok, if you *really, really, really* want it, 4tH also provides floating point number support. If you only need the basic operations and are willing to settle for limited accuracy and error checking, you should try `zenfloat.4th`. It is small and very easy to use. Just include it[19] and go right ahead:

```
include lib/zenfloat.4th
include lib/zenfpio.4th
314159265 -8 f. cr
```

That will print the first eight decimals of "pi". Just read it like "314159265e-8" or in laymans terms "314159265" with eight places after the decimal point. Of course you can use positive exponents as well. Because ZEN floating point numbers are stored as two numbers[20] on the datastack[21], you can use '2DUP', '2DROP' and '2SWAP' to manipulate them. This is the way to calculate the surface of a circle with a radius of 10.55:

```
include lib/zenfloat.4th
include lib/zenfpio.4th
314159265 -8 1055 -2
2dup f* f* f. cr
```

Which will happily print:

```
349.667115
```

You can also convert a number to a float and back:

```
1960 s>f 2dup f. f>s . cr
```

If you try to convert a floating point number that is bigger than 'MAX-N' to a single number it doesn't work of course. You can store a floating point number in a variable if you want:

```
include lib/zenfloat.4th
include lib/zenfpio.4th

2 array pi

314159265 -8 pi 2!
```

If you want to write ANS Forth compatible code, you can. Just include `zenans.4th` just after `zenfloat.4th`. It will allow you to write code like this:

[19] You *always* have to include `zenfloat.4th` manually before you include any other floating point library member.

[20] The exponent on the TOS, the mantissa on the 20S.

[21] Consequently, this implementation uses what ANS Forth calls a "shared floating point stack". Since 2008, ZEN float is considered a non-standard implementation (see: http://www.forth200x.org/fp-stack.html).

```
[DEFINED] 4TH# [IF]           \ if this is 4tH
include lib/zenfloat.4th      \ include the ZEN fp library
include lib/zenfpio.4th       \ include the ZEN fp I/O library
include lib/zenans.4th        \ make Zen ANS compatible
include lib/zenfsin.4th       \ include the SIN library
[ELSE]                        \ if this is ANS Forth
s" easy.4th" included         \ load the compatibility layer
[THEN]

FLOAT array fVar              \ a floating point variable
FLOAT array Pi                \ a FP variable holding PI
                              \ store PI in the variable
cr s" 314159265e-8" s>float Pi f!
Pi f@ 4 s>f f/                \ get PI and calculate sine
fsin fdup f. cr fVar f!       \ print the result and save it
fVar f@ f. cr                 \ print the variable
```

And this is the way it is run:

```
habe@linux-471m:~/Forth> gforth
Gforth 0.6.2, Copyright (C) 1995-2003 Free Software Foundation, Inc.
Gforth comes with ABSOLUTELY NO WARRANTY; for details type `license'
Type `bye' to exit
s" fpdemo.4th" included
( Lots of messages you can ignore )
0.707106780551956
0.707106780551956
 ok
bye
habe@linux-471m:~/Forth> 4th cxq fpdemo.4th
0.707106778
0.707106778
habe@linux-471m:~/Forth>
```

Apart from some rounding errors they are identical. Note that this "S>FLOAT" implementation is quite basic[22], but you can feed it most floating point numbers without any problems. If you use `zenans.4th`, most of the constructions in section 12.24 will work with `zenfloat.4th` as well.

The error handling of `zenfloat.4th` is *very* basic. If you try to get the square root of a negative number, it will just stop with an appropriate message. There are lots of floating point routines included[23], just check the glossary for more details.

12.24 Floating point numbers (separate stack)

Warning!
This is *really* complex stuff, I *cannot* guarantee that it functions flawlessly. You may *lose accuracy* or get the *wrong result*. **Don't use any of this for any real life applications.**

If you want something more sophisticated than `zenfloat.4th` you should try `ansfloat.4th`. This library is a full ANS-Forth implementation. In `ansfloat.4th` mantissas are double-cell unsigned. Single-cell exponents contain the mantissa's sign and a

[22] There is also a full ANS Forth implementation available: `fpin.4th` and `fpout.4th`. If you want to use these instead, remove `zenfpio.4th` and include them after you've included `zenans.4th`.

[23] All compatible library members are prefixed with '`zen*`'. See also chapter 23.2.

signed exponent. The format used is non-standard for simplicity. Exponents are almost a whole cell wide, leading to a wider dynamic range than most IEEE formats. Range and digits of precision versus cell width are listed below.

CELL WIDTH	DIGITS OF PRECISION	RANGE = 10^{-x} to 10^{x}
32	18	$x = 323196289$
64	37	$x = 1.3881175 * 10^{18}$
n	$INT((n-1)*0.602)$	$x = 0.301 * (2^{n-2})$

Table 12.8: Range and digits of precision

Floating point arithmetic is easy to use, but be careful to watch where your inaccuracies are coming from. Floating point numbers are *approximations*. You can lose up to half a bit of precision in each operation. Differences between large numbers can be trouble spots. If you need transcendental functions, they can often be done in integer arithmetic since you don't need floating point's run-time auto-scaling. Floating point arithmetic uses a dedicated, shallow stack[24]. There is no depth checking, so underflows and overflows may occur unless error checking is added. Since floating point support is implemented in high level 4tH it is also rather *slow* and *big*.

So, how does it work. First of all, if you want to use floating point, you *always* need to include this library *before* any other floating point library:

```
include lib/ansfloat.4th      \ floating point words
```

This will also create the floating point stack. If you want a larger stack, just create this constant accordingly *before* including the library:

```
32 CONSTANT FLOATING-STACK    \ size of float stack
include lib/ansfloat.4th      \ floating point words
```

This will create a stack of 32 floating point items. You probably want to do some I/O, so let's take care of that one too:

```
32 CONSTANT FLOATING-STACK    \ size of float stack
include lib/ansfloat.4th      \ floating point words
include lib/ansfpio.4th       \ floating point I/O words
```

Then you probably need some variables. But hey, if floating point numbers take up more than a cell, you can't use ordinary variables. That's true. You will need small arrays:

```
FLOAT array fVar1             \ double variable one
FLOAT array fVar2             \ double variable two
```

Finally, you have to initialize the library and specify the precision[25]:

```
fclear                        \ initialize library
8 set-precision               \ set precision to eight
```

And here comes the next problem. How do you enter floating point numbers? Depending on the size of the number, you can use two approaches. The easiest one is to convert a single number to a floating point number with "S>F".

[24] ANS Forth defines a six item stack. A larger FP stack is an environmental dependency.
[25] The number of significant digits used by F.

```
500 s>f 60000 s>f f+          \ add 500 and 60000
fdup f. cr                    \ print the floating point number
fVar1 f!                      \ store it in variable one
```

Note the double numbers library is loaded by default as well, so ”D>F” is available too. Every single operation has a *floating point* counterpart:

SINGLE	FLOAT
+	f+
/	f/
*	f*
negate	fnegate
.	f.
max	fmax
min	fmin
dup	fdup
@	f@
!	f!

Table 12.9: Examples of single and floating point number counterparts

But what if you want to enter a floating point number right away? In that case you will have to convert a string to a double number with ”>FLOAT”.

```
s" 5000.575" >float           \ convert a string to a float
drop fdup f. cr               \ drop the flag and print the number
fVar1 f@ fmax fVar2 f!        \ save the largest in variable two
```

A drawback of defining floating point constants that way is that ”>FLOAT” is notoriously slow. However, there is a library that enables you to enter floating point numbers the ”mantissa + exponent” way[26]:

```
include lib/ansfloat.4th      \ include the fp library
include lib/ansfpio.4th       \ include the fp i/o library
include lib/fpow10.th         \ include the F10** library

  5000572 -3 me>f f. cr
314159265 -8 me>f f. cr
```

Defining constants this way is about *four times faster*. This is especially useful when you need a constant in a tight loop. Note that the exponent must be in the range of -24 to 24, otherwise you won't get the correct result.

Most ANS-Forth floating point words are available, although these words usually have their own library file. For technical reasons they will *not* automatically include the floating point number library for you, but abort instead if it is *not* loaded. So if you want to calculate the sine of 45 degrees, you have to do this:

```
include lib/ansfloat.4th      \ include the fp library
include lib/ansfpio.4th       \ include the fp i/o library
include lib/fsinfcos.4th      \ include the library

fclear                        \ clear the fp stack
8 set-precision               \ set precision to eight

pi 4 s>f f/ fsin f. cr        \ calculate the sine
```

[26] See section 12.23.

Since a full circle (360 degrees) requires 2 times PI, we have to divide it by 4 to get the equivalent in radians[27]. But what if you make an error? What if a conversion overflows, you divide by zero or try to get the square root of a negative number? There is where "FERROR" comes in. It is a variable that holds the last floating point error that occurred. Here you got an example:

```
include lib/ansfloat.4th      \ include the library
include lib/ansfpio.4th       \ include the fp i/o library

fclear                        \ clear the fp stack
8 set-precision               \ set precision to eight
1 s>f 0 s>f f/                \ divide by zero
ferror ? cr                   \ examine FERROR
```

In this case, "FERROR" returns two. But what does that mean? Well, here you got a handy table of all IEEE 754 exceptions. In 4tH, these are predefined constants.

CODE	MEANING
FE.NOERRORS	No error
FE.OVERFLOW	FP overflow
FE.UNDERFLOW	FP underflow
FE.DIVBYZERO	Division by zero
FE.INEXACT	Inexact result
FE.INVALID	Invalid operation

Table 12.10: IEEE 754 FP math errors

You can simply clear any errors by invoking "FCLEAR" again. Note that this clears your floating point stack too, so you have to start all over again. And no, 'THROW' and 'CATCH' do *not* restore your floating point stack. No easy recovery here!

12.25 Floating point functions

ANS-Forth floating point comes with a large number of libraries, sometimes two or more for the same mathematical function. Table 12.11 may help you to select the proper one for your specific problem.

Note several ANS-Forth floating point libraries are used within other libraries. If a library is marked "default", it is the default that will be selected for inclusion if you haven't included another library already. Most of the time you can override the default selection by simply explicitly including it, e.g. instead of:

```
include lib/ansfloat.4th
include lib/ansfpio.4th
include lib/fexpint.4th
```

You could set up the program like this:

```
include lib/ansfloat.4th
include lib/ansfpio.4th
include lib/fexpt.4th
include lib/flnflogb.4th
include lib/fexpint.4th
```

[27] "FSIN" takes its parameter in radians.

In some cases a library member *may* assume a certain dependency is already resolved and bomb out. In that case, you'll have to figure out yourself which dependency that is and resolve it manually. If that happens, you may find chapter 23.2 quite useful.

Note that Zen float comes with lots of floating point functions as well, but it lacks the more specialized functions like numerical integration, Bessel, Riemann Zeta, elliptic integrals, etc.

12.26 Floating point configurations

In order to simplify setting up floating point support, there are five predefined configurations. The only thing you have to do is to include a single file. All configurations support basic floating point operations and floating point I/O. You can see in the table which configuration meets your particular needs.

For your information: precision is definitely *not* the same thing as accuracy. If you're still not sure what to choose configuration 2 (`fp2.4th`) is a safe bet. Note that "FLOAT" is always defined when having included any of the configurations, which enables you to check whether floating point support has been loaded, e.g.:

```
[UNDEFINED] float [IF] [ABORT] [THEN]
```

If you need to differentiate between Zen floating point or ANS floating point, you just have to check whether the constant "ZENFP" has been defined, e.g.:

```
[DEFINED] ZenFP   [IF] .( Zen float is loaded) cr [THEN]
[UNDEFINED] ZenFP [IF] .( ANS float is loaded) cr [THEN]
```

If you enter the realm of full fledged floating point numbers, you may find it very hard to resolve all dependencies. Well, it is not so hard as you think as long as you apply the following rule of the tumb. Before including any other floating point library file include this one:

```
include lib/fp3.4th
\ now you may include other library files
```

If you don't mind the size or the speed but *do* require full ANS Forth compliance, begin your program like this:

```
include lib/fp4.4th
\ now you may include other library files
```

You will see that this resolves most of your dependency problems[28]. Bottom line: if you don't need floating point numbers, avoid them and apply other techniques. But sometimes it cannot be avoided and I guess you'll agree with me that it's good it's there.

[28] If not, take a look at chapter 23.2.

12.27 Forth Scientific Library

4tH includes serveral library members of the Forth Scientific Library. The FSL contains several very complex and highly specialized mathematical words. Some of these words have been adapted to work with 4tH. All you have to do is to include `fsl-util.4th` after you've included `ansfloat.4th`, `ansfpio.4th` and preferably also `fpconst.4th`. It doesn't work with `zenfloat.4th`.

```
include lib/ansfloat.4th
include lib/ansfpio.4th
include lib/fsl-util.4th
```

Note that like `ansfloat.4th`, `fsl-util.4th` isn't included automatically by other library members, so you have to include it *explicitly*. If not, the compiler will simply abort. Since the FSL uses special datatypes, you have to do some work in order to get them to work with 4tH. If you want to declare an fsl-array of e.g. ten floats:

```
10 FLOAT MARRAY MyFSL
```

You have to declare it like this:

```
10 FLOATS 1+ ARRAY MyFSL            ( allocation)
FLOAT LATEST FSL-ARRAY              ( initialization)
DOES> (FSL-ARRAY) ;                 ( runtime behavior)
```

If you want to declare an fsl-matrix of e.g. 16 by 8 floats:

```
16 8 FLOAT MMATRIX MyFSL
```

You have to declare it like this:

```
16 8 * FLOATS 2 + ARRAY MyFSL       ( allocation)
16 8 FLOAT LATEST FSL-MATRIX        ( initialization)
DOES> (FSL-MATRIX) ;                ( runtime behavior)
```

Surely, this is a bit awkward, but it is required to be as compatible as possible with sources that use the FSL. Note that all restrictions and reservations concerning floating point support also apply to the FSL words.

12.28 Statistical functions

4tH features a small array of statistical functions that can be used in conjunction with both the ANS and the Zen floating point libraries. First you have to declare the floating point stack of your choice:

```
false constant shared-fp            \ select ANS- or ZEN float
                                    \ include the ZEN float library
shared-fp
[IF]   include lib/fp2.4th
[ELSE] include lib/fp4.4th
[THEN]                              \ include the ANS float library
```

After you've included the statistical library itself you have to allocate the statistical structure required:

```
include lib/statist.4th              \ include the statistical library

/st_var array stats                  \ allocate a statistics structure
```

That was the hardest part. Now, initialize the floating point library and clear the structure so it is ready for use:

```
fclear 100 set-precision             \ initialize FP library
stats st.clear                       \ clear the structure
```

You can now add your floating point numbers to the structure by issuing:

```
2 s>f stats st.add
4 s>f stats st.add
4 s>f stats st.add
4 s>f stats st.add
5 s>f stats st.add
5 s>f stats st.add
7 s>f stats st.add
9 s>f stats st.add
```

If you have forgotten how many numbers you added, simply query the structure as if it were a normal variable:

```
stats @
```

A number of statistical functions are now at your disposal, e.g. variance, standard deviation, and several means. You can call them at any time by issuing, e.g.:

```
stats st.stddev f. cr
```

Note you can have several statistical structures at the same time and manipulate or query them independently. You can also clear them individually at any time without any side effects.

12.29 Numerical integration

Although 4tH comes with lots of functions, you may encounter a situation where you need a function and it isn't there. If you're willing to make a full implementation, please do - and send the result to me for inclusion. However, if you need it *fast*, you may want to resort to numerical integration. A well-known method for numerical integration is "Simpson's rule[29]". We offer three versions of it:

1. Standard method;
2. Composite method;
3. The "3/8" rule.

[29] https://en.wikipedia.org/wiki/Simpson%27s_rule

None of these methods is "the best". You simply have to try which one works best for you in a given situation. So how do you go about it? First you have to find the derivative of the function you want to approximate. If you are not very good at math, you may have to find one. In this example we want to approximate $ln(x)$. The integral of the derivative $\frac{1}{x}$ is $\int_1^x \frac{1}{t} dt$. Now we have to turn the derivative into a function:

```
: f() 1 s>f fswap f/ ;
```

That wasn't too hard, was it? Finally, we have to put the entire Simpson construction into place:

```
include lib/fp3.4th
include lib/simpson.4th

: f() 1 s>f fswap f/ ;
: (ln) 1 s>f fswap ['] f() 1000 simpson ;
```

Of course that's all very nice, but where do all these parameters come from? Well, look at the integral. The parameters of the integral state that it goes from "1 to x", where x is the parameter of the $ln(x)$. That's why we do a "FSWAP" in the "(LN)" function. We also pass the execution token of the "F()" function. That has always one parameter, which is the parameter that "SIMPSON" feeds it. The dt simply indicates what the name of this parameter is: t.

Basically, an integral calculates the surface of a function. "SIMPSON" simply chops that curve up in little, tiny segments and t represents the number of chops[30]. The more chops, the more accurate the result will be - but also the more time it will take. The number of chops must be *even* for the composite rule and *divisible by three* for the "3/8" rule. So, let's put the thing to work now:

```
fclear 100 s>f (ln) f.
```

That results in *4.60517038495714201*, which is not half bad when you think of it, since 4tH's own "FLN" returns *4.60517018598809136*.

So let's do another one, the $erf(x)$ function. It's defined as $\frac{2}{\sqrt{\pi}} \int_o^x e^{-t^2} dt$. Since $\frac{2}{\sqrt{\pi}}$ is about *1.1283791671*, we already got that one and since it is not part of the integral we can leave that one out of the equation. So we're left with $\int_o^x e^{-t^2} dt$:

```
: g() fdup f* fnegate e fswap f** ;
```

Now let's assemble the whole thing:

```
include lib/fp3.4th
include lib/falog.4th
include lib/simpson.4th

: g() fdup f* fnegate e fswap f** ;

: (erf)
  0 s>f fswap ['] g() 100 simpson
  s" 1.1283791671" s>float f*
;
```

[30] It's a bit more complex than that, but it gives you an idea.

Why only a hunderd slices? Because it is enough. It returns *0.71115563366031653* for 0.75, where our native "FERF" function returns *0.71115563365344696* and the actual value is *0.71115563365351513*. That ain't half bad!

Sometimes you got a *really* difficult one, like the gamma function. That one is defined as $\int_0^\infty t^{x-1}e^{-t}dt$. There are several problems with this one. First it calculates the entire thing from here to eternity. The only way out of this is to find a sane way to define our own eternity. I won't go into the details, but 10,000 works. The second thing is, that the parameter is now part of our integral. The only way out of this is to save it in a variable and subsequently use it in the formula. Finally, we end up with something like this:

```
include lib/fp3.4th
include lib/falog.4th
include lib/simpson.4th

float array (x)

: h() fdup fnegate fexp fswap (x) f@ 1 s>f f- f** f* ;
: (gamma) (x) f! 0 s>f 10000 s>f ['] h() 10000 simpson ;
```

Ten thousand slices? Yup, that's what's needed here - this one takes a *long* time. No results? No. We're not out of the woods yet, but the fix has nothing to do with numerical integration, so we'll leave that one be. Note the other Simpson variants work exactly the same and you can combine the lot if you want it. But we have another way to do numerical integration - and it's a *lot* faster. If it works.

The only thing you have to do is initialize it - and then you can relax, because you won't be bothered with the question how many slices you want. The method is Gauss-Legendre. And this is how it works.

```
include lib/fp3.4th
include lib/falog.4th
include lib/gauslege.4th

: g() fdup f* fnegate e fswap f** ;

: (erf)
  0 s>f fswap ['] g() lege_inte
  s" 1.1283791671" s>float f*
;

lege_coef lege_roots
3 s>f 4 s>f f/ (erf)
```

You only have to initialize these roots and coefficients *once* before the first use of "LEGE_INTE". After that you're home free and you can do all the calculation you want. The result isn't half bad: *0.71115563365634331*. So why wouldn't use it and leave Simpson be? Because it doesn't always work. Our $ln(x)$ function is a lot worse: *4.52401046228249623*. And when functions got singularities[31], this method does not work at all.

[31] In mathematics, a singularity is in general a point at which a given mathematical object is not defined, or a point of an exceptional set where it fails to be well-behaved in some particular way. E.g. $f(x)=\frac{1}{x}$ has a singularity at zero.

Algorithm	Library	Returns	Standard API
KISS	gmkiss.4th	single	Yes
super KISS	gmskiss.4th	single	No
Mersenne twister	mersenne.4th	single|float	No
MINSTD	minstd.4th	single	Yes
Mifare Crypto-1	mif-prng.4th	single	No
ACM V31N06	plecuyer.4th	float	No
GGUBS	prng.4th	float	No
RAN4	ran4.4th	double	No
Frank Buss	randbin.4th	single	No
Lehmer	randlcg.4th	single	Yes
SimScript	randlcg.4th	single	Yes
EPM2	randlcg.4th	single	Yes
RANDU	randlcg2.4th	single	Yes
CUPL	randlcg2.4th	single	Yes
SuperDuper	randlcg2.4th	single	Yes
Derive	randlcg2.4th	single	Yes
BCPL	randlcg2.4th	single	Yes
Turbo C++	randlcg2.4th	single	Yes
Turbo Pascal	randlcg2.4th	single	Yes
ANSI-C	randlcg2.4th	single	Yes
Simula BS2000	randlcg2.4th	single	Yes
Atari ST	randlcg2.4th	single	Yes
C-RAND	randlcg2.4th	single	Yes
APL	randlcg2.4th	single	Yes
BSPR	randlcg2.4h	single	Yes
EPM	randlcg2.4th	single	Yes
Starting Forth	randlcg2.4th	single	Yes
MWC v/d Horst	randmwcp.4th	single	Yes
BSD	random.4th	single	Yes
Microsoft	random.4th	single	Yes
Doom	randoom.4th	single	Yes
ACM V12N04	runiform.4th	float	No
TT800	tt800.4th	float	No
WELL512	well512.4th	single	No
Wichman Hill	wichill.4th	float	No
ZX Spectrum	zxrandom.4th	float	No

Table 12.3: 4tH randomizers

ALGORITHM	LIBRARY	COMBINE?	SIZE	SPEED
Slow sort	slowsort.4th	No	2.3	∞
Stooge sort	stoosort.4th	No	2.7	>500,000
Bubble sort	bublsort.4th	No	1.4	2500
Cocktail sort	cocksort.4th	No	2.6	2800
Cocktail sort (improved)	coc2sort.4th	No	2.0	1800
Simple sort	simpsort.4th	No	1.0	1800
Simple sort (improved)	ismpsort.4th	No	1.1	1300
Oyelami sort	oyelsort.4th	No	2.8	600
Insertion sort	instsort.4th	No	1.5	1500
Insertion sort (improved)	ins2sort.4th	No	1.7	1000
Selection sort	selcsort.4th	No	1.2	1200
Cycle sort	cyclsort.4th	No	2.7	4100
Pancake sort	pancsort.4th	No	2.4	3600
Heap sort	hea2sort.4th	No	3.9	2.5
Introsort	intrsort.4th	No	11.5	1.0
Quicksort	qsort.4th	No	4.0	1.0
Quicksort (unsafe)	qsort.4th	No	3.0	1.0
Comb sort	com2sort.4th	No	3.0	1.3
Shell sort	shelsort.4th	No	2.6	2.5
Shellsort (improved)	shelsort.4th	No	4.1	2.0
Circlesort	circsort.4th	No	2.0	20
Circlesort (improved)	cir2sort.4th	No	5.6	10
Bitonic sort	bitosort.4th	No	3.6	10
Merge sort	mergsort.4th	No	2.6	20

Table 12.4: 4tH sorting algorithms (address based)

ALGORITHM	LIBRARY	COMBINE?	SIZE	SPEED
Heap sort	heapsort.4th	Yes	3.0	3.0
Comb sort	combsort.4th	Yes	3.2	2.5
Selection sort	sel2sort.4th	Yes	1.3	3900
Bubble sort	bub2sort.4th	Yes	1.4	6000
Gnome sort	gnomsort.4th	Yes	1.3	6000
Gnome sort (improved)	gno2sort.4th	Yes	1.8	4200

Table 12.5: 4tH sorting algorithms (index based)

2OS	TOS	Word	Result
fraction	fraction	V*	fraction
fraction	cell	V*	cell
cell	fraction	V*	cell
fraction	fraction	V/	fraction
cell	cell	V/	fraction
cell	fraction	V/	cell
fraction	fraction	+	fraction
fraction	fraction	-	fraction

Table 12.6: Fraction words

FAMILY	FILE	DE-FAULT	SPEED	ACCU-RACY	SIZE	ALGORITHM
FSIN	`fsinfcos.4th`	Y	+	+	+	Remez
	`fsincost.4th`	N	+	o	+	Taylor
FEXP	`fexp.4th`	Y	+	+	o	Remez
	`fexpt.4th`	N	o	o	+	Taylor
FLN	`flnflog.4th`	Y	+	+	o	Remez
	`flnflogb.4th`	N	-	+	+	Brute force
FERF	`ferf.4th`	*N/A*	+	o	+	Taylor, approx.
	`erf1.4th`	*N/A*	o	-	o	Approximation
	`erf.4th`	*N/A*	-	+	-	Chebyshev
GAMMA	`gamma.4th`	*N/A*	+	o	o	Fourier series
	`gammaln.4th`	N/A	o	+	+	Lanczos
	`fgamma.4th`	N/A	o	++	+	Spouge

Table 12.11: ANS-Forth functions

FILE	Family	FP stack	ANS-FORTH COMPLIANCE	SIZE	PRECISION	PORTING TO ANS-FORTH	PORTING ANS-FORTH
fp0.4th	Zen	Unified	None	Small	Low	No	No
fp1.4th	Zen	Unified	ANS 1994 compatible	Small	Low	Yes	Limited
fp2.4th	Zen	Unified	ANS 1994 compilant	Medium	Low	Yes	Yes
fp3.4th	ANS	Separate	ANS 200x compatible	Medium	High	Yes	Limited
fp4.4th	ANS	Separate	ANS 200x compliant	Large	High	Yes	Yes

Table 12.12: Floating point configurations

FUNCTION	WORD
Standard deviation	st.stddev
Variance	st.variance
Arithmetic mean	st.amean
Geometric mean	st.gmean
Harmonic mean	st.hmean
Root mean square	st.rms

Table 12.13: Statistical functions

Chapter 13

Special libraries

13.1 Infix formula translation[1]

Part of the elegance and simplicity of Forth comes from the use of Reverse Polish or postfix notation. As such, there are no requirements for precedence rules, parentheses, or complicated parsers. However, complicated mathematical and scientific equations that are often written in infix notation can be tedious to convert to Forth code. To simplify such a conversion, Julian V. Noble (one of the founders of the Forth Scientific Library, see section 12.27) wrote a FORmula TRANslator[2] to convert infix code to Forth code. Later Wil Baden, using Noble's concepts implemented a different version. For an implementation in 4tH, Wil Baden's version was chosen as it seemed simpler. The 4tH version of this (`opgftran.4th`) consists of three main words:

`FTRAN`	(a1 n1 – a2 n2) converts a text string from infix to postfix notation
`4TH-MATH`	(–) sets FTRAN to produce 4tH code as the output text
`STANDARD-MATH`	(–) sets FTRAN to produce standard postfix notation for the output text

To see how these work, consider the following code segment:

```
include lib/opgftran.4th

s" (12 - 2)/(6 - 3)" 2dup
forth-math    ftran type cr cr
standard-math ftran type cr
```

Which results in the following output:

```
s" 12E0" s>float
s" 2E0" s>float F-
s" 6E0" s>float
s" 3E0" s>float F- F/
12 2 - 6 3 - /
```

[1] Article written and contributed by David Johnson.
[2] Also known as "Operator Precedence Grammar".

As you can see, the Forth version of the output produces the 4tH code needed to do the calculation, while the standard version just produces the RPN output. Much more complicated expressions can be evaluated in which functions and variables are also handled. Variables begin with a letter and are not followed by a left parenthesis mark; whereas, function-calls are followed by parentheses.

Constants can also be handled by using embedded 4tH code. Line brackets (e.g., lembedded 4tH codel) are used for this, and all the text between the 'l' brackets is taken as 4tH code. For example, consider that PI has been defined as a floating point constant (or similar.) Then `|PI|` can be used in the expression to be evaluated as a constant.

```
include lib/opgftran.4th

81 string Eq$
s" Stress=G/(2*|PI|*(1-v)*r)*b*sin(theta)" Eq$ place

Eq$ count forth-math ftran type cr cr
Eq$ count standard-math ftran type cr
```

In the equation for Stress, the variables are `G`, `v`, `r`, `b`, `theta` and `Stress`, while `PI` is a constant and `sin` is a function. These are expressed in the resulting output as:

```
G F@ s" 2E0" s>float PI F*
s" 1E0" s>float v F@ F- F* r F@ F* F/
b F@ F* theta F@ Fsin F* Stress F!
G @  2  PI *  1  v @ - * r @ * /
b @ * theta @  sin * Stress !
```

The 4tH version of the output uses "F!" and "F@" for the variables and assumes that all functions should start with the letter 'F'. If this is not the case, the function is renamed (e.g. `sin` vs. `fsin`). However, the "standard-math" version does not make such an assumption. Note, `PI` is treated the same in both cases as embedded 4tH code.

13.2 Evaluating infix formulas at runtime[3]

In the case in which the end-user my not wish to use RPN notation, the use of `opgftran.4th` allows for the conversion to postfix notation. A simple example is a calculator where the user can use either postfix or infix notation. The 4tH library already supplies an interpretive format; as a minimal example (from `dc.4th`), consider:

```
include lib/zenfloat.4th
include lib/zenans.4th
include lib/fpin.4th
include lib/fpout.4th
include lib/evaluate.4th
include lib/opgftran.4th

: _+ f+ ;
: _- f- ;
: _* f* ;
: _/ f/ ;

: _.  f.  space ;
: let  [char] :  parse ftran evaluate ;
: _quit quit ;
```

[3] Article written and contributed by David Johnson.

```
create wordlist
  ," +"          ' _+ ,
  ," -"          ' _- ,
  ," *"          ' _* ,
  ," /"          ' _/ ,
  ," ."          ' _. ,
  ," quit"       ' _quit ,
  ," let"        ' let ,
  NULL ,

wordlist to dictionary                \ assign wordlist to dictionary
                                      \ The interpreter itself
: mycalc
  standard-math
  begin                               \ show the prompt and get a command
    ." OK" cr refill drop             \ interpret and issue oops when needed
    ['] interpret catch if ." Oops " then
  again                               \ repeat command loop eternally
;

mycalc
```

Thus, the calculator could be used with infix notation such as:

```
let (2-1)/3: .
```

or with postfix notation such as:

```
2 1 - 3 / .
```

A more complete implementation is given by the `fdc.4th` example.

13.3 Converting infix formulas[4]

Using the 4tH library and `convert.4th`, infix equations can be added to your source code and converted before compiling. An example program to do this is `opgconv.4th`. In Wil Baden's original program, the single end-user word was "LET" based on same word in BASIC. This idea is implemented in `opgconv.4th` where a formula to is found by parsing the text between a "LET" and ":" pair. For example, consider the following program (`mytest.opg`):

```
include lib/zenfloat.4th
include lib/zenans.4th
include lib/fpin.4th
include lib/fpout.4th
include lib/zenfsqrt.4th

FLOAT array a   FLOAT array b   FLOAT array c
FLOAT array disc                          \ Used for discriminant
                                          \ Example from
: FQUADRATICROOT  ( F: a b c -- r1 r2 )   \ Wil Baden's OPG.TXT
   c F!  b F!  a F!                       \ Pickup coefficients
   LET disc = SQRT(b*b-4*a*c):            \ Set discriminant
   LET (-b+disc)/(2*a), (-b-disc)/(2*a):  \ Put values on f-stack
;

( Solve x*x-3*x+2 )  LET FQUADRATICROOT (1,-3, 2) : F. F. cr
```

[4] Article written and contributed by David Johnson.

This program can be converted to 4tH by using `opgconv.4th`. For example (assuming that it is named `mytest.opg`) one could use:

```
4th cx opgconv.4th mytest.opg mytest.4th
```

Some more complicated examples are also given in the `testopg.opg` file.

13.4 Portable Bitmap graphics[5]

You can make simple graphical drawings in 4tH using a virtual screen or bitmap. For example, consider the following:

```
include lib/graphics.4th
10 20 100 200 line
s" test.ppm" save_image
```

This short program will create a portable Netpbm file which can be converted to other graphical formats using the Netpbm libraries or viewed in numerous imaging programs. The image `test.ppm` is 640*480 pixels in size and simply consists of a white line plotted on a black background. However, one is not limited to such simple images. The PPM-format allows for true color resolution with each pixel consisting of 3 bytes for red, green, and blue. Each byte can have a maximum value or intensity of 255.

The ppm-format is a simple way to describe a "raw" image file in a widely recognized format. The format of the file header consists of a magic word, the size of the image, and the maximum intensity of the pixels. Following this is then the raw data. This format was created by Jeff Poskanzer as part of his Pbmplus graphical utilities (now called Netpbm). The PPM/PGM formats are used in the 4tH library routines, and since the PGM format is the simpler of the two, it will be presented first.

Here is a simple 8*8 PGM image of a diagonal line (white) on black background. The header consists of the magic number "P2" which indicates that the file is PGM image with the data stored as text. The next line is a comment (marked with a '#'), followed by the image size (8x8) and the maximum value of the data (or pixel). If the data after the file header is stored as binary, the the magic number is given by P5.

```
P2
# Simple test of 8x8 PGM grayscale
8 8
255
255 0 0 0 0 0 0 0
0 255 0 0 0 0 0 0
0 0 255 0 0 0 0 0
0 0 0 255 0 0 0 0
0 0 0 0 255 0 0 0
0 0 0 0 0 255 0 0
0 0 0 0 0 0 255 0
0 0 0 0 0 0 0 255
```

Color PPM images are stored with a similar format except 3 bytes are required to specify each pixel (e.g. red, green, blue). Thus PPM images are 3 times larger than the PGM grayscale images. The magic numbers for PPM files are "P3" for text and "P6" for binary. Both text and binary PPM/PGM files can be loaded using the 4tH library. However, images are always saved using the binary (P3, P6) format.

As an example for the P3, P6 color format, try loading and saving `icon_p3.ppm`.

[5] Article written and contributed by David Johnson.

```
include lib/graphics.4th
s" icon_p3.ppm" get_image
cr image_comment$ count type
s" 4th_icon.ppm" save_image
```

Now compare the contents and size of the two files. Better yet, see if you can view the file[6].

When a PPM/PGM image is loaded in 4tH (via "GET_IMAGE"), the data from the image file header is is stored in the 4tH variables: "PIC_HEIGHT", "PIC_WIDTH", and "PIC_INTENSITY". Additionally, both color PPM files (P3, P5) and grayscale PGM (P2, P5) files are supported. The main difference is that there is only one byte per pixel for grayscale image. As another example, consider the following:

```
\ Define, draw and save an image
include lib/graphics.4th

: bar ( -- ) 10 0 do 50 i + 50 50 i + 200 line loop ;

300 pic_width !              \ Define the current image size
300 pic_height !

color_image                  \ Define and draw our image
blue background
yellow bar

s" testimage.ppm" save_image \ saved to file
```

Try using "GRAYSCALE_IMAGE" instead and view the results. Note, that grayscale Netpbm files often have a file extension of *.pgm instead of *.ppm, but most viewing/graphical programs do not care.

After defining the picture type and size, a couple of basic commands can be used to create and manipulate the image. These commands work with both color and grayscale images. However, there is one difference. For color images, each pixel requires three bytes which we'll abbreviate as *rgb;* whereas, grayscale images have only one byte per pixel. Therefore, the stack pictures corresponding to the pixel count will depend upon the type of image in use. For example, if you were to specify a "GRAYSCALE_IMAGE" instead of "COLOR_IMAGE" in the following program , then "COLOR@" would leave just one value on the parameter stack.

```
include lib/graphics.4th

300 pic_width !
300 pic_height !
color_image

0 0 255 color!          \ sets the "current" color to blue (e.g., rgb: 0 0 255)
cr ." current color is " color@ . . .

white                     \ now set the "current color" to white (255 255 255)
background                \ set every pixel in the image to white

blue                      \ current color is blue
100 100 set_pixel         \ will color the pixel at (100,100) to blue
cr ." Pixel at (100,100) is "
100 100 pixel@ . . .      \ check to see if it's blue
255 0 0 100 100 pixel!    \ change it to red (e.g. rgb: 255 0 0)

10 10 98 98 line           \ draw line from (10,50) to (90,90)
s" dotline.ppm" save_image \ save the image to view it
```

[6]Gimp, OpenOffice, xnview and many more should work fine

You may notice that the word ”BACKGROUND” is rather slow. If you just wanted to set the background to white or to some level of gray, you can use the word ”WHITEOUT” which is much faster. That is:

```
255 whiteout              \ set the image to white;
190 whiteout              \ set image to some gray level;
0 whiteout                \ set image to black.
```

To summarize, the basic graphical commands with stack pictures are[7]:

WORD	STACK EFFECT	COMMENT
pic_width	– x	address contains image width
pic_height	– x	address contains image height
pic_intensity	– x	address contains maximum byte value for pixels
save_image	a n –	save portable bitmap to file
get_image	a n –	load file containing portable bitmap
color_image	–	work with color (rgb) pixels
grayscale_image	–	work with grayscale (single byte pixels)
color!	rgb | n –	define current color
color@	– rgb | n	get current color
pixel@	rx cy – rgb | n	get pixel values at specified point
pixel!	rgb | n cr cy –	set pixel values at specified point
set_pixel	rx, cy –	set pixel at specified point using the "current color"
line	rx1 cy1 rx2 cy2 –	draw line using the "current color"
background	–	set image background to current color (slow)
whiteout	n –	set image background to white (n=255) or gray (fast)
red	–	set the "current" color in use for color or grayscale images to red
blue	–	set the "current" color in use for color or grayscale images to blue
green	–	set the "current" color in use for color or grayscale images to green
white	–	set the "current" color in use for color or grayscale images to white
black	–	set the "current" color in use for color or grayscale images to black
yellow	–	set the "current" color in use for color or grayscale images to yellow
magenta	–	set the "current" color in use for color or grayscale images to magenta
cyan	–	set the "current" color in use for color or grayscale images to cyan

There are some graphical words used in converting between image types (color vs. grayscale and so on) and these will be discussed later (also check the example 4tH files). *Lastly,*

[7] rx = row or x; cy = column of y; rgb = red green blue.

*should you wish to work with image that contain more than 800*600 pixels, then simply change the maximum defaults in the* `graphics.4th` *file.*

13.5 Turtle graphics[8]

Turtle graphics provide a simple set of words for drawing simple and complex shapes. There are three basic words: ”XHOME”, ”XROTATE”, and ”XMOVE”. The ’x’ here represents the turtle. A simple program to draw a box can be defined as:

```
include lib/graphics.4th
include lib/gturtle.4th

200 pic_width !
200 pic_height !
color_image

: box ( n -- ) 4 0 do dup xmove 90 xrotate loop drop ;

255 whiteout
xhome
50 blue box
30 green box
s" box.ppm" save_image
```

The motion of the turtle can also be directed by the words: ”FORWARD”, ”BACK”, ”RIGHT”, ”LEFT”, ”XPENUP”, and ”XPENDOWN”. Here is an interesting example where a simple shape can be used to make a more complex drawing.

```
include lib/graphics.4th
include lib/gturtle.4th

300 pic_width !
300 pic_height !
color_image

clear-screen
xhome
xpendown

: octagon ( -- )  8 0 do  25 forward  45 right  loop ;
: tile ( -- )   8 0 do  octagon  45 right  loop ;

tile s” tile.ppm”  save_image
```

A couple of other words that may prove useful. Should the turtle get lost, then ”TURTLE@” will give the coordinates while ”COMPASS@” will give the directional heading. Likewise ”TURTLE!” and ”COMPASS!” will reposition the turtle.

In summary, we have the following:

WORD	STACK EFFECT	COMMENT
turtle!	x y –	set turtle position
turtle@	– x y	get turtle position
compass!	n –	set turtle directional heading in degrees

[8] Article written and contributed by David Johnson.

WORD	STACK EFFECT	COMMENT
compass@	– n	get turtle directional heading in degrees
xhome	–	place turtle in center of image and set the heading to 0 degrees
xrotate	n –	Rotate turtle by n degrees, positive numbers are counter clockwise.
xmove	n –	Move n pixels, can move forward (+) or backwards (-)
forward	n –	Move n pixels forward
backward	n –	Move n pixels backwards
left	n –	Turn left (rotate) by n degrees
right	n –	Turn right (rtate) by n degrees
clear-screen	–	Clear to white background
xpenup	–	Change color to white so that movent of turtle does not leave a trail on white background
xpendown	–	Change color to blue; e.g., same as using the word blue.

13.6 Annotating Portable Bitmap images[9]

To annotate the turtle graphic image from the last example, try the following:

```
include lib/graphics.4th
include lib/gbanner.4th

s" box.ppm" get_image

red horizontal text_up 5 25 s" boxes!" gbanner
black vertical text_up 75 20 s" yea!" gbanner
green vertical text_down 190 190 s" Hello there!" gbanner
magenta horizontal text_down 190 150 s" Bye bye!" gbanner

s" box2.ppm" save_image
```

The general syntax is just the screen position and the string with the key word being ”GBANNER”. To make it interesting, the orientation and direction of the text can easily be changed. Note the starting positions for the text in this example as the text may be written from right to left for some orientations.

The basic words are summarized as follows:

WORD	STACK EFFECT	COMMENT
gbanner	rx cy a n –	print the string a n at position rx cy
horizontal	–	plot the text horizontally
vertical	–	plot the text vertically
textup	–	change the text direction to upright
textdown	–	change the text direction to upside down

[9] Article written and contributed by David Johnson.

13.7 Color Palettes[10]

Since the Netpbm grayscale images are 3 times smaller than the color ones, one way to improve the efficiency (if true color resolution is not needed) is to store the color information in the grayscale format. While the setup shown here is not that useful, it was easily implemented. Instead of converting a color pixel to a grayscale pixel, a 6 level rgb format is used. For this, colors have a maximum intensity of 5, and the conversion is : palette = 38*red + 6*green + blue giving a maximum value of 215. The word "PALETTE_IMAGE" will set the correct defaults, e.g.:

```
include lib/graphics.4th
include lib/gturtle.4th
include lib/palette.4th

300 pic_width !
300 pic_height !
palette_image
( use some graphical routines)
s" ptest.ppm" save_image
```

This will save the image as a PGM image (with a P5 magic word). Thus when you view the image, it will appear as a strange looking grayscale image (as another example, try the `gtest4.4th` example program) To view `ptest.ppm` in color, you must convert the image to a PPM color file. There are two example programs to do this. One is `color2pal.4th` and the other is `pal2col.4th`. Note for this particular color palette, once it is converted back to the color format (with a P6 magic word) it should have a maximum pixel intensity of 5. Unfortunately, many graphical viewers have trouble displaying this correctly (though many work fine). Therefore, the `col2pal.4th` example will scale up the pixels values to the 255 level.

Conversion between color and grayscale pixels can be performed using the words "COLOR>GRAY" and "GRAY>COLOR". If color palette and/or grayscale images are defined, the words "COLOR>N" and "N>COLOR" can be used instead.

WORD	STACK EFFECT	COMMENT
color>gray	rgb – n	Covert 3 byte color format to 1 byte grayscale
gray>color	n – rgb	Convert grayscale to color format. The pixel still has the gray color though
to_color	–	Set the library primitives so that pixel and color data will use 3 bytes.
to_gray	–	Set library primitives so that pixel and color data will use 1 byte
color_image	–	Set defaults for using color (PPM) images
grayscale_image	–	Set defaults for using grayscale (PGM) images
palette_image	–	Set defaults for using color palette. Similar to grayscale and image will be save in PGM format.
color>palette	rgb – n	Convert color information to palette code

[10] Article written and contributed by David Johnson.

WORD	STACK EFFECT	COMMENT
palette>color	n - rgb	Convert palette code to color
to_palette	–	Set library primitives so that pixel and color data will use 1 byte palette code.
color>n	rgb – n	Convert color to palette (or grayscale) format
n>color	n – rgb	Convert palette (or grayscale) to color format

13.8 3D plotting[11]

Plotting or drawing in 3D using the 2D graphics library in 4tH simply requires a few words to map the 3D coordinates to 2D. The orientation of the default coordinate system has the origin towards the top left of the screen with the x-axis running down the screen rows and the y-axis running left across the screen columns.

With a right-handed coordinate system the z-axis is then pointing out of the screen (towards the observer). Two words that show this are “CUBE” and “AXIS”. The mapping from 3D to 2D is done with the word “3D>2D”. The use of these words is shown below:

```
\ Example of cube using the "mirror-left" and "mirror-right" words
\ The blue line from different corner-corner orientations are
\ better viewed using different mappings of 3D to 2D
\ Loading this also loads the graphics.4th and gbanner.4th word-set
include lib/gplot3d.4th

\ Place the 3D axes in center of image
colorimage 255
whiteout
center-plot

\ Line from top-left to bottom-right is clearly visible.
mirror-left
0 0 0 100 black cube
0 0 0 3d>2d 100 100 100 3d>2d blue line
-50 0 0 red axis

\ Conversely, line from top-right to bottom left is clearly visible.
mirror-right
0 150 0 100 black cube
0 250 0 3d>2d 100 150 100 3d>2d blue line
-50 150 0 red axis

s" 2cubes.ppm" saveimage
```

The resulting image is shown in figure 13.1. Note the orientation of the two cubes. The right cube is drawn with the box coming out of the page torwards the right side of the image and this is the default orientation. One could have easily drawn the cube with an orientation tilted towards the left size of the image.

The orientation of 3D mapping can be controlled with two other words “MIRROR-LEFT” and “MIRROR-RIGHT” (which is the default). One reason for switching between these orientation is that the blue line drawn in the two cubes would not be easily visible if the orientation was switched.

[11] Article written and contributed by David Johnson.

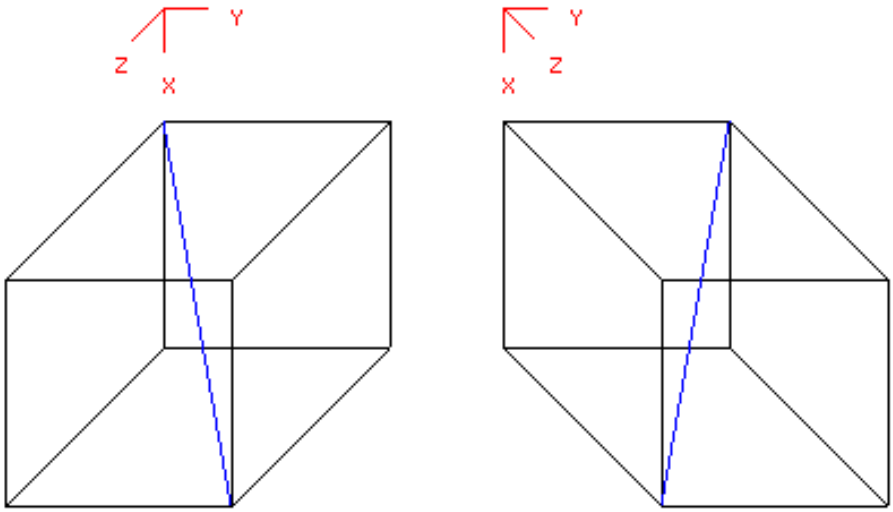

Figure 13.1: Mirrored boxes

For the 3D plotting to be really useful, one would also need to be able to rotate the coordinate axes to any desired orientation. This can be accomplished by defining the Euler angles ϕ_1, Φ, and ϕ_2. These represent a rotation about the *z-axis*, then about the new *x-axis*, and finally about the newest *z-axis*.

Positive values for the Euler angles represent a counter-clockwise rotation. For example, assume the *z-axis* should be placed parallel to the vector $znew = -1i+2j+3k$ or $[-1\ 2\ 3]$. This can be accomplished using ϕ_1 and Φ.

The first rotation about the *z-axis* is $\phi_1 = angle$ between $vector = -1i+2j+0k$ and the $x-axis+90°$. That is $\phi_1 = angle$ between $[-1\ 2\ 0]$ and $[1\ 0\ 0]+90°$ or $\phi_1 = acos\left(\frac{-1}{\sqrt{5}}\right)+\frac{\pi}{2}$ and hence $\phi_1 = 206.6°$.

Now a rotation about this *x-axis* will place the *z-axis* to the desired position, or $\Phi = angle$ between $[-1\ 2\ 3]$ and $[0\ 0\ 1]$. This gives $\Phi = acos\left(\frac{3}{\sqrt{14}}\right) = 36.7°$.

Lastly, a rotation about the newest *z-axis* ϕ_2 can place the the *x-* and *y-axes* to any desired location. Thus with the Euler angles of $\phi_1 = 206.6°$, $\Phi = 36.7°$ will place the *z-axis* parallel to$[-1\ 2\ 3]$ and the rotation ϕ_2 can then be used to set the *x-* and *y-axes*.

In 4tH, the word "EULER!" is used to set the Euler angles and these angles are defined in degrees. Here, integer math is used to calculate the coordinate transformation. Once the Euler angles are set, plotting is done as usual with the new orientation, which can be viewed with the word "AXIS". See the simple demo example below:

```
include lib/gplot3d.4th

colorimage                \ define bitmap picture
600 picwidth !
400 picheight !
center-plot
255 whiteout

magenta                   \ draw a big box
0 0 0 euler!              \ keep default axis orientation
0 0 -50 100 cube
```

```
0 0 -50 black axis

green                       \ draw a smaller rotated box
90 45 -20 euler!            \ rotate the coordinate system
0 0 0 50 cube
0 0 0 blue axis

s" box.ppm" saveimage
```

WORD	STACK EFFECT	COMMENT
origin!	x y –	set the location of 3D-axis within the 2D image
center-plot	–	place the 3D-axis axis origin at the image center
euler!	phi1 Psi phi2 –	Set Euler angles to define plot orientation
xyznew	xold yold zold – x y z	apply coordinate transformation to map point to the new system
3d>2d	rx cy z1 – rx cy	map x,y,z to 2D for the graphic.4th wordset
axis	x y z –	draw the current axis orientation at point xyz.
cube	x y z size –	draw a cube at xyz
mirror-left	–	when mapping from 3d to 2d use the “left” version of the cube view
mirror-right	–	Use the “right” version of the cube view (default condition)

Table 13.4: 3D plotting wordset

The corresponding image is shown in figure 13.2. To summarize, the 3D plotting word set is to be used in conjunction with those words already defined in the graphics package. The complete wordset is listed in table 13.4.

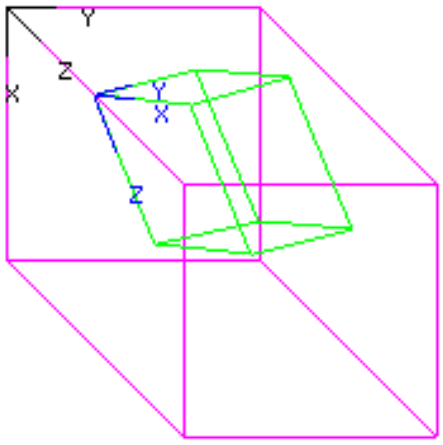

Figure 13.2: Two boxes

13.9 3D turtle graphics[12]

For the turtle graphics library[13] `gturtle.4th`, the word that moves the turtle by the number of pixels issued is "XMOVE". Essentially, a short line of length *n* is drawn. If the turtle is moving in 3D then this can be specified in "XMOVE" using the defered word "XDSCALE". By default, "XDSCALE" executes the word "2DPLOT", which takes *no* parameters and does absolutely nothing. However, in 3D one *could* define:

```
variable turtlez
0 turtlez !

: zturtle! ( x y z -- ) turtlez ! turtle! ;

\ place the turtle in 3D
: 3dplot ( x1 y1 -- x2 y2 ) turtlez @ 3d>2d ;
' 3dplot is xdscale
```

As a result, when "XMOVE" is executed the turtle moves on the specified z-plane to create an image. Some included examples and demo files using 3D plotting are listed in table 13.5.

File	Description
`gmirex.4th`	Example shows the use of `cube`, `axis`, `3d>2d`, and `mirror-left` and `mirror-right`.
`gdemo3d.4th`	Demo shows use of `euler!` to rotate the coordinate axes
`plt3dex.4th`	Example to plot of a simple 3D surface with two different views.
`3ddemo.4th`	This demo plots $\frac{\sin\sqrt{x^2+y^2}}{\sqrt{x^2+y^2}}$ with many views.
`g3spiral.4th`	Example of using 3D plotting with turtle graphics.

Table 13.5: 3D plotting examples

13.10 Interpreters

Those of you who know Forth will be very surprised to see that 4tH doesn't have a Forth prompt. Some will be even more surprised to see that 4tH does have an interpreter. It is a library routine, written in 4tH, that can easily be adapted and expanded. If you can write 4tH and maintain a table, you can use it. The next question you have to ask yourself, is do you want your interpreter to be case sensitive or not? If it is, "id" will work, but "Id" or "ID" will not. If you want it to be case sensitive, add the pragma `casesensitive`. Example:

```
[pragma] casesensitive        \ don't ignore case
[needs lib/interpret.4th]

: _+ + ;
: _. . ;
: id ." This is 4tH" cr ;
```

[12] Article written and contributed by David Johnson.
[13] See section 13.5.

Well, that isn't very hard, is it. Now we add a table to all that:

```
create wordlist
  ," +"  ' _+ ,
  ," ."  ' _. ,
  ," id" ' id ,
  NULL ,
```

Remember to terminate your table with "NULL"! Every entry consists of a string and an address to your routine. What will happen is that your user enters the string and the appropriate routine will be called. In this case, your interpreter has three commands: "+", "." and "id". We're a hair away from a real interpreter. We just have to assign our table to the dictionary. These lines do the job:

```
wordlist to dictionary
refill drop interpret
```

Now you can compile your application and run it. Enter:

```
45 12 + .
```

And it will print:

```
57
```

Yes, it's just as easy as that! If you enter something the interpreter doesn't recognize it will try to convert it to a number and throw it on the stack. But you will also see that it exits after you've entered that single line. That is because the interpreter is called just once. If you change that to:

```
begin refill drop interpret again
```

It will return with an new prompt. In that case it is wise to add a routine like:

```
: _quit quit ;
```

And add it to your interpreter, because otherwise your user will not be able to leave the application. Note that you have to do all the error-checking. E.g., if your user calls "_+" without putting sufficient items on the stack, 4tH will exit with an error. Of course, you can catch any exceptions. "INTERPRET" has a builtin word, "NotFound", that deals with any unrecognized strings. You can define your own if you want to. The only thing you have to do is to write a word which takes an address/count string and returns nothing, e.g.:

```
:noname 2drop ." I don't understand this!" cr ; is NotFound
```

Or more elaborate:

```
:noname ." I don't what '" type ." ' means!" cr ; is NotFound
```

You could even integrate it with the exception trapping, if you defined one:

```
1 constant #UndefName

:noname #UndefName throw ; is NotFound
```

When a word is not found, a user exception is thrown. This example is taken from `dc.4th`:

```
: dc
  begin                         \ main interpretation loop
    ." OK" cr                   \ print prompt
    refill drop                 \ get input from use
    ['] interpret               \ interpret it
    catch dup                   \ catch any errors
    if                          \ if one occurred
      ShowMessage               \ show a message
    else                        \ otherwise
      drop                      \ drop the throw code
    then
  again                         \ loop back
;
```

You can still see the basic structure, but this one is much more advanced. You can also remove the code from the interpreter that decodes numbers. In that case, if a word is not found in the "dictionary" table it will exit immediately and report an error. Simply define this pragma *before* including the library:

```
[pragma] ignorenumbers
include lib/interprt.4th
```

Note that only the use of integers is supported in these interpreters. If you want floating point support you have to include ZEN floating point support[14] *before* including `interprt.4th`, either:

```
include lib/zenfloat.4th
include lib/zenfpio.4th
include lib/interprt.4th
```

Or:

```
include lib/zenfloat.4th
include lib/zenans.4th
include lib/fpin.4th
include lib/fpout.4th
include lib/interprt.4th
```

The difference between both form is that the former only supports floating point entry in the scientific form. Consequently, the latter is more suited for humans like you and me.

Don't let anybody ever tell you you can't make interactive applications with 4tH. As you have seen, you can with very little effort.

13.11 Menus

The menu is a well-known user interface on the console. Typically, it displays a list of options from which the user can choose. 4tH allows you to create such menus with *very* little effort. First, include the appropriate library file:

```
include lib/menu.4th
```

[14] See section 12.23

Each option needs his own word. Note that an option - any option - can't take nor leave any items on the stack:

```
: helloworld ." Hello world!" cr ;
: calculate ." 1 + 1 = " 1 1 + . cr ;
: bye ." Goodbye!" quit cr ;
```

Now we have to define our menu. The first item is the title of the menu:

```
create MyMenu
  ," My beautiful program"
```

All subsequent entries are options. Each option consists of a description and an execution token. The menu is terminated by "NULL":

```
create MyMenu
  ," My first little programs"
  ," Hello world" ' helloworld ,
  ," 1 + 1"       ' calculate ,
  ," Exit"        ' bye ,
  NULL ,
```

Now all we have to do is to bind it to the main menu and run it:

```
MyMenu MainMenu
RunMenu
```

That's it! But what if we want to nest menus? Well, a nested menu is simply an option which opens up a new menu. For each additional menu we need to do two things. First, define a variable:

```
variable NextMenu
```

The variable will hold the address of the menu table you will declare later on. Second, define a word that allows us to enter the menu:

```
: >NextMenu NextMenu >Menu ;
```

That particular word is needed when we add an extra option to the main menu, which gives the user access to the sub menu we're about to define:

```
create MyMenu
  ," My first little programs"
  ," Hello world"   ' helloworld ,
  ," 1 + 1"         ' calculate ,
  ," More programs" ' >NextMenu ,
  ," Exit"          ' bye ,
  NULL ,
```

Next, we have to define the sub menu itself:

```
create MyNextMenu
  ," More little programs"
  ," Bottles of beer" ' bottlebeer ,
  ," FooBar"          ' foobar ,
  ," Return to main"  ' >MainMenu ,
  NULL ,
```

Hey, where did ”>MainMenu” come from? Well, it’s part of the library, so it’s always at your disposal. And it’s included in this menu, because otherwise the user would be stuck in this menu and wouldn’t be able to return to the main menu. Finally, you have to add the menu by feeding the table and the variable you declared to ”AddMenu”:

```
MyNextMenu NextMenu AddMenu
```

Done! Next time you execute ”RunMenu”, you’ll see have an extra option giving you access to the sub menu you just created. BTW, You can have as many menus as you like and can nest them as deep as you want.

Another nice feature is that when an error occurs during execution, the menu will display the appropriate message and simply return. In other words, it won’t bomb out, no matter how bad a programmer you are.

13.12 Finite state machines[15]

Some tasks are easy to specify, but hard to program procedurally. Consider the task of accepting numbers from the keyboard. An unfriendly program lets the user re-enter the entire number because he made a small error. A friendly program, by contrast, filters the input so small errors are automatically corrected. For example, a number input routine that only allows signed decimal numbers. Decimal points, numerals and leading signs are all legal, but no other ASCII characters (including spaces) will be recognized.

If you try writing a single routine to ’EMIT’ or ’DROP’ a key according to these rules and you will find that there are just too many conditions for comfort. A test for each of the three classes of acceptable input, and tests for whether a sign or a decimal point have already been received.

The first state tests for all three possibilities, the second for two, and the last for only one (0-9). Is there a simple way to program that clearly? Yes, instead of nesting words within each other we can pass control as states on the data stack. In this context a state is simply an execution token, which when executed leaves another state on the stack for each possible exit condition. The program is simply a loop which continuously executes states.

Several states are identical in the tests made and actions taken, differing only in the state to be executed next. We need to be able to define the action to be taken and the transition to be made separately. The number returned can be used as an index into a two-dimensional array holding the action to be taken and the index of the state to be executed next. Well, this is what we call a Finite State Machine (FSM). It is very easy to define an FSM in 4tH.

First, you have to declare an array large enough to hold the FSM and its header information:

```
include lib/fsm.4th
3 4 * 2 * fsm.head + array fixed.pt
```

We have three states and four possible inputs. Each entry requires two cells and finally we have to take the overhead into account. But before we can define our FSM, we have to create the execution tokens. These tiny definitions do all the hard work:

```
:token (drop) drop ;
:token (emit) emit ;
```

[15] Part of this section is based on an article in Forthwrite UK by Jenny Brien.

And these take care of the transitions to another state:

```
:token >0 0 ;
:token >1 1 ;
:token >2 2 ;
```

Now we can setup the FSM itself by declaring the number of classes (its width) and the array that holds it. Note the use of the word "WIDE" is *mandatory*:

```
4 wide fixed.pt fsm
\ input   other    ||  digit    ||  - sign   ||  decimal point
\ state  ------------------------------------------------------
  ( 0 )  (DROP) >0 || (EMIT) >1 || (EMIT) >1 || (EMIT) >2   ||
  ( 1 )  (DROP) >1 || (EMIT) >1 || (DROP) >1 || (EMIT) >2   ||
  ( 2 )  (DROP) >2 || (EMIT) >2 || (DROP) >2 || (DROP) >2   ||
```

That's all! Before you can use an FSM, you first have to activate it and set the initial state. Note that upon definition the FSM already is activated with state zero, but is a good habit to explicitly initialize it:

```
fixed.pt to fsm.current
0 fsm.state !
```

We only have to transform the character to the appropriate class in order to make this baby fly:

```
0 enum 'other' enum 'digit' enum 'sign' constant 'point'

create categorize
  char 0 , 'digit' ,
  char 1 , 'digit' ,
  char 2 , 'digit' ,
  char 3 , 'digit' ,
  char 4 , 'digit' ,
  char 5 , 'digit' ,
  char 6 , 'digit' ,
  char 7 , 'digit' ,
  char 8 , 'digit' ,
  char 9 , 'digit' ,
  char + , 'sign'  ,
  char - , 'sign'  ,
  char . , 'point' ,
  char , , 'point' ,
  NULL ,
does>
  2 num-key row if cell+ @c else drop 'other' then nip
;
```

Time to execute, which is hardly rocket science either:

```
s" bad-345.rubbish4467invalid"
begin
  dup
while
  over c@ dup categorize
  fsm.run
  chop
repeat 2drop cr
```

Mission accomplished. Note you can define as many FSMs as you need. If you want to switch between them just activate them before use. Since the state is part of the FSM, it will remain unchanged between calls.

13.13 Virtual memory

Although computers have lots of memory nowadays, you may find yourself in a situation where you don't have enough memory to cater for your needs. In that case you may resort to using 4tH's virtual memory library.

4tH's virtual memory library uses block files, so you may need to create one before you can use it. Since block files are character-oriented you need to include another library if you need numerical data:

```
include lib/vmem.4th
include lib/ncoding.4th
```

That's all so let's define some variables and open the block file we created previously:

```
variable n1 nell vallot latest ! does> paging ;
variable s1   32 vallot latest ! does> paging ;
variable s2  128 vallot latest ! does> paging ;
variable s3  512 vallot latest ! does> paging ;
variable s4  512 vallot latest ! does> paging ;
s" vmemtest.scr" open-blockfile
```

When setting this up, you have to take three things in consideration:

1. Although it isn't required, you'd better define virtual memory variables in order, from the smallest to the largest;
2. The size of a virtual memory variable may *not* exceed "B/BUF" address units;
3. You have to make sure that all data fits in the block file. Take into consideration that no variable can span two blocks, so there'll be some overhead. The easiest solution is not to take any risks and simply oversize it - disk space is relatively cheap.

Now you can access them like any other variable:

```
2960 n1 n! &
s" This is a virtual memory string" s1 place &
```

It looks (almost) completely normal, doesn't it? Except for that ampersand, of course. This ampersand has to be added *every* time you write to a virtual memory variable. It signals to the underlying block management system that the block is now "dirty". If you think that's too cumbersome for your taste, you may adopt another solution - it's not *that* complicated.

Reading a variable is no problem at all: do as you've always done:

```
n1 n@ . s1 count type cr
```

And what about arrays? Well, first of all they have to be character based. And they're restricted to "B/BUF" address units. Or are they? As a matter of fact, they don't - but they do need some special treatment.

First of all, you have to think of the element size, which *must* be a power of two and may *not* exceed "B/BUF" address units. Second, you need to know how many rows you require. Now we can define it:

```
64 constant /row
80 constant #row
variable a1    20 nells vallot latest !
  does> swap nells +paging ;
variable a2 /row #row * vallot latest !
  does> swap /row * +paging ;
```

So "a1" is an integer arrays and "a2" is a string array. So far, so good. But what have we defined at the 'DOES>' part? Well, to address an array we're going to write:

```
5 a1 10 a2
```

Which will return the addresses of the sixth element of "a1" and the eleventh element of "a2" - we're counting from zero. Consequently, we pass the element number and the address of the array to the 'DOES>' part of the definition.

That's were the 'SWAP' comes in, since that one puts the element number on top, which we multiply with the element size to get the offset of that specific element in the array. Then we pass the whole bunch to "+PAGING". That's where the magic begins! If you e.g. prefer a more FSL-like array, now you know what to do..

The same rules apply here, if you want to write to a variable don't forget the ampersand:

```
2960 5 a1 n! &
s" This is a virtual memory string" 10 a2 place &
```

And you can just as easily retrieve it:

```
5 a1 n@ . 10 a2 count type cr
```

That's all! When you're done, you'll have to close your blockfile, of course. But that ain't too tricky:

```
close-blockfile
```

Will do just fine. Tip of the day: there is also a preprocessor library[16] which supports this virtual memory. Not only is it far more efficient, it also allows for much cleaner code.

13.14 Triple numbers

Sometimes you need to handle integers so big, they won't even begin to fit in a double number. Well, not all is lost - 4tH offers triple numbers as a last resort. But don't expect a nice and clean wordset, it is more like a toolbox. We'll tell you here how to use it.

First you have to include the appropriate library:

```
include lib/triple.4th
```

This gives you access to triple word arithmetic and comparison words. If you want to enter triple length words, you need this one:

```
include lib/totriple.4th
```

[16] See section 14.1.

And if you want to print a full triple word number, this one comes in handy:

```
include lib/trisharp.4th
```

Both I/O libraries will automatically load `triple.4th`.

BTW, you get what the name implies: facilities to construct your own triple word numeric output words. You won't find things like "T." or "UT.R" - you want anything like that, you'll have make them yourself.

Finally, you might want to include `3dup3rot.4th`, because that one provides three item stack operators like "3SWAP", "3DUP", etc. But if you already loaded `trisharp.4th`, you got that one automatically anyway.

Now we're loaded and ready. First, let's enter a triple number:

```
s" -295147905179352825855" s>triple
```

Now that's easy isn't it? If you want to convert a double number to a triple, you'll have to do some work, e.g.

```
0 constant ud>t
s" 4294967295" s>double ud>t
```

Adding or subtracting them is easy enough, just use "T+" or "T-". However, when you want to divide or multiply them, you will be confronted with a myriad of words like "UT*", "MT*" and "TU*" and nothing as simple as "T*".

The point is, that calculating the triple result of these operations becomes harder and harder each time a cell is added in order to increase the range of an integer. The best advice I can give you is to study them carefully and pick the most appropriate word for your particular situation.

If you want to print a triple word, you will have to put some work into it. Yes, the complexity here increases as well - e.g. setting a triple number up for signed output is so complex, we defined a special word for it, "(TSIGNED)". So if you want the triple word equivalent of "D.", this is it:

```
: t. (tsigned) <t# t#s tsign t#> type space ;
```

The reason we're not holding your hand here all the time is, that you will rarely need triple word precision - and if you do, you need to know 4tH inside out already in order to avoid errors and bugs. Most likely, you won't need all of the functionality of the triple word libraries - that's why we kept it lean.

13.15 Timers

There is a very low level word in 4tH that keeps track of time. It has several uses. Like a timer that measures how long certain operation takes, like the execution of a colon-definition ("DO-SOME-WORD" in this case):

```
time do-some-word time
swap -
." Do-Some-Word took " . ." seconds." cr
```

There is a somewhat more elaborate library member that does it all for you:

```
[needs lib/timer.4th]

timer-reset
do-some-word
.elapsed
```

This always prints the number of seconds that have elapsed. If you want to create your own display, you can define one easily:

```
[needs lib/timer.4th]

:noname <# # 6 base ! # decimal 58 hold # #> type ."  mins" ;
is timer-stop
```

You define ”TIMER-STOP” *after* inclusion of ”time.4th”, but *before* the first usage of ”.ELAPSED”.

13.16 Time & date

There is also an internal word in 4tH that will tell you what time and what date it is. With a little trouble ;). That word is called ’TIME’ and it will tell you how many seconds have gone since January 1st, 1970. That is the POSIX time format, which probably is of little use to you. However, combined with the appropriate library you can find out how late it is right now:

```
[needs lib/time.4th]

now ." hours:" . ." minutes:" . ." seconds:" . cr
```

Note that it doesn’t know about daylight-saving! It does know about timezones, which may be neccesary on *some* systems. You can determine your timezone by looking at an email message from a local friend. It will probably say somewhere:

```
Date: Mon, 25 Feb 2002 22:28:59 +0100 (CET)
```

The ’+0100’ means that you’re in timezone CET, which is one hour later than GMT. If it said:

```
Date: Sun, 16 Dec 2001 02:19:40 -0800 (PST)
```

This indicates that you’re in timezone PST, which is eight hours earlier than GMT. In that case ’tz’ would be:

```
-8 3600 * +constant tz          \ Pacific Standard Time
```

If you need it, define it accordingly *before* the inclusion of `time.4th`. There are also several words that will allow you to convert any POSIX time:

```
time posix>time . . . cr
```

Which will return the number of seconds (TOS), the number of minutes and the number of hours. ”POSIX>JDAY” will convert any POSIX time to a Julian day. The day of the week is another thing you can easily calculate:

```
[needs lib/time.4th]

: Weekdays
  dup 0 = if drop s" Monday"    exit then
  dup 1 = if drop s" Tuesday"   exit then
  dup 2 = if drop s" Wednesday" exit then
  dup 3 = if drop s" Thursday"  exit then
  dup 4 = if drop s" Friday"    exit then
  dup 5 = if drop s" Saturday"  exit then
  dup 6 = if drop s" Sunday"    exit then
;

today weekday Weekdays type cr
```

By the way, didn’t you hate the way we had to define "Weekdays"? Ugly, isn’t it? Well, there is a better way to do it. You’ll learn that in the next chapter (see section 12.17)! You can also print the full date:

```
[needs lib/time.4th]

today ." year:" . ." month:" . ." day:" . cr
```

Just don’t ask me how this thing works, Everett F. Carter figured this one out. ”TODAY” does the easy work. ”JDATE” converts the Julian day to the Gregorian date. There is also a way to convert a Gregorian date to a Julian day, called ”JDAY”:

```
[needs lib/time.4th]

26 02 2002 jday 64 - jdate
today ." year:" . ." month:" . ." day:" . cr
```

This can be quite handy if you want to calculate which date it was 64 days ago. ANS-Forth also defines a word that does it all called ”TIME&DATE”. This word throws seconds, minutes, hours, day, month and year (TOS) on the stack, but always returns GMT:

```
[needs lib/ansfacil.4th]

time&date . . . . . cr
```

And finally, we even got a word that returns the date of easter:

```
[needs lib/easter.4th]

2005 easterSunday ." year:" . ." month:" . ." day:" . cr
```

Well, tell me, isn’t that kind of neat?

13.17 Tokenizing strings

Sometimes you want to split up a string in several different parts. This is called ”tokenizing”. Doing it with 4tH is (as usual ;-) quite easy. Just include `tokenize.4th`. Now

you got several words to get what you want. `tokenize.4th` creates a deferred word[17] called "IS-TYPE". It decides whether a character is of a certain type. In this example, we just want to know whether it is a lowercase 'a':

```
include lib/tokenize.4th
:noname [char] a = ; is is-type
```

Now it's time to play ball:

```
s" 01234aBcDe01234" scan> type cr
```

"SCAN>" will now skip all characters unless it is an 'a'. When it is found, it stops and returns the remainder of the string:

```
aBcDe01234
```

Yes, "SCAN>" starts at the beginning of the string. But there is also a word that starts *at the end* of the string:

```
s" 01234aBcDe01234" scan< type cr
```

It returns a different result too:

```
01234a
```

And what about the rest of the string? Well, that is discarded. But if you need it, there is also a word that just splits up the string:

```
s" 01234aBcDe01234" split> type cr type cr
```

So, "SPLIT>" returns *two* strings:

```
01234
aBcDe01234
```

And of course, he's got a little brother that works the other way around:

```
s" 01234aBcDe01234" split< type cr type cr
```

"SPLIT<" returns two strings too:

```
01234a
BcDe01234
```

That was quite easy. But what if you want to find the *first non-digit*? That's what we got "SKIP>" for! "SKIP>" skips all characters of a certain kind. We've already seen how we can distinguish characters in section 8.26. So in this case we just got to include `istype.4th` and assign "IS-DIGIT" to "IS-TYPE":

[17] See section 11.7.

```
include lib/tokenize.4th
include lib/istype.4th

' is-digit is is-type

s" 01234aBcDe01234" skip> type cr
```

”SKIP>” returns a single string:

```
aBcDe01234
```

And he has a counterpart too:

```
include lib/tokenize.4th
include lib/istype.4th

' is-digit is is-type

s" 01234aBcDe01234" skip< type cr
```

May be this result will surprise you, although it is completely correct:

```
01234aBcDe
```

What exactly did we ask for? We wanted the *first non-digit*, starting from *the end.* ’e’ is the first non-digit, so ”SKIP<” is completely correct.

13.18 Regular expressions[18]

Regular expressions (also called ”regex”) represent a way to identify patterns in a text. They can be used to search for patterns. You can define simple ones in 4tH. They don’t even have to take up too much code. The library `tre.4th` for instance, only supports the metacharacters ”*****” and ”**.**”.

In regular expression syntax, a dot means “any single character.” So the regular expression ”b.b” will match “bib”, “bob”, “brb”, or “bub". It will also match two b’s with a space or a tab between them; the dot matches these “whitespace” characters also. The star means “zero or more of the previous character.” So the regex ”Bb*” will match “B”, “Bb”, “Bbb”, “Bbbb”, and so on. (Note that you must have the capital B, or the match will fail; the B must match before b* can try to). So, with library `tre.4th` you can define these queries:

```
s" aa" s" a*" match-reg . cr
s" ab" s" .*" match-reg . cr
s" aab" s" c*a*b" match-reg . cr
```

You can do some pretty sophisticated matching with these two already. If that isn’t enough, you may require `kpre.4th`. That one was originally written by Brian W. Kernighan and Rob Pike. It supports all the metacharacters listed in table 13.6.

The question mark means “zero or one of the previous character.” So the regex ”Bb?” will match “B” (one capital B, zero lowercase b) or “Bb” (one capital B, one lowercase b). The plus means “one or more of the previous character.” So the regex ”Bb+” will match “Bb”, “Bbb”, “Bbbb”, and so on, but will not match “B” by itself. A few other characters have special meanings in a regex:

[18] Part of this text is derived from *http://http://misc.yarinareth.net/regex.html* by Dorothea Salo under the ”Creative Commons Attribution 3.0 United States License”.

META CHARACTER	MEANING
.	Matches any single character.
^	Matches the starting position within the string.
$	Matches the ending position of the string or the position just before a string-ending newline.
*	Matches the preceding element zero or more times.
?	Matches the preceding element zero or one time.
+	Matches the preceding element one or more times.

Table 13.6: KPRE supported metacharacters

- The ^ character, if it appears at the start of a regex, means "at the beginning of a document."
- The $ character, if it appears at the end of a regex, means "at the end of a document."

These are a few of the expressions you can define with `kpre.4th`:

```
s" 0,9" s" ^0,?9$" match-reg .  cr
s" 0:9" s" ^0,?9$" match-reg .  cr
s" aaaaaabac" s" ab*a$" match-reg .  cr
s" abbd"      s" ab*d$" match-reg .  cr
```

You might exclaim "but where are my bracket expressions". Rightly so, but we don't have 'em. Instead we have predefined (non-standard) character classes. Regex engines allow you to use "character classes" that narrow a search to specific collections of characters. Think of it as a dot, but more specific.

In 4tH, character classes are prefixed with a percent sign. E.g. "**%_**" is about equivalent to "**\s**" and "**%&**" approximates "**\w**". But there is more, as you can see in table 13.7.

CHARACTER CLASS	EQUIVALENT	VIM
%9	[0-9]	**\d**
%a	[a-z]	**\l**
%A	[A-Z]	**\u**
%#	[A-Za-z]	**\a**
%&	[A-Za-z0-9]	**\w**
%_	[\x00-\x20]	**_s**

Table 13.7: KPRE character classes

With character classes, it is very easy to e.g. recognize a floating point number:

```
s" -1234.56" s" -?%9+\.?%9*$" match-reg . cr
s" -1234"    s" -?%9+\.?%9*$" match-reg . cr
s" 1234.56"  s" -?%9+\.?%9*$" match-reg . cr
s" 1234"     s" -?%9+\.?%9*$" match-reg . cr
```

But what if you want to find an actual period (or an actual plus, or an actual question mark, or an actual star) in a regular expression? The backslash signals to a regular expression that

the following character, if it has a special regular expression meaning, should be interpreted literally, not in its special regex sense. So "**.**" by itself means "any single character," while "**\.**" with the backslash means an actual period. This works for any character that has a special meaning in a regular expression: "*****" means an actual star, and "**\+**" means an actual plus. Of course, the backslash works on itself, too; to search for a literal backslash, a regex pattern must contain "****" .

The disadvantage of regular expression is that you have to drag along the entire pattern matching engine, which is written in high level code. Advantage is that you may change the expression you want to evaluate at runtime - you can even enter it at the prompt if you like.

13.19 String pattern matching

If that isn't enough for you, 4tH also offers a small library for string pattern matching. Most regular expressions are static, that means they're not entered at runtime, but hard-coded into the program. That means you're dragging along the entire engine just to parse a string. 4tH offers almost the same possibilities, but you have to do a *little* more effort. Let's say I want to see if a string contains my first name:

```
include lib/chmatch.4th

: myname?
  s" Hh" char-match >r
  s" Aa" char-match r> and >r
  s" Nn" char-match r> and >r
  s" Ss" char-match r> and
;

s" Hans is the creator of 4tH" myname? . type cr
```

Of course, there are easier ways to do this, but you'll see it works. "CHAR-MATCH" will return a "true" value if one of the characters specified matches and a "false" value if it doesn't. E.g.:

```
s" Hans" s" Hh" char-match
```

Will return a "true" value and "ans". On the other hand:

```
s" Hans" s" Aa" char-match
```

Will return a "false" value and "Hans". Now you understand how it works: "MYNAME?" chops off character by character until it is done. But what it I want to check whether "Hans" appears in the middle of a string? Easy:

```
include lib/chmatch.4th

: myname?
  s" Hh" skip-until
  s" Hh" char-match >r
  s" Aa" char-match r> and >r
  s" Nn" char-match r> and >r
  s" Ss" char-match r> and
;

s" In October 1994 Hans created 4tH" myname? . type cr
```

This will wind the string upto the first ”H” and then start scanning. But what if there’s another ”H” in the string? Again, you can handle that:

```
include lib/chmatch.4th

: myname?
  begin
    s" Hh" skip-until
    s" Hh" char-match >r
    s" Aa" char-match r> and >r
    s" Nn" char-match r> and >r
    s" Ss" char-match r> and >r
    dup 0<> r@ 0= and
  while
    r> drop
  repeat r>
;

s" The 4tH compiler was created by Hans in 1994" myname? . type cr
```

You want to know whether it was at the very end of the line? Yes, you can:

```
include lib/chmatch.4th

: myname?
  begin
    s" Hh" skip-until
    s" Hh" char-match >r
    s" Aa" char-match r> and >r
    s" Nn" char-match r> and >r
    s" Ss" char-match r> and >r
    dup 0<> r@ 0= and
  while
    r> drop
  repeat r> over 0= and
;

s" The 4tH compiler was created by Hans" myname? . type cr
```

Let’s try something more difficult now, let’s see if we can parse a number. The first part is very easy:

```
include lib/chmatch.4th

s" 0123456789" sconstant ’digits’
s" 0endshere" ’digits’ char-match . type cr
```

That works, the correct flag is returned and the string ”endshere” is returned. But a number may be of arbitrary size, how do we do that? Simple, after the first number we can simply discard any trailing digits:

```
include lib/chmatch.4th

s" 0123456789" sconstant ’digits’

s" 0123456789endshere"
’digits’ char-match >r
’digits’ skip-while
r> . type cr
```

Still, it returns the correct flag and the correct string. But a number may also come with an *optional* sign. Now that complicates life, does it? No it doesn't:

```
include lib/chmatch.4th

s" 0123456789" sconstant 'digits'
        s" +-" sconstant 'sign'

s" -123456789endshere"
'sign'   char-match drop
'digits' char-match >r
'digits' skip-while
r> . type cr
```

What? Aren't we evaluating the flag? No, why should we. The sign is *optional*, isn't it? Note that "CHAR-MATCH' not only sets a flag, but also *discards* the character matched. So it it *isn't* a sign, it will simply remain there and be caught by the digit test. But we'll take it a step further. What about a decimal point? A number may come with one or without one. And it may appear anywhere in the number. Now how we do that?

```
include lib/chmatch.4th

s" 0123456789" sconstant 'digits'
        s" +-" sconstant 'sign'
        char . constant  'point'

s" -123456.789endshere"
'sign'   char-match drop
'digits' char-match >r
'digits' skip-while
'point'  char-equal drop
'digits' skip-while
r> . type cr
```

When parsing stops, because the program reached the point where there are no more digits, we simply check for an optional decimal point. If it is there, we continue checking for digits. If it isn't there, parsing stops. Note that the check for trailing digits fails too, but you may make it optional if you want, it won't change the result:

```
include lib/chmatch.4th

s" 0123456789" sconstant 'digits'
        s" +-" sconstant 'sign'
        char . constant  'point'

s" -123456.789endshere"
'sign'   char-match drop
'digits' char-match >r
'digits' skip-while
'point'  char-equal if 'digits' skip-while then
r> . type cr
```

Note that "CHAR-EQUAL" performs the same function as "CHAR-MATCH", it's only more efficient if there is only one character to compare[19]. Our program still doesn't allow for "-.45". Yes, you can add an optional test for that, but we don't want "-.45.67" to succeed. That one can be handled too:

[19] If you want to speed up a single character search, `scanskip.4th` offers the words you're looking for.

```
include lib/chmatch.4th

s" 0123456789" sconstant 'digits'
        s" +-" sconstant 'sign'
        char . constant  'point'

s" -.789.000endshere000."
'sign'   char-match drop
'point'  char-equal >r
'digits' char-match r> swap >r >r
'digits' skip-while
r> 0= if 'point' char-equal if 'digits' skip-while then then
r> . type cr
```

Now the test will only be performed if the previous test failed, which is exactly what we want. Try it. Where ever you place the dot, it won't make any difference: it will *always* return ".000endshere000.". Unfortunately, that also applies to:

```
s" -0..000endshere000."
```

If you require that a decimal point is *always* followed by a digit, simply enforce it:

```
include lib/chmatch.4th

s" 0123456789" sconstant 'digits'
        s" +-" sconstant 'sign'
        char . constant  'point'

s" -0..000endshere000."
'sign'   char-match drop
'point'  char-equal >r
'digits' char-match r> swap >r >r
'digits' skip-while r> 0=

if
  'point' char-equal
  if
    'digits' char-match r> and >r
    'digits' skip-while
  then
then

r> . type cr
```

Note this *doesn't* change the string returned (up to the trailing point it was still a valid number), but it *does* change the flag: this is *not* a valid number!

I hope you see that 4tH can do some pretty sophisticated pattern matching with very little effort without dragging along an entire regular expression engine, either in code or in the compiler itself.

13.20 Wildcard pattern matching

If you feel that regular expressions are a bit too complex for your taste, you can also use wildcards for basic pattern matching. First you have to include it:

```
include lib/wildcard.4th
```

Or:

```
include lib/wildcrdr.4th
```

For the recursive version, which is a bit shorter. You've probably used wildcards before. It is very easy. A "*" stands for *zero or more characters* and a "?" stands for *a single character*. E.g. if you're looking for a line that begins with a date in the 21st century, then the word "INVOICE" and finally ends with a name, you could try this:

```
mystring count s" 20??-??-??  == INVOICE ==  *" match-wild
```

"MATCH-WILD" returns true if the string matches and false if the string doesn't match, which is different from "COMPARE". Both strings are consumed, so save their address/count pair if you need them later on. Note that "MATCH-WILD" is *case sensitive*, so if you need a case insensitive comparison you will have to convert them first.

Finally, "MATCH-WILD" is faster than regular expressions, but also less precise. E.g. "gr?y" will not only return "grey" and "gray", but also "groy"; "reg*exp*" will not only return "regular expressions", but also "registered express mail".

13.21 Escape characters

Sometimes you need a string with embedded control characters. You could try to patch them into your sourcecode, but that is usually not a good idea. You could also patch them into a variable, but that is cumbersome. 4tH offers a word that will allow you to use the control characters listed in table 13.8.

ESCAPE CHARACTER	MEANING
\a	Bell
\b	Backspace
\e	Escape
\f	Formfeed
\l	Linefeed
\n	Linefeed
\q	Quote (")
\r	Carriage return
\t	Tab
\v	Vertical tab
\z	Null character

Table 13.8: Supported control characters

Using this facility is quite simple:

```
include lib/escape.4th
s" \tThis is the \qC:\\4tH\q directory" s>escape type cr
```

Which will print:

```
        This is the "C:\4tH" directory
```

Note that other escaped characters are printed ”as is”, like the backslash in this example. If you need even more flexibility, we have another card up our sleeve. The only catch is you'll have to know the ASCII code of the characters you want to insert, either in binary, octal, decimal or hex:

```
include lib/embed.4th
s" %1001This is the&42C:\4tH#34$20directory" s>embed type cr
```

Which will also print:

```
This is the "C:\4tH" directory
```

- Every number prefixed with a '**%**' character is interpreted as a binary ASCII code;
- Every number prefixed with '**&**' character is interpreted as an octal ASCII code;
- Every number prefixed with a '**#**' character is interpreted as a decimal ASCII code;
- Every number prefixed with a '**$**' character is interpreted as a hexadecimal ASCII code.

How much flexibility do you need? Of course, all this flexibility comes at a price. The strings are not expanded at compile-time but at *run-time*, so it will cost you a little more space and a little more time compared to other string literals.

13.22 Chinese characters

There are several encodings for Chinese characters, e.g. Unicode, GBK and Big5. As long as the ASCII-7 range of characters is respected, 4tH is able to handle them. 4tH can even convert between some of these encodings. The `unicdgbk.4th` file forms the centerpiece of this functionality. Basically, it offers conversion between Unicode and GBK/2. Since it is *very* large, it isn't included automatically.

4tH offers conversion between UTF-8 and GBK/2 - and vice versa. GBK/2 offers about 7,500 characters. UTF-8 is more prevalent under Linux, while a GBK/2 derivate is more prevalent under Windows. Using the UTF-8 to GBK/2 conversion couldn't be easier:

```
include lib/unicdgbk.4th
include lib/utf8gbk2.4th

s" Some UTF-8 encoded Chinese characters"
utf8>gbk2 type cr
```

Note this is a *destructive* conversion. After this, your original string is gone. If you want to preserve it, you have to save it first to another string variable. The string it returns is *not* terminated, so if you're converting a string variable this might be a good idea:

```
include lib/unicdgbk.4th
include lib/utf8gbk2.4th
64 string mystring

s" Some UTF-8 encoded Chinese characters"
mystring place

mystring count utf8>gbk2 >string
```

This ”in place” conversion is possible, because a GBK/2 encoding *never* requires more space than an UTF-8 encoding. However, the opposite isn't true, so depending to the size of your original GBK/2 source string, you'll need a larger UTF-8 buffer, e.g.:

```
include lib/unicdgbk.4th
include lib/gbk2utf8.4th
64 string myfirststring
128 string mysecondstring

s" Some UTF-8 encoded Chinese characters"
myfirststring place

myfirststring count mysecondstring 128 gbk2>utf8
>string mysecondstring count type cr
```

Note you don't need any external libraries or programs to accomplish this. 4tH is all you need.

13.23 Managing INI files

An INI file is a plain ASCII file that is used to configure software. It was designed by Microsoft and dates back to the times of MS-DOS. However, you will find it as well nowadays on Linux and Unix systems, because INI files are human-readable and simple to parse.

It's for that reason we chose to support them in 4tH. As a matter of fact, we created two distinct libraries. The first and the most simple one just offers an interface to evaluate an INI file.

It is quite simple. Let's say you just want to store the values in the INI file in your program. First you create a few words to achieve that goal. Note that values are presented to you as strings and your word must consume them entirely:

```
include lib/inifile.4th

variable num1
16 string str1

: getnum1 number num1 ! ;
: getstr1 str1 place ;
```

Then you create a table with the following layout: first the section, then the key and finally the execution token of the word that must process the value:

```
create myini
  ," config" ," number1" ' getnum1 ,
  ," init" ," string1" ' getstr1 ,
  NULL ,
```

And finally you just send the name of this table and the filename (with path if required) to "INIFILE". That's all!

```
myini s" myini.ini" inifile
```

However, if you want to create or modify an INI file, you need something more sophisticated. It's time for the INI file management library. Note that this piece of software is highly dependent on dynamic memory, so if your program is already using the heap or if you want to manage a *really* big INI file, you may want to increase the size of the heap(s).

But first, let's define a new INI file. Of course, you do have to include the library:

```
include lib/inimanag.4th
```

Then you have to signal you want to define an INI file from scratch:

```
begin.inifile
```

Next, define a new section. Yes, you can do without, but I don't think it is really neat:

```
s" Compiler" section.inifile
```

And now you can define your keys and values. Note, *first* the value *then* the key. Let's not forget we're using Forth ;-)

```
s" 4tH" s" Name" keypair.inifile
s" 3.62.5" s" Version" keypair.inifile
s" Forth" s" Language" keypair.inifile
```

Of course, you can start a new section and add some more keys:

```
s" Platform" section.inifile
s" Linux" s" Name" keypair.inifile
s" 4.1" s" Version" keypair.inifile
```

And when you're finally done, you can save it to disk with a single line:

```
s" myfile.ini" put.inifile abort" Cannot write .INI file"
destroy.inifile
```

These are two distinct actions. The first line saves the file to disk. The second line removes the INI file from memory. So yes, you can save your file preliminary and continue working with it - as long as you don't issue "DESTROY.INIFILE", of course.

Reading it back is just as simple:

```
inifile get.inifile abort" Cannot read .INI file"
```

Just don't forget that both "PUT.INIFILE" and "GET.INIFILE" return a non-zero flag when there's any trouble. Ok, we got this thing in memory now, let's make some changes:

```
s" Version" s" Platform" delkeypair.inifile
abort" Cannot delete key from named section"
```

That one deleted a key. It also returns a non-zero flag on error, e.g. because the key or the section could not be found.

```
s" J.L. Bezemer" s" Author" s" Compiler" addkeypair.inifile
abort" Cannot add key to section"
```

This one adds a key to a certain section. You can also use it when defining an INI file, of course, but you can save yourself a lot of typing by using "KEYPAIR.INIFILE" instead.

```
s" Hans Bezemer" s" Author" s" Compiler" set.inifile
abort" Cannot modify key in named section"
```

And of course, you can modify a value as well - initials are so formal. Finally, you can write it to disk - we've seen that one before. Yes, you can also retrieve a value with this library:

```
s" Author" s" Compiler" value.inifile
```

But you have to write your own handling of the string it returns (address and length). And if that's the only reason you're using this library, you're creating a lot of overhead and you'd be better off with `inifile.4th`.

13.24 Extract, transform and load

"Extract, transform and load" is the name of one of the most used conversion methods. First you rip the data from the source, then you manipulate it and finally you move it to its destination. Well, although what we do is more appropriately called "Extract, load and transform" you can apply the same principles when reading a .CSV file with 4tH. It is very handy if you don't know the exact layout of the file, but do have a header line with the correct field names[20]. Of course, first thing you have to do is to include the library:

```
include lib/etl.4th
```

The data has to end up somewhere, so next thing to do is to define the layout and allocate enough space for a record:

```
struct
  16 +field Population
  32 +field State
  16 +field Capital
buffer: States
```

You know I prefer to define a word to parse a field[21], so let's do that now:

```
: field> [char] ; parse-csv csv> ;
```

Now the real fun starts. We have to make a table that maps the column header of the .CSV file to the fieldname of the structure. On top of that, we have to determine if we want to manipulate that field and how:

```
create Mapper
  ," Population" ' Population , ' >comma ,
  ," State"      ' State ,      ' none ,
  ," Capital"    ' Capital ,    ' >upper ,
  NULL ,
```

[20] See RFC 4180, section 2.3
[21] See section 9.23.

Note our field is maniplated after it has been placed in the structure - not before. The reason is simple. We don't want to clobber our source record - which could easily happen when a field expands.

You probably wonder where ">COMMA", "NONE" and ">UPPER" come from. Well, these words manipulate our fields. They all take the address and length from their respective fields and return nothing. E.g. "NONE" could easily be defined as:

```
: none 2drop ;
```

Next, 4tH needs to know how much lines there are in "MAPPER". Of course, you could define:

```
3 constant #etl
```

But you could also let 4tH figure it out for you:

```
here Mapper 1+ - /fieldmap / constant #etl
```

Finally, we need to define the "transformation array" this library requires:

```
#etl /etl * array Transpose
```

Now we're ready to rock 'n roll. Let's read the header:

```
refill 0= abort" Cannot read header"
Transpose Mapper ['] field> read-header
```

Yes, we feed it the "transformation array", the mapping and an execution token to read a field. That's all. After it has executed, the address of the "transformation array" and its actual length are on the stack and 4tH truly knows how to read your file. All you have to do is call "READ-TUPLE" for each following line:

```
2dup States ['] field> read-tuple
```

"READ-TUPLE" takes four parameters: the address of the "transformation array", its length, the address of the target structure and an execution token to read a field. That's all to need to read an entrire record. Note the first two stack items are provided by "READ-HEADER".

You may be asking yourself why you should use this. Well, this library does't care in what order the fields are presented and simply ignores any superfluous ones. With this you are immune to slight changes in the layout of your .CSV files. That's why!

13.25 Writing spreadsheet files

4tH is able to write to a variety of spreadsheet formats, both proprietary and FOSS (see table 13.9). All these formats are known to work with OpenOffice 3.x, Libreoffice 4.x and MS-Excel 20xx. KOffice 1.x can read `.ksp` files. The interface is almost identical and very straight forward:

	Type	*Extension*	*4tH library*	*Worksheets?*	*Integers only?*
XLS 2.1	Binary	.xls	msxls2-w.4th	No	Yes
FODS	XML	.fods	oofods-w.4th	Yes	No
MS-XML	XML	.xml, .xlsx	msxlms-w.4th	Yes	No
KSP	XML	.ksp	koksp-w.4th	Yes	No
CSV	CSV	.csv	csv-w.4th	No	No
HTML	HTML	.html	html-w.4th	No	No

Table 13.9: Spreadsheet formats supported by 4tH

- Open a spreadsheet file;
- Start writing at cell A1;
- If you're finished, go to the next row;
- Write some more cells;
- All done, close the file.

Let's assume you want to write a flat Open Document Sheet. First you have to open the file:

```
include lib/oofods-w.4th
s" mysheet.fods" FODSopen
```

It will return a flag, which is true when an error has occured. No, you don't have to specify it's an output file - you can't read it anyway. It won't return a handle either, the library will take care of that. Note that the moment you've successfully opened the file, it is active. If you want to use another output file, you have to 'USE' it first. And every time you've written to your spreadsheet you have to repeat that procedure. Now, let's handle any errors:

```
abort" Cannot open spreadsheet"
```

Since flat Open Document Sheet supports different sheets, we have to open a worksheet first:

```
s" Sheet1" FODSsheet
```

We're ready to do business now. Let's punch out a few labels. We're starting in cell A1 and continue with cell B1 and C1:

```
s" Label1" FODStype
s" Label2" FODStype
s" Label3" FODStype
```

Let's continue with row 2 and write some data:

```
FODScr
1 FODS.
2 FODS.
3 FODS.
```

We move to row 3 and repeat the same procedure:

```
FODScr
100 FODS.
450 FODS.
325 FODS.
```

Finally, we finish the worksheet and close the file:

```
FODSend
FODSclose
```

	A	*B*	*C*
1	Label1	Label2	Label3
2	1	2	3
3	100	450	325

Table 13.10: Example spreadsheet

That's all! You got a spreadsheet like the one in table 13.10. Writing other formats is just as easy. Take a look at table 13.11 which words are available for which format.

	msxls2-w.4th	*oofods-w.4th*	*msxlms-w.4th*	*koksp-w.4th*	*csv-w.4th*	*html-w.4th*
OPEN	XLSopen	FODSopen	XMLSopen	KSPopen	CSVopen	HTMLopen
sheet	-	FODSsheet	XMLSsheet	KSPsheet	-	-
.	XLS.	FODS.	XMLS.	KSP.	CSV.	HTML.
#	-	FODS#	XMLS#	KSP#	CSV#	HTML#
TYPE	XLStype	FODStype	XMLStype	KSPtype	CSVtype	HTMLtype
CR	XLScr	FODScr	XMLScr	KSPcr	CSVcr	HTMLcr
atxy	XLSatxy	-	-	KSPatxy	-	-
end	-	FODSend	XMLSend	KSPend	-	-
CLOSE	XLSclose	FODSclose	XMLSclose	KSPclose	CSVclose	HTMLclose

Table 13.11: Spreadsheet words

13.26 Writing other tabular formats

There are a few more tabular formats 4tH can write although they cannot be considered "spreadsheets", since there is no major spreadsheet program that actually supports them. However, the API of the library members that support them follows the conventions of those listed in section 13.25.

	json-w.4th	*create-w.4th*
OPEN	JSONopen	CREATopen
.	JSON.	CREAT.
#	JSON#	CREAT#
TYPE	JSONtype	CREATtype
CR	JSONcr	CREATcr
CLOSE	JSONclose	CREATclose

Table 13.12: Other tabular formats

The first format is a new kid in town, called JSON. The second one is able to write a complete 4tH table, the kind you find in section 12.17, which is great for making e.g. program generators.

13.27 Writing LaTeX files

You can write LaTeX files with 4tH very easily. The files it produces can be processed by most popular LaTeX typesetting programs. Although 4tH only supports a tiny LaTeX subset, one could typeset most of this manual with it. If you don't know any LaTeX, don't worry: you don't need to. As a matter of fact, it is very much like HTML.

So let's start by including the library:

```
include lib/latex.4th
```

Next we have to decide what kind of document we want to write. Let's say we want to write an entire book:

```
%texBook
```

If you are a little less ambitious, try writing an article:

```
%texArticle
```

Now you have to give the document a title and claim your rightful place as the author. The flag decides whether you get a "table of contents". If you want one, supply a TRUE flag:

```
%beginTex
s" Hans Bezemer" s" My life with 4tH" true %texTitle
```

Define your first section and you're off:

```
s" Foreword" %subSection
s" First, I want to thank my parents, my teachers," %print
s" my girlfriend, my hairdresser and most of all" %print
s" my dog Snoopy, who never lost faith in me." %print
```

Ok, that was good enough for a Pulitzer Prize. Let's go on with the next section:

```
%endSection

s" The beginning" %subSection
s" It was late at night, I couldn't catch sleep." %print
s" So I decided to create a compiler in order to" %print
s" archieve world peace and love and harmony for" %print
s" all mankind." %print
```

That's enough babble, now let's make a subsection. The difference between making a section and a subsection is that if you want to make a new section you close it with "%endSection". If you want to make a subsection, you don't:

```
s" Basic philosophy" %subSection
s" To make the best product on the market, give the" %print
s" best possible service to our paying customers and" %print
s" squeeze each and every penny out of their pockets." %print
```

Note you have have to balance the "%subSection" and "%endSection" as if they were loop control words. Now let's make a numbered list:

```
%endSection
%endSection

s" Pedigree" %subSection
%enumerate begins
  s" Artic Forth" %item
  s" SpecForth" %item
  s" gForth" %item
%enumerate ends
```

You can also make descriptions:

```
%description begins
  s" Word" s" A function in Forth." %describe
  s" Flag" s" A boolean in Forth." %describe
  s" Cell" s" A native integer number in 4tH." %describe
%description ends
```

You can even insert source code:

```
%listing begins
  ." : rot >r swap r> swap ;     \ define ROT" cr
  ." 2 3 4 rot                   \ application of ROT" cr
%listing ends
```

Even tables are not much of a problem:

```
%table begins
```

Now you have to decide on a layout. For that you have to compose a string, using a combination of the characters in table 13.13.

CHARACTER	MEANING
l	left justified column
c	centered column
r	right justified column
|	vertical line
||	double vertical line

Table 13.13: LaTeX table format

Let's make this table two left justified columns and one centered column, separated by single vertical lines:

```
s" | l | l | c |" %layout %line
```

The "%line" word draws a single horizontal line. Now we're ready to insert the cells:

```
s" Artic Forth" s" Spectrum" s" 1983" 3 %cells %line
s" FPC"         s" MS-DOS"   s" 1991" 3 %cells %line
s" 4tH"         s" Linux"    s" 1994" 3 %cells %line
```

The "%cells" word needs to know how many cells you're going to insert. Since three strings were supplied, three strings need to be declared. They will be printed in the order you specified, *not* the stack order. Finishing a table is just as uneventful:

```
%table ends
```

If you need links, that's supported too, but it needs the "hyperref" package. If you don't know if that one is installed, enter:

```
kpsewhich hyperref.sty
```

If it is, you may write something like:

```
s" Ok, guys. If you want to know more just" %print
s" google" s" http://www.google.com" %link
s" it! I don't have the time for this." %print
```

You have to enable it, though. All it needs is a '[PRAGMA]' at the beginning of your program:

```
[pragma] UseURLs
```

Ok, that's enough writing for tonight, let's end this:

```
%endSection
%endTex
```

When you run this program you will find it writes to the screen. That's because you have to take care of all the filehandling yourself; 4tH only takes care of the proper LaTeX generation. That may not seem like much, but note you've just combined a typesetting language with a powerful programming language, which enables you to handle the most complex document generation tasks with ease.

If you have installed pdfTeX[22] you can make beautiful PDF files out of the box. For those who need to generate office documents, keep an eye on the LaTeX2RTF[23] converter. Both are fully compatible with the LaTeX 4tH generates and easily integrate with 4tH[24].

13.28 Writing RTF files

If you find generating LaTeX is dead easy, you're in for a treat because writing RTF is not much different. There is a catch, though. You can write either RTF *or* LaTeX files, both libraries are incompatible - but don't worry, 4tH will catch that error for you. Like LaTeX, you start with including the library and the kind of document you want to write.

```
include lib/rtf.4th

%RTFbook
%beginRTF
  s" Hans Bezemer" s" My life with 4tH" false %rtfTitle
```

Note the flag is only there for LaTeX compatibility - it doesn't do anything. After that you're in business and if you know how to do a LaTeX file, this should come as no surprise:

[22] http://www.tug.org/applications/pdftex/
[23] http://latex2rtf.sourceforge.net/
[24] Provided your 4tH implementation supports pipes. MS-DOS is not supported.

```
s" Foreword" %subSection
  s" First, I want to thank my parents, my teachers," %print
  s" my girlfriend, my hairdresser and most of all" %print
  s" my dog Snoopy, who never lost faith in me." %println
```

Note RTF is a bit quirky where linebreaks are involved, so it's good practice to terminate the last line of a paragraph with "%println" instead of "%print". If you ever need additional linebreaks, you can add them with "%cr". Now let's make a numbered list:

```
%enumerate begins
  s" Artic Forth" %enumeration
  s" SpecForth" %enumeration
  s" gForth" %enumeration
%enumerate ends
```

Note that the items of a numbered list are terminated by "%enumeration" instead of "%item". You can also make descriptions:

```
%description begins
  s" Word" s" A function in Forth." %describe
  s" Flag" s" A boolean in Forth." %describe
  s" Cell" s" A native integer number in 4tH." %describe
%description ends
```

Yep, completely identical. Links, listings, sections, quotes, lines, linebreaks, you name it: it's all the same. Tables are only *slightly* different[25]:

```
%table begins
  s" Artic Forth" s" Spectrum" s" 1983" 3 %cells
  s" FPC"         s" MS-DOS"   s" 1991" 3 %cells
  s" 4tH"         s" Linux"    s" 1994" 3 %cells
%table ends
```

Ok, that's enough writing for tonight, let's end this:

```
  %endSection
%endRTF
```

Big surprise.. If you use this library properly it will generate fully *RTF Specification 1.6* compliant files. Like LaTeX, when you run this program you will find it writes to the screen. That's because you have to take care of all the filehandling yourself; 4tH only takes care of the proper RTF generation.

13.29 Writing HTML files

If you find generating LaTeX and RTF is dead easy, you're in for a treat because writing HTML is not much different. There is a catch, though. You can write either RTF *or* HTML *or* LaTeX files, all three libraries are incompatible - but don't worry, 4tH will catch that error for you. Like LaTeX and RTF, you start with including the library. After that you go *straight* to the title, after you've defined *one* (and only one) *optional* stylesheet:

[25] Although you can get away with a lot of LaTeX specific stuff, since it's simply ignored. Only `%line` is problematic.

```
include lib/html.4th
%beginHTML
  s" mystyle.css" %styleSheet
  s" Hans Bezemer" s" My life with 4tH" false %htmlTitle
```

Note the flag is only there for LaTeX compatibility - it doesn't do anything. Because if you want to do fancy stuff, you will need to do it with CSS - that's the only way. For arguments sake, let's say you want a very plain-looking table - then define a CSS file `mystyle.css` like this:

```
div.plain table { width: 50%; }
div.plain td { padding: 5px 5px 5px 0; vertical-align: top; }
div.plain th { padding-right: 10px; text-align: left; }
```

Keep that in mind, we'll come back to it later. After that you're in business and if you know how to do a LaTeX file, this should come as no surprise:

```
s" Foreword" %subSection
  s" First, I want to thank my parents, my teachers," %print
  s" my girlfriend, my hairdresser and most of all" %print
  s" my dog Snoopy, who never lost faith in me." %print
```

Now let's make a numbered list:

```
%enumerate begins
  s" Artic Forth" %item
  s" SpecForth" %item
  s" gForth" %item
%enumerate ends
```

Yep. Completely identical. You can also make descriptions:

```
%description begins
  s" Word" s" A function in Forth." %describe
  s" Flag" s" A boolean in Forth." %describe
  s" Cell" s" A native integer number in 4tH." %describe
%description ends
```

Again, no different. Links, listings, sections, quotes, lines, linebreaks, you name it: it's all the same. Tables are only *slightly* different[26]:

```
%table begins
  s" Artic Forth" s" Spectrum" s" 1983" 3 %heads
  s" FPC"         s" MS-DOS"   s" 1991" 3 %cells
  s" 4tH"         s" Linux"    s" 1994" 3 %cells
%table ends
```

Remember the stylesheet we made? What we actually did was defining a `DIV` class, which is a blocktag as a matter of fact. Within that class you can format whatever tag you like. You're not limited to tables, it can be anything. Now let's apply this class to the table we just defined:

[26] Although you can get away with a lot of LaTeX specific stuff, since it's simply ignored. Only `%line` is problematic.

```
s" plain" %setStyle
  %table begins
    s" Artic Forth" s" Spectrum" s" 1983" 3 %heads
    s" FPC"         s" MS-DOS"   s" 1991" 3 %cells
    s" 4tH"         s" Linux"    s" 1994" 3 %cells
  %table ends
%endStyle
```

You'll see that the format we defined is applied to the table, but after "%endStyle" it returns to the default format. This way you can apply all sorts of styles to different sections of your document. If you defined default styles in your stylesheet they're automatically applied to your document and you don't have to do anything. Needless to say, if you don't use a stylesheet[27] it will be a pretty *plain* looking document altogether.

Ok, that's enough writing for tonight, let's end this:

```
  %endSection
%endHTML
```

Big surprise.. If you use this library properly it will generate fully *HTML 4.01 Transitional* compliant files. Like LaTeX, when you run this program you will find it writes to the screen. That's because you have to take care of all the filehandling yourself; 4tH only takes care of the proper HTML generation.

13.30 Converting to XML and HTML

You can't ignore XML and HTML nowadays and 4tH also supports these formats. Let's say you want to write XML. The problem with XML is that there are several characters which aren't allowed in XML files, like apostrophes, quotes and ampersands. How do you know you're dealing with a special character? Well, that's easy:

```
include lib/istype.4th    \ for XML detection
include lib/asciixml.4th  \ we'll need this later
                          \ string with XML characters
s" Here's Johnny! >>> (Johnny)"
bounds ?do                \ let's scan it for XML
  i c@ dup emit is-xml    \ check for XML
  if ."  Yes, it's XML" else ."  Nope" then cr
loop                      \ all done
```

Ok, now we know which characters are XML. But how do we convert them to XML character entities? That is where the second library comes in:

```
include lib/istype.4th    \ for XML detection
include lib/asciixml.4th  \ we'll need this later
                          \ string with XML characters
s" Here's Johnny! >>> (Johnny)"
bounds ?do                \ let's scan it for XML
  i c@ dup is-xml         \ check for XML
  if ASCII>XML type else emit then
loop cr                   \ all done
```

Note that "ASCII>XML" will *always* convert a character to an XML character entity, that's why you need "IS-XML" to determine whether it needs conversion or not.

[27] You will find a stylesheet `html4th.css` in the standard distribution.

If you're converting an ISO-8859-1 or a CP437 encoded document to HTML, things get even more complicated since you'll have to take even more special characters into consideration. Fortunately, 4tH has all the tools you need to get the job done. First, we'll have to determine whether we're dealing with a special character or not. Second, if we *did* find a special character we'll have to convert it.

```
include lib/istype.4th    \ for XML detection
include lib/westhtml.4th  \ the conversion table
include lib/cp437htm.4th  \ the conversion word
                          \ string with HTML characters
: convert-line            ( a n --)
  bounds ?do              \ let's scan it for XML
    i c@ dup is-html      \ check for XML
    if CP437>HTML type else emit then
  loop cr                 \ all done
;
                          \ read and convert a file
: convert-file            ( --)
  begin
    refill                \ read a line
  while                   \ while not end-of-file
    0 parse convert-line  \ parse line and convert it
  repeat
;
( Opening files and writing headers, don't bother..)
```

The library `westhtml.4th` contains the ISO-8859-1 and CP437 conversion tables. Because of its size, you'll have to include it *manually* before you include the `cp437htm.4th` or `i8859htm.4th` libraries. "IS-HTML" works just like "IS-XML", but also reports a HTML character entity when it's not printable or simply outside the ASCII-7 range.

Like "ASCII>XML" the "CP437>HTML" word *always* returns a HTML character entity, even when there is no real equivalent to HTML. E.g. the first CP437 block character renders to "°". If you don't want that you'll either have to maintain an additional table or examine the character entity "CP437>HTML" returns. Needless to say that "ISO8859>HTML" works the same way. The only exception is the NULL character, which is not allowed at all. In that case an empty string is returned.

13.31 Databases

4tH also has a small database package which perfectly blends in with the language. First define a structure. That will be your database buffer:

```
include lib/dbm.4th

struct
  16 +field Firstname
  32 +field Lastname
end-struct /Person

/Person buffer: (Person)
```

Second, create a database file. You have to do this only *once*. If you do it again, it will create another *empty* database file, so take care!

```
s" Persons.dbm" db.create
```

Now add it to the data dictionary. All you have to do is to declare the buffer, its size and the database file. It will return a database handle.

```
(Person) /Person s" Persons.dbm" db.declare to Person
```

After that, you have to tell the datadictionary that you want to use it - or make it current, if you prefer:

```
Person db.use
```

Now we're ready to add records. Simply clear the buffer, fill in the fields and insert the record. When you're done, repeat the operation:

```
db.clear
s" Hans"    db.buffer -> Firstname place
s" Bezemer" db.buffer -> Lastname  place
db.insert

db.clear
s" Chuck"  db.buffer -> Firstname place
s" Moore"  db.buffer -> Lastname  place
db.insert
```

Finally, when you're done simply shut it down:

```
db.close
```

That's all! If you want to use it again, simply define the buffer and declare it to the datadictionary:

```
include lib/dbm.4th

struct
  16 +field Firstname
  32 +field Lastname
end-struct /Person

/Person buffer: (Person)

(Person) /Person s" Persons.dbm" db.declare to Person
```

Of course you can add additional structures and files, the procedure remains the same. If you want to use a table, simply use it:

```
Person db.use
```

Since this is the first time you are accessing the table it is automatically positioned at the first (sequential) record. Just fetch the values:

```
db.buffer -> Firstname count type space
db.buffer -> Lastname count type cr
```

`db.buffer` always points to the current table. Of course, you can also address the buffer directly - that is up to you:

```
(Person) -> Firstname count type space
(Person) -> Lastname count type cr
```

If you explicitly want to go to and fetch the first sequential record, you can also issue:

```
db.first
```

The next (undeleted) sequential record is reached by issuing:

```
db.next
```

If you want to know which record you're at, you can consult `db.rowid`:

```
db.rowid . cr
```

This number is important, because it *never* changes and uniquely identifies a record. If you want to return to a specific record, you can by using this number:

```
1 db.goto
```

Note that whether you access the record random or sequentially, the buffer is *always* updated accordingly. You can also search for a specific string:

```
db.first
db.key Firstname s" Hans" db.find
```

This will return you the first record containing "Hans". In order to find the *next* record simply issue:

```
db.key Firstname s" Hans" db.find
```

If there isn't a second "Hans", the database will return an *End of file* error. You can query the database by using `db.error`:

```
db.error EDB.EOF =
```

You'll find a full list of error codes in the source. You can print the error message by using `db.message`:

```
db.error db.message
```

If you want to delete the current record, simply issue:

```
db.delete
```

Note the record is invalidated at that moment and you will have to navigate away from it. If you want to change the current record, simply make the changes to the buffer and issue `db.update`:

```
s" Chuck" db.buffer -> Firstname place
db.update
```

Finally, you can switch between tables at all times. E.g. you can make changes to a ”Profession” table and return to the ”Person” table at any time. The ”Person” table will be untouched:

```
Profession db.use
( Here we consult or change the ”Profession” table)
Person db.use
( As if we never left it..)
```

Note that `db.close` closes the entire database system, not just the current table. However, if we want to revive it all you have to do is to `db.use` it. You will start at the first record though.

13.32 Indexing a database

Sometimes neither random nor sequential access is sufficient: you want to retrieve the records in order. 4tH supports simple indexes, but don't try to maintain millions of records with it, because are maintained in dynamic memory. If the default allocation is not enough, you can tweak it[28] before including the library file.

Having said that, it is not very difficult. Let's say you already have a database and you want to index it. First you have to decide how many records have to be indexed. Second, which sorting routine do you want to use. You can also use any address based sorting algorithm[29] if you want to, the compiler will figure it out:

```
include lib/shelsort.4th
include lib/dbmidx.4th
```

Now open your database as usual:

```
(Person) /Person s" Persons.dbm" db.declare to Person
Person db.use
```

Then you have to index your database. You have to do this only once:

```
8192 {char} db.key Firstname idx.create
abort" Can't make index"
to Pers.First
```

What actually happened is that 4tH stored all the rowids in dynamic memory and sorted them on the field ”Firstname”. In this case you can index a database table with up to 8192 records. The ”{CHAR}” word tells the index to treat it as a string when sorting or searching. The word ”{CELL}” is available for numeric fields, but you can add your own ”field type” if you need to. Access it like this:

```
Pers.First idx.first
```

And it fills the buffer with the very first record. The subsequent record can be reached by issuing:

[28] See section 12.7
[29] See section 12.12.

```
Pers.First idx.next
```

And so on. But how do you know you've reached the last record? Easy, the word "IDX.ERROR" will be set to non-zero. So printing a records in the proper order is as easy as:

```
Pers.First dup idx.first begin
  dup idx.error 0= while
  db.buffer -> Firstname count type cr
  dup idx.next
repeat idx.clear
```

Note you can clear any errors by invoking "IDX.CLEAR". The word "IDX.KEY" will return the address of the key field in the current database buffer. E.g. this is perfectly valid:

```
Pers.First idx.key count type cr
```

In this example it would print the "Firstname". Of course, you will rarely create an index on an already filled table. Normally, you will define your table, define your indexes and off you go. Sure, you can do that too:

```
s" Persons.dbm" db.create
(Person) /Person s" Persons.dbm" db.declare to Person
Person db.use

8192 {cell} db.key Firstname idx.init
abort" Cannot create 'First' index" to Pers.First
8192 {char} db.key Lastname idx.init
abort" Cannot create 'Last' index" to Pers.Last
```

Note you *have to* declare indexes when the table is actually "used" at the time, otherwise it will not be properly initialized. The snippet above will create two empty indexes along with an empty database. If creating an index fails, it will return a "true" flag. In any case it will return a reference to the index created, which you will need to pass to almost any word in the index library. Finally, you *don't need* to include a sorting routine, because there won't be any sorting.

13.33 Binding the indexes

Ok, we got two indexes now. How do we update 'em? Well, like this:

```
db.clear
s" Hans"    db.buffer -> Firstname place
s" Bezemer" db.buffer -> Lastname  place
db.insert

Pers.First idx.insert drop
Pers.Last  idx.insert drop
```

It works the same for "DB.DELETE" and "DB.UPDATE": *first* you commit changes to the database buffer, *then* you update all indexes. This is very important. If you don't, some system variables may not be updated properly and you may corrupt your indexes. "IDX.DELETE" does not return a flag. The flag "IDX.INSERT" and "IDX.UPDATE" return indicates that the index now contains duplicate keys. If you're not interested, you can discard it just like we did.

You can also use a dictionary extension. It begins by including a different file:

```
include lib/dbms.4th
```

The second step is to bind the indexes after we've issued the usual red tape. Note the table to bind has to be current:

```
s" Persons.dbm" db.create
(Person) /Person s" Persons.dbm" db.declare to Person
Person db.use

8192 {cell} db.key Firstname idx.init
abort" Cannot create 'First' index" to Pers.First
8192 {char} db.key Lastname idx.init
abort" Cannot create 'Last' index" to Pers.Last

Pers.First Pers.Last 2 dbs.bind
abort" Cannot bind indexes"
```

After that you can insert, update and delete records using "DBS.INSERT", "DBS.UPDATE" and "DBS.DELETE" without worrying about updating the indexes. But it comes at a cost. If you're shutting down, you must unbind all the indexes. You can do that one by one:

```
db.clear
s" Hans"    db.buffer -> Firstname place
s" Bezemer" db.buffer -> Lastname  place
dbs.insert

Person dbs.unbind abort" Cannot unbind Person indexes"
```

And repeat that for all tables you've whose indexes you've bound. That's a lot of work, but this way you can mix bound and unbound tables in the same program. You can close them all with one single word, but in that case you have to bind *all* your tables to their indexes - even if they have *no* indexes at all, e.g.:

```
0 dbs.bind abort" Cannot bind table"
```

Now you can use "DBS.RELEASE" to release *all* indexes of *all* tables at once:

```
dbs.release abort" Cannot release indexes"
```

Finding a value in an index is a breeze as well. Again, you have to make sure that the corresponding table is used:

```
s" Hans" Pers.First idx.find 0= abort" Not found"
```

If "IDX.FIND" returns a true flag, your database buffer will contain the first record containing that value. You can use the "IDX.PREVIOUS" word to get any records that are smaller than that value or "IDX.NEXT" to get any records that equal or are greater than that value. If "IDX.FIND" returns false, your value was not found and the contents of the database buffer are undetermined.

If you're not sure about the first value, you can use "IDX.SEARCH". E.g. if you want the first surname beginning with "H", you can issue:

```
s" H" Pers.Last idx.search 0> abort" End of index"
```

If ”IDX.SEARCH” returns a positive value, it is simply not found. If it returns a zero value, the exact value was found in the database and you can use the associated buffer. If it returns a negative value, a value *bigger* than requested was put in the database buffer. If a buffer is valid, you can navigate from there using ”IDX.NEXT” or ”IDX.PREVIOUS”.

Finally, you can search any field using ”DBS.FIND” - whether it is indexed or not. It takes the same parameters as ”DB.FIND” and returns a non-zero value when the record was found. It also returns the index it used or a negative value (”N/A”) when none was available:

```
( find a record by key, abort if not found)
db.key Firstname s" Hans" dbs.find 0= abort" Not found"

( if an index was used, get previous record)
dup N/A = if drop else idx.previous then
```

Of course, it is a shame when you have to rebuild all your indexes when you restart a program. That’s why you can save and reload them. Saving them just takes a single line:

```
Pers.First s" persfrst.idx" idx.save
```

No, it won’t return a flag, but you can examine ”IDX.ERROR” if you want to. Loading takes little code as well:

```
{char} s" persfrst.idx" idx.load abort" Can’t load index"
to Pers.First
```

It returns the same stuff as ”IDX.CREATE” and ”IDX.INIT”, you’re already familiar with that. You don’t have to supply any more information than the type, because the rest of the required information was saved with the index itself. Finally, you will have to close the index. I suggest you close it when you’re closing the table or database itself:

```
dbs.close abort" Cannot release indexes"
Pers.First idx.close abort" Cannot close index"
Pers.Last  idx.close abort" Cannot close index"
```

If it returns a true flag, it failed - which means your dynamic memory is corrupted somehow. Finally, do not forget you can retrieve lots of information on the current status of an index by using the following words:

`idx.error`	Returns the last error encountered;
`idx.size`	Returns the total capacity of an index;
`idx.used`	Returns the number of entries used.

That’s all folks!

13.34 Speech synthesis

4tH can talk! All you need is the ”Festival” speech synthesis package[30] and a small 4tH interface. If you want to imitate old Arnold, this will do:

[30] Homepage: *http://www.cstr.ed.ac.uk/projects/festival/.*

```
include lib/say.4th
s" I'll be back!" say abort" Festival not available"
```

Well, that's cool, isn't it? You can also use the "eSpeak" package[31], which offers even more possiblities. E.g. this will be spoken in a Scottish female voice:

```
include lib/speak.4th
s" I'll be back!" s" en-sc+f1" speak abort" eSpeak not available"
```

Not only English is available, other languages are supported as well. An added advantage is that the package is also available for Windows.

13.35 GUI applications

You can create GUIs with 4tH. Not because 4tH has extensive language bindings, but because it has an interface with GTK-server[32]. After you've successfully installed GTK-server and made sure it's in your path, simply write[33]:

```
include lib/gtkserv.4th
gtk-srv-start
```

GTK-server is now initialized and aways your commands. There are two simple words to do that, "GTK{" and "}GTK". Everything between it is sent to GTK-server. Creating a window is as simple as:

```
gtk{ ." gtk_window_new 0" }gtk
```

Now this command returns a result, the number of the window. Now how can you get it? Very simple: any responses from GTK-server are stored in TIB, which you can process as usual:

```
0 parse number error? abort" Invalid response"
```

Ok, we got the window number now on the stack. Let's give the thing a title:

```
s" My title" gtk{ ." gtk_window_set_title " dup . -rot "type" }gtk
```

""TYPE"" is a word of the interface, which works just like 'TYPE', but places the string within double quotes. Note that after this the window number is still on the stack. You're probably gonna need it..

[31] Homepage: *http://espeak.sourceforge.net/*

[32] Homepage: *http://www.gtk-server.org/*

[33] The following interface only works with Unix-like platforms, except OS/X. This is due to limitations of GTK-server.

Since this is not a tutorial on GTK-server, we simply show you how to shut GTK-server down:

```
gtk-srv-stop
```

This implementation works with named pipes and is pretty fast. However, if it doesn't work for you, you may try an alternative interface which uses an unnamed pipe. Starting it is very similar:

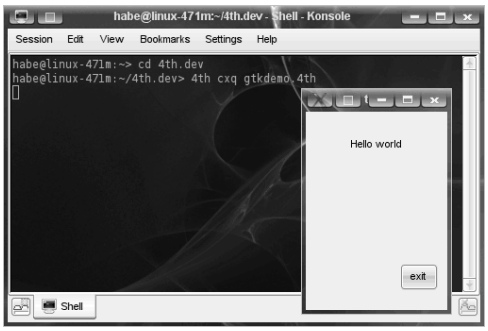

Figure 13.3: GTK demo

```
include lib/gtkipc.4th
gtk-srv-start
gtk{ s" gtk_window_new 0" s>msg }gtk
0 parse number error? abort" Invalid response"
```

The difference is that the GTK command
has to be assembled in memory in its entirety *before* it is sent to GTK-server. For that, three words have been defined: "S>MSG", "'S'>MSG" and "N>MSG". Each one adds a leading blank before the actual payload is appended, so you don't have to worry about that. The first one, "S>MSG" appends a string. Its little brother, "'S'>MSG" adds a quoted string, much like ""TYPE"". Finally, "N>MSG" adds an integer. And yes, any responses are stored in TIB.

So, now let's see how we can give this window a title:

```
s" My title" gtk{ s" gtk_window_set_title" s>msg dup n>msg -rot 's'>msg }gtk
```

Well, that isn't shocking, isn't it? Finally, when you're done shut GTK-server down[34]:

```
gtk-srv-stop
```

Note this interface is *slower*, uses *more* CPU and is slightly more difficult to handle than the default one, so if both interfaces work: use the default.

13.36 Card games

One of the least useful, but funniest libraries may be the playing card library, `cards.4th`, which is dedicated to making card games. It's very easy. After including the library, you take a new deck:

```
include lib/cards.4th
new-deck
```

Like any deck of cards, you have to shuffle it first. You need another library for that, called `shuffle.4th`. It contains two shuffle words, one for a range of cells, the other for a range of characters. The deck is made up of characters. You have to feed it the address of the character array and its length:

[34] This might trigger an error due to 4tHs implementation of `popen()`, but the interface handles it transparently for the programmer.

```
include lib/cards.4th
include lib/shuffle.4th
new-deck
deck /deck cshuffle
```

Ok, now we got a deck of shuffled cards. Now, let's deal:

```
deal
```

This will return an integer, representing the card. If you take modula 13, you will have the value of the card, numbered from 0 to 12 or *Ace* to *King*. If you divide it by 4 you will have the suit of the card, numbered from 0 to 3 or *Diamonds*, *Hearts*, *Clubs* and *Spades:*

```
dup 13 mod . cr
dup 4 / . cr
```

If you want a string representation, simply use "CARD". This will return a string (address/count) containing the card. Simply type it:

```
card type cr
```

Of coure, a deck of cards is limited. If you want to know if it's time for a new deck of cards, call "CARDS-LEFT". This will tell you how many cards are left in the current deck:

```
cards-left . cr
```

Note it's *your* responsibility to track how many cards are left in the deck. If you don't the program will abort with an error! And what when you run out of cards? Well, you simply take a new deck and start all over again. It's simple! This tiny program implements a blackjack game:

```
include lib/shuffle.4th
include lib/cards.4th
include lib/yesorno.4th

: score 13 mod dup if 1+ 10 min else 11 + then ;
: next-card deal dup card space type cr score + ;
: player ." Player: " cr 0 begin next-card s" Hit" yes/no? 0= until cr ;
: dealer ." Dealer: " cr 0 begin next-card dup 16 > until cr ;
: won? dup 21 > if drop ." is bust!" else ." has " 0 .r ." ." then cr ;
: shuffle-deck new-deck deck /deck cshuffle ;
: minijack shuffle-deck player dealer ." Dealer " won? ." Player " won? ;

minijack
```

The beauty is that it is *very* easy to read and understand. All the technical stuff is handled by the `cards.4th` library. So next time you want to make a nice cards game, don't forget about this one!

Chapter 14

Preprocessor libraries

14.1 Introduction

Some people start to frown when you mention preprocessors. But most don't even realize that when you compile C, you're *always* using a preprocessor - and with good reason! Without a preprocessor you really wouldn't want to write C, it would be awkward. C++ started its life as a preprocessor[1]. Several Forth compilers won't even compile without a preprocessor. Face it: there is nothing wrong with preprocessors!

Preprocessors are wonderful tools to hide awful constructs under a silky smooth syntax, enhance your productivity, simplify maintenance and prevent programming errors. 4tH comes with a preprocessor[2] that is more powerful than C's and shares many characteristics with M4[3], like:

- Recursive macro expansion;
- Text replacement;
- Parameter substitution;
- File inclusion;
- Conditional evaluation;
- Arithmetic expressions.

In order to further enhance it, 4tH provides preprocessor libraries, which are in fact predefined macro's. You use a preprocessor library like you would use an ordinary one, e.g.:

```
include 4pp/lib/standard.4pp
```

After you've run the preprocessor, you compile as usual. Note there is no protection against including a preprocessor library twice, but on the other hand it won't be included by (or include) other libraries, so that is pretty academic. If you're running a Unix-like Operating System, you could use `make` to make your life a little easier, e.g. by defining the following rules:

[1] http://www2.research.att.com/~bs/hopl2.pdf
[2] See chapter 18 for more information.
[3] http://en.wikipedia.org/wiki/M4_%28computer_language%29

```
%.c : %.4th
        4th cgq $< $@
%.c : %.hx
        4th lgq $< $@
%.4th : %.4pp
        pp4th -o $@ $<
```

Refer to your system documentation for more information. Finally (and this is very important), *always include preprocessor libraries after including all other libraries*. If you don't, you risk that your macro's are expanded while you're loading 4tH code that wasn't designed for it - which may have adverse effects. You don't want to chase bugs that were never there, don't you?

14.2 Stack instructions

What sets the 4tH preprocessor apart from all other preprocessors is its built-in string stack. And of course, with a stack comes stack instructions. Some, like `@drop`, `@dup`, `@nip` and `@over` are built-in, but some like `@swap` and `@rot` are missing. That is exactly what this library file provides. First of all, you have to include it:

```
include 4pp/lib/standard.4pp
```

Don't forget to use the `.4pp` extension or you'll get an error message, since the preprocessor won't be able to find the file. After that you'll have a full range of stack instructions at your disposal to manipulate the string stack. It allows you to do things like this:

```
include 4pp/lib/standard.4pp
:macro _fac >4> >3> @minus @add @sign @if @drop
       >3> @mul >3> >>> 1 @add <3< _fac ;
:macro __fac >>> 1 @swap >>> 1 @add >>> 2 @max
       <4< >>> 2 <3< _fac ;
:macro factorial >#> __fac @drop <1< #1# ;
." 10! = " factorial 10 . cr
```

Which is nothing less than calculating the factorial of 10 at *compile time*! Now let's break this thing down, how does it work?

- `factorial` puts "10" on the stack and calls `__fac`. After that, it takes the number from the string stack and prints it;
- `__fac` puts "1" on the stack, swaps both values and puts "1" on the stack once more. Then it adds TOS and 2OS[4], puts "3" on the stack and takes the largest value (like 'MAX'). TOS is stored in register ***4*** and the value "2" is stored in register ***3***. Then `_fac` is called.
- `_fac` puts the contents of register ***4*** and ***3*** on the stack, negates TOS and adds TOS and 2OS. If the result of this subtraction is not zero, the contents of register 3 are thrown on the stack once more and multiplied. Finally, the contents of register ***3*** are incremented and a recursive call to `_fac` is made.

[4] "TOS" means: "top of stack" and 2OS: "second of stack". It is a common way to refer to the top two values.

In short, the text `factorial 10` is completely consumed and replaced by the result of the calculation. That's cool, isn't it?

You might ask what's wrong with registers ***1*** and ***2***. Well, nothing.. But most preprocessor libraries use *at least* one register themselves - which is register ***1*** by convention. For the `@rot` macro this library even uses register ***1*** and ***2***. That means if you have stored a value there yourself and you use one of these predefined macro's, it will get clobbered - which is very nasty. In this example, we use `@swap`, which uses just register ***1***, so register ***2***, ***3*** and ***4*** are free to use.

So, save yourself hours of fruitless debugging and check whether a particular macro in a library uses a particular register before using it yourself. Finally, note the preprocessor is powerful enough to calculate the factorial recursively using *only* the string stack. But that's quite complex, you can make your life much easier using `@swap`, `@rot` and registers.

14.3 Coroutines

Calling a subroutine in any computer language is much like launching a program on a single-tasking operating system: you launch it, it takes control and you won't get it back until it relinquishes it. That's quite different from launching a program on a multitasking operating system. You take the control back, give it back to the running program and so on.

Wouldn't it be nice if 4tH could do that too? Well, I may seem like magic, but it can. Take a look at this tiny program:

```
include 4pp/lib/yield.4pp
: fib 0 1 yield begin dup rot + yield again ;
: .fib fib begin dup 100 < while dup . yield repeat drop drop grab ;
: .fibs 0 ?do .fib cr loop ;
5 .fibs
```

So, what happens here? ".FIB" simply prints all Fibonacci numbers smaller than 100. ".FIBS" does that trick a number of times (5 in this case). Finally, "FIB" calculates the Fibonacci numbers. That's all.

It all starts simple enough: ".FIB" calls "FIB", which sets up the Fibonacci sequence. And then it does something peculiar: it *yields*. Yielding is much like calling 'EXIT', but with one small difference: it isn't permanent. As a matter of fact, it returns to the calling word (the "caller"), but it remembers where it was when it gave back the control, much like if it were *calling* the word.

Ok, we're now back at ".FIB" and enter the loop with 'BEGIN', with 0 and 1 at the top of the stack. Sure, 1 is smaller than 100, so we pass the test and print "1". Now we encounter another "YIELD", which *yields* the control *back* to "FIB". We enter the loop there and calculate the next Fibonacci number - and yield again, back to ".FIB".

".FIB" is at 'REPEAT' now and jumps back to 'BEGIN', so we test and print the number before yielding control again. And so on, until we reach the limit and jump to "GRAB". We will see why we need "GRAB" when we examine what's going on.

This is heavy duty stuff. C can't do it without using every dirty trick in the book. We can do it in two lines of preprocessor code, so what's the secret. It's simple, just take a look at section 10.14 and the definition of "YIELD":

```
:macro yield r> execute ;
```

Whenever you yield, you take the return address of the ”caller” from the return stack - just like you would with 'EXIT'. 'EXECUTE' jumps to it, but *leaves* its own return address on the return stack - like a call would. In fact, the ”caller” and the ”callee” *swap places*, or - to put it in proper perspective - we started out with ”.FIB” calling ”FIB” and now we end up with ”FIB” calling ”.FIB”.

Now, let's say ”.FIB” encounters 'EXIT' (which was compiled by ';'). What do you think is going to happen? It returns to its caller. Which is now ”FIB”.

That's not what we want. First of all, ”FIB” never ends, so we can wait 'till kingdom come until it relinquishes control. Second, we want to return to the caller of ”.FIB”, which is ”.FIBS”. Note its return address was never changed - it's just buried beneath the return address of ”FIB”. So we get rid of it and *grab* control. No surprise that this is the definition of ”GRAB”:

```
:macro grab r> drop ;
```

If we had chosen to let ”.FIB” finish ”normally” and relinquish control through 'EXIT' instead of ”YIELD”, we wouldn't have needed ”GRAB”, of course. And the same goes for ”FIB”, if it had returned to ”.FIB” by 'EXIT'.

Note this is *extremely* powerful, but also extremely difficult to do right. Whenever control is yielded, the stack diagram has to be exactly what the other coroutine expects. Furthermore, you have to know at any given time what is calling what. For two words, that is feasible - but three or more becomes increasingly difficult. It's easy to do it wrong, but hard to do it right.

Technically speaking, ”YIELD” does not implement coroutines, but generators. The difference between the two is that true coroutines can choose who they yield to, while generators give you no choice - they're always bound to one particular word. Finally, Coroutines are *also* supported by an ”ordinary” 4tH library, but that implementation is slower and the optimizer *may* corrupt the return stack if you don't know what you're doing.

14.4 Interpretation

By defining just a few macro's the preprocessor is capable of interpreting and executing a sequence of *user defined* macro's at runtime. Let's say you want to define a constant and do some calculations e.g.:

```
include 4pp/lib/interprt.4pp

:macro /line >>> 64 ;
:macro #line >>> 16 ;
:macro @push >#> ;
:macro @* @mul ;
:macro @print @eval ;

[ /line @print ] constant bl_chars
[ #line @print ] constant bl_lines
[ #line /line @* @push 8 @* @print ] buffer: blocks
```

Bottom line: everything between the brackets is interpreted. Of course, there are other (much better) ways to achieve this, but you catch my drift. After you've run the preprocessor you will find it has expanded the source to:

```
64 constant bl_chars
16 constant bl_lines
8192 buffer: blocks
```

Note you can use built-in macro's or phony variables as long as you have wrapped them into a macro of your own. This library only uses register *1*, leaving you plenty of registers to play with.

14.5 Closures

Yes, you're reading it correctly: use can use the preprocessor to turn an archaic, low level language like Forth into something very hip and modern. Closures are functions that carry their own, persistent variables along. Some call them "a poor mans objects". Defining a closure is very simple. It is almost like defining a structure:

```
include 4pp/lib/closures.4pp
                                    \ counter creation word
:: counter                          ( n1 n2 addr --)
   vars                             \ define closure data
     var: start                     \ initial value
     var: inc                       \ increment per call
   end-vars {                       \ only cells may be used!
     this -> inc ! this -> start !  \ initialise closure
     {: this -> inc @ this -> start dup ? +! ;}
   }                                \ assign default behavior
;
```

You can't use "colon" to define a closure, you have to use the "double colon" notation. Then you start defining the variables of the closure right away. *Note you can only define cell variables*. The rest is quite straight forward: instead of `struct`, you use `vars`. Instead of using `end-struct`, you use `end-vars` to finish your data definition.

Then we start with the initalization routine. Note the *first* parameter of *any* closure routine is *always* its address (just like 'DOES>'), which is put on the stack by "{" and accessible through "THIS". Any additional parameters follow after that. In this case, when creating the closure the initial value of the counter and its increment are set.

Finally, we define the run time behavior - that is the code which is enclosed by "{:" and ";}". Done. Now let's define a few closures of the "COUNTER" type:

```
5 2 sets counter counter1           \ make a counter
```

That's all? Yes, that's all you need to do to create and initialize "COUNTER1". Now let's call it:

```
counter1                            \ 5
counter1                            \ 7
counter1                            \ 9
```

First it will print "5", then "7" and finally "9". Now let's try another "COUNTER:

```
13 3 sets counter counter2          \ make another counter
counter2                            \ 13
counter2                            \ 16
counter2                            \ 19
```

Now that wasn't too hard, was it? Ok, before you start asking stupid questions, consider this: wouldn't you be better off with full fledged object orientation? If you answer this question with "yes", read on. You're in for a surprise.

14.6 Object orientation

We already told you that C++ started its life as a mere preprocessor. In 4tH, all that is required to achieve almost full object orientation is a few humble preprocessor macro's. If you don't like object orientation, that's fine. You take the blue pill, the story ends, you wake up in your bed and believe whatever you want to believe. But if you take the red pill, you stay in Wonderland, and I show you how deep the rabbit hole really goes.

Object orientation comes with an abstract terminology that is incomprehensible to mere humans, but scratch away the shiny surface and you'll understand what it's all about. Stay with me.

14.6.1 Encapsulation

Object orientation defines encapsulation as "a language construct that facilitates the bundling of data with the methods operating on that data". So what are we actually talking about. Let's start with a structure that holds the data. Now add a few fields that hold execution tokens and you're done. That's your "class" definition.

Building an object is nothing more than allocating storage space for your "class". You can reserve that space in static or dynamic memory. That's the easy part. The problem is, the fields for our execution tokens are still empty and when you "instantiate" an object, you want these execution tokens to be initialized. The dirty trick is, you have to do that secretly - in true object oriented tradition.

In 4tH you define a class with a "double colon" definition. A class consists of three parts, the data definition, the "binding" of the "methods" and information hiding, e.g.

```
:: Account
   class                                \ data definition
     ffield:  accountBalance
     virtual: CheckBalance
   end-class {                          \ data initialization

     500 s>f this -> accountBalance f!
     :virtual CheckBalance { this -> accountBalance f@ } ;method
   }                                    \ method binding
   private{ accountBalance }            \ information hiding
;
```

You can only use cells in the data definition. Sorry, you have to store your strings somewhere else. A data definition is almost identical to defining a structure, the only thing is you have to do it within the "double colon" definition. Object orientation also requires a thing called "open recursion", which is nothing more than a "this" or "self" variable, which just holds the address of the "object". That is where the curly braces come in.

The curly braces puts the address (that is always on the top of stack when an object is invoked) on the return stack. Yes, you can still use the return stack, just think of the "THIS" variable between two curly braces as the 'I' in a 'DO..LOOP' and you'll be fine.

The body of a "double colon" definition is actually executed when an object is created, so the floating point "property" (we call it a field) "accountBalance" is set to 500. The colon definition that starts with ":VIRTUAL" is just a ':NONAME' definition, which leaves an execution token. ;METHOD" on its turn pokes this execution token into the equivalent "VIRTUAL:" field of the "class". That's it. We're done, now let's use it.

```
fvariable myBalance                     \ define an FP variable
instance Account myAccount              \ create an Account object
```

The ”INSTANCE” word creates an object on the heap. Well, not just that; it also secretly calls ”Account”, so the object is properly initialized. It also passes the address of the newly created object to ”Account”, so it knows *what* to initialize. The curly braces in ”Account” make sure that ”THIS” works.

```
myAccount => CheckBalance myBalance f! \ now pass a message
myBalance f@ f. cr                     \ and store the result
```

You pass a message to an object by using the ”=>” operator. You can also use an ”adjective”[5]. E.g. you can write the same piece of code like this if you want:

```
myAccount -> virtual CheckBalance myBalance f!
myBalance f@ f. cr
```

If you use ’->’ on its own you’ll just end up with the address of the methods field. When ”a message is passed”, a ”method is invoked”. ”Passing a message” means calling the object by name and passing all parameters. The object retrieves the *reference* of the actual method and executes it: that’s the ”invocation of a method”. In 4tH, ”methods” can be ”virtual”, which means you can store another execution token in that field at any time.

When a virtual ”method” is ”invoked”, 4tH secretly passes the address of the object, as a matter of fact right on top of the stack. That’s why the method itself has curly braces as well - to store the address. Now it starts making sense:

- The opening curly brace puts the address on the return stack;
- ”THIS” holds that address, so now ”accountBalance” points to the right place;
- The closing curly brace clears the return stack;
- The data stack was uncluttered the whole time.

I told you folks, it’s all smoke and mirrors. 4tH also supports static ”methods”, which means you can’t change them later on and they’re automatically passed to all subclasses. We could rewrite our previous example like this:

```
:: Account
   class                                \ data definition
     ffield: accountBalance
     method: CheckBalance
   end-class {                          \ data initialization

     500 s>f this -> accountBalance f!
     :method CheckBalance { this -> accountBalance f@ } ;method
   }                                    \ method binding
   private{ accountBalance }            \ information hiding
;
```

You cannot invoke static methods with the ”=>” operator, you’ll have to use ”->” instead. Why? Well, a static ”method” is just an ordinary definition, so when you call it, the address of the object is consumed immediately. It is not expanded to an address to the ”method” field itself. Worse, static ”methods” do not even need a METHOD:” declaration, since they’re never changed and consequently don’t require any space. It’s just a little syntactic sugar that does absolutely *nothing*.

[5] Yes, I know: normal people would call it a *cast*, but allow me to introduce another, completely superfluous concept to the already incomprehesible object orientation vocabulary.

14.6.2 Subtype polymorphism

”Polymorphism” is the ability of ”objects” belonging to different types to respond to ”method” or ”property” calls of the same name, each one according to an appropriate type-specific behavior. Ok, that’s a hard one to swallow. Let’s break it down. We now know that is ”class” is nothing more than a structure.

In section 11.15 we already learned we could *extend* a structure. You may not realize it, but you can pass a pointer to an *extended* structure to a word that takes a pointer to the *unextended* structure and manipulate *any* of the fields they share - without problems. Of course, that includes fields that hold execution tokens. *That* is the basic principle of polymorphism.

Let’s investigate that one a but further. Lots of animals create sounds. Let’s make a ”class” for them:

```
:: Animal
   class
     virtual: Talk
   end-class {}
;
```

That’s all. A ”class” with one ”method”, no ”properties” and no ”binding”. You’d probably call it a ”class with uninitialized methods”, but in object orientation it is usually called an ”interface”. Even if you don’t initialize or bind anything you *have to* use curly braces in order to keep the stack balanced. You can also use ”{}”, which is shorthand for ”{ }”.

Ok, now we have an animal, let’s define a dog:

```
:: Dog
   extends Animal
   end-extends {
     :virtual Talk { ." Woof" cr } ;method
   }
;
```

The ”EXTENDS” word allows you to indicate from which class this type is derived - in this case an ”Animal”. In this case, we don’t add any ”properties” or ”methods”, but we do bind the previously defined method ”Talk”. We can do the same thing for a cat:

```
:: Cat
   extends Animal
   end-extends {
     :virtual Talk { ." Meow" cr } ;method
   }
;
```

If we now create a ”cat” object and a ”dog” object we can make them talk in their own perculiar way:

```
static Cat MyCat
static Dog MyDog

MyCat => Talk
MyDog => Talk
```

That’s what’s called ”dynamic dispatch”: when a method is invoked on an object, the object *itself* determines what code gets executed. You can take that a step further by defining e.g. a figure:

```
:: figure                               \ define an empty class figure
   class                                \ with no properties and two
     virtual: surface                   \ uninitialized methods
     virtual: outline
   end-class {}
;
```

Yes, that's an "interface". Now let's define a rectangle:

```
:: rectangle                            \ define a subtype rectangle
   extends figure                       \ with two specific properties
     field: _width                      \ and a private method
     field: _height
     method: double
   end-extends {
                                        \ now initialize surface and outline
     :method  double { 2* } ;method
     :virtual surface { this -> _width @ this -> _height @ * } ;method
     :virtual outline {
       this -> _width  @ this -> double
       this -> _height @ this -> double +
     } ;method

     private{ double }                  \ make method private
   }
;
```

Note the method "double" is private, which is means it can only be used *within* the class itself. That's why we made it a "static" method, because it's not gonna be redefined any time soon. And let's also define a circle while we're at it:

```
:: circle                               \ define a subtype circle
   extends figure                       \ with one specific property
     field: radius                      \ and a private method
     method: pi*
   end-extends {
                                        \ now initialize surface and outline
     :method  pi* { 103993 33102 */ } ;method
     :virtual surface { this -> radius @ dup * this -> pi* } ;method
     :virtual outline { this -> radius @ 2*    this -> pi* } ;method

     private{ pi* }                     \ make method private
   }
;
```

Both subclasses have very different properties, e.g. "radius" has very little meaning in the context of a rectangle. However, they do share some methods, so if we instantiate them, we can "ask" them to calculate their surface - or their outline:

```
static rectangle MyRectangle            \ make a rectangle instance
static circle    MyCircle               \ make a circle instance

4 MyRectangle -> _width !               \ initialize the rectangle
5 MyRectangle -> _height !
MyRectangle => surface . cr             \ use both methods
MyRectangle => outline . cr

25 MyCircle -> radius !                 \ initialize the circle
MyCircle => surface . cr                \ use both methods
MyCircle => outline . cr
```

Nothing to it - if you know what's under the hood.

14.6.3 Inheritance

The final feature true object orientation requires is inheritance. Inheritance is a way to reuse code, which means that classes can inherit attributes and behavior from pre-existing classes called ”superclasses”. So, how is this done in 4tH? Let’s define a traffic light, a simple one:

```
:: two-light                            \ create a two light traffic light
   class
     field:   Red                       \ red color string
     field:   Green                     \ green color string
     field:   State                     \ state of the traffic light
     field:   #lights                   \ number of lights
     field:   Description               \ description of lights
     virtual: Switch                    \ change to the next state
     virtual: Destroy                   \ destroy the object
     method:  Show                      \ show the current state
   end-class {
```

Ok, this is our traffic light, the one you find on pedestrian crossings. It’s got a red light and a green light - which totals two lights. It’s got a state so we can see what light is on at any time and a few methods. ”Switch” makes it change from one state to another, ”Show” shows the traffic light and ”Destroy” frees the memory the object occupies.

By the way, ”Destroy” may walk as a ”destructor” and talk as a ”destructor”, but it’s not. A destructor is a method that is *automatically* called when an object is ”destroyed”. In other words: when the method ”Destroy” is invoked, it kills the object. With a ”real” destructor, an object is killed and the method is invoked. Not quite the same thing.

Now let’s initialize our object:

```
s" Red"   this -> Red   place!          \ save red light color
s" Green" this -> Green place!          \ save green light color
                                        \ we need a table of descriptions
here ['] Green , ['] Red , this -> Description !
2 this -> #lights !                     \ it has two lights
0 this -> State !                       \ set initial state
```

”PLACE!” is a helper word that allocates a bit of dynamic memory for the string and pokes the address in property ”Red”. The same for ”Green”, of course. We already concluded that it has two lights, so no surprises there and we set the initial state. ”Description” is a pointer to an unnamed table which we use in the next method:

```
 :method Show {                         \ show the current color
    1+ cells this -> Description @ + @c cells this + @ count type cr
 } ;method                              \ note we fix the description offset
```

On a ”method” level it just takes an integer, representing the ”State” of our traffic light. Next we fetch the item from the table, which is the offset to the appropriate ”property” of the ”object”. So we add it and fetch its contents, which is the address to a string in dynamic memory, so we can type it. Job done. The rest is straight forward:

```
:virtual Switch {
   this -> State dup @ dup              \ get the current state
   this -> Show 1+                      \ show it, go to next state
   this -> #lights @ mod swap !         \ and set it
} ;method                               \ assign it to the Switch method

:default Destroy {                      \ free the green and red light
   this -> Green free? this -> Red free?
   this free abort" Cannot free object"
} ;method
```

The method ”Switch” not only reads property ”State”, it also updates it. In short: it completely encapsulates it. Nothing else, but this class alone, has any business with ”State”, so we can make it private (which we will do later on). Note we use a ":DEFAULT" definition. The difference with a ":VIRTUAL" definition is that we can never refer to the "original" method again once we reassign a virtual method. With ":DEFAULT" you can[6], although you can only do it *once*: multiple defaults for the same virtual method are not allowed.

As a matter of fact, ”:DEFAULT” secretly defines a *static* method and assigns its execution token to the *virtual* method. So, the ”original” static method still has its own symbol in the symbol table, which you can refer to later. The ”Destroy” method we defined simply frees the memory taken by the ”lights” and finally its own. You can do that, because the code that frees the object has no dependencies with the object itself. The variable ”THIS” in the context of ”FREE” only represents the address of the object itself, nothing more.

```
private{ State }
```

We already told you ”State” would become a ”private method”. Now let’s make a heavier duty traffic light:

```
:: three-light                          \ create a three light traffic light
   extends two-light                    \ based on the two light traffic light
     field: Yellow                      \ add the color Yellow
   end-extends {
```

We only need to add the yellow light, don’t we? Sure, we need to add a string for the yellow light:

```
s" Yellow" this -> Yellow place!
```

The configuration may be a bit different, so we have to make ”Description” point to another unnamed table:

```
here ['] Green , ['] Yellow , ['] Red , this -> Description !
```

And of course, we have three lights now:

```
3 this -> #lights !
```

But the rest is just about the same, is it? Not quite. If we destroy our traffic light, we don’t want to leave a yellow light on the sidewalk:

```
:virtual Destroy {                      \ free all lights
   this -> Yellow free?                 \ destroy the yellow color
   this <- Destroy                      \ destroy all previous colors
} ;method                               \ using the static method
```

Yes, we destroy our yellow light first, then we can destroy the rest of the traffic light (including the traffic light itself) by calling its original (default) method. Note the use of ”<-”, which is for ”default” methods what ”=>” is for ”virtual” methods. What we’ve done for ”Destroy” is what is called ”overriding a method”, which means changing its behavior. By the way, a ”default” method has its own ”adjective”, so this is allowed as well:

[6]The reason is that 4tH doesn’t have a virtual method table. See: http://en.wikipedia.org/wiki/Virtual_method_table

```
:virtual Destroy {                      \ free all lights
   this -> Yellow free?                 \ destroy the yellow color
   this -> default Destroy              \ destroy all previous colors
} ;method
```

A two-light traffic light won't behave the same as a three-light traffic light. But we only need to change the bits that are different. The rest is *inherited*. If it's a three-light traffic light, it will behave accordingly. You can see that when you actually use the classes:

```
instance two-light DontWalk             \ define a pedestrian light

." A pedestrian traffic light:" cr cr

DontWalk => Switch                      \ go to the next state four times
DontWalk => Switch
DontWalk => Switch
DontWalk => Switch
DontWalk => Destroy cr                  \ now destroy it

instance three-light TrafficLight       \ define a normal light

." A normal traffic light:" cr cr

TrafficLight => Switch                  \ go to the next state four times
TrafficLight => Switch
TrafficLight => Switch
TrafficLight => Switch
TrafficLight => Destroy cr              \ now destroy it
```

Now you think: "how does he do that?" Well, frankly: by cheating! If you expand the source, you will see what is actually going on:

```
['] two-light ['] three-light (~~parent!)
```

It passes two execution tokens to a secret helper word that invokes the "superclass" initialization before invoking the initialization of the "subclass". If that superclass is a subclass itself, it will call that superclass initialization and so on. When a superclass has finished it relinquishes control to the subclass, so it can do its thing. Finally, the original subclass itself gets the chance to override any "properties" or "methods" it sees fit - and then we're done. Hurray, we got an object - initialized and all.

It's like you make a painting and paint over it a few times. What you see is the final picture. You're unaware of what is hidden under the various layers of paint - but that doesn't mean it's not there or it was never painted. It's that simple.

14.6.4 Determining the type and size of an object

You can determine the type of an object by simply fetching it, e.g.:

```
MyCircle type@
```

You can compare it to a predetermined class by using the "TYPEOF" word:

```
typeof circle
```

You can also determine the superclass of an object:

```
MyCircle parent@
```

And its counterpart on the class side is used like this:

```
parentof circle
```

You can compare these values with the '=' word, nothing special. If you want to go *really* deep and decide whether some interface was incorporated somewhere down the line, you can do that too:

```
MyCircle type@ YourCircle type@ derived?
```

This will leave a 'TRUE' flag if "MyCircle" was derived from the class of the object "YourCircle". Usually you may want to check against a predefined class, like this:

```
MyCircle type@ kindof circle
```

The value all these words return is actually an execution token of the class, so you can use it when you e.g. want to clone objects. It is secretly stored in a hidden field of the class. Of course, what's missing there is the *size* of a class. You can determine that one with the "SIZEOF" word, e.g.

```
sizeof circle
```

If you've allocated objects on the heap, getting their sizes isn't rocket science either:

```
MyCircle allocated
```

For static objects there is no way to determine their sizes (in true 4tH fashion), but that will not be a problem, because you won't be allocating them in a userdefined word.

14.7 This is the end

This is the end of it. If you mastered all we have written about 4tH, you may be just as proficient as we are. Or even better. In the meanwhile you may even have acquired a taste for this strange, but elegant language. If you do, it may be time to step up to Forth, since 4tH does have it limitations. This is in no way an obligation. If you feel comfortable with 4tH, please do stick with it!

If you need any help, you can contact us by sending an email to:

```
The.Beez.speaks@gmail.com
```

Note that we do appreciate *any* input, so if you've written a state of the art application in 4tH, used 4tH in some special way or do have any comments or suggestions on 4tH, we'd like to hear from you! We do also have a web-site:

```
http://thebeez.home.xs4all.nl/4tH/
```

You will find there lots of documentation and news on 4tH. We'd like to thank you for putting so much effort in 4tH. We tried to be of assistance and we hope we did it well!