And so Forth...

Copyright J.L. Bezemer

2001-04-25

Table of Contents

1 Preface

1.1 Copyright

1.2 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 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 ’hansoft@bigfoot.com’. You will not only get an answer, but you will help future Forth 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.

If this isn’t enough to teach you Forth you can always get a real good textbook on Forth, like "Starting Forth" by Leo Brodie. Have fun!

1.3 About this primer

This primer was originally written for 4tH, my own Forth compiler. 4tH isn’t ANS-Forth compliant, by ANS-Forth standards not even a ANS-Forth system. After a while I got questions why certain examples weren’t working. Since I tested every single one of them I wondered why. Until it dawned on me: people learning Forth were using my primer!

So due to high demand I started to rewrite it for ANS-Forth compliant systems. Most of these systems don’t even have a manual *at all* so the need for it should be great. The next question was: which format. Since I wanted to learn LyX anyway, I settled for that. You can produce various formats with it which are readable on most systems, including MS-DOS, MS-Windows and Linux.

The next question was: how far do you go. The original version was heavily geared towards 4tH, which reflects my own views on Forth. And those views sometimes contradict to those of ANS-Forth. However, since Leo Brodie took the liberty in ”Thinking Forth” to express his views, I thought I should have the freedom to express mine.

Some examples, especially in the ”Advanced topics” chapter, use special 4tH extensions. Fortunately Wil Baden had helped me to write a 4tH-to-ANS-Forth interface. Since some of these extensions cover functionalities commonly found in other languages I decided to keep those sections in, using the Easy4tH definitions. In the previous chapters you’ll find some 4tH words as well, but very sparingly.

You may find that some examples are not working with your specific Forth compiler. That may have several reasons. First, your compiler may not support all ANS-Forth wordsets. Second, your compiler may not be completely ANS-Forth compliant. I’ve tested most of these examples with GForth or Win32Forth, which are (almost) 100% ANS-Forth compliant. Third, your compiler might be case-sensitive.

The ANS-Forth standard is a very important document. I can only advise you to get it. You should have no trouble finding it on the internet. I can only hope that the compiler you chose *at least* documented its ANS-Forth compatibility.

This primer was written in the hope that it will be useful and that starting Forthers aren’t put off by the high price of Forth textbooks. It is dedicated to Leo Brodie, who taught me much more than just Forth.

Hans Bezemer
Den Haag, 2001-03-07

2 Forth fundamentals

2.1 Making calculations without parenthesis

To use Forth 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 Forth 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 Forth 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.) Forth 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.) Forth then pushes the result of the addition back onto the stack in place of the two numbers. Thus the stack contains:

```
2 7
```

When Forth 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)
```

2.2 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
```

Forth 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 Forth will answer:

```
2 3
```

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

```
2 . . cr
```

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

```
2 dup . . cr
```

Forth 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’!)
```

2.3 Deep stack manipulators

There are two manipulators that can dig deeper into the stack, called ’PICK’ and ’ROLL’ but I cannot recommend them. 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.

2.4 Pass arguments to functions

There is no easier way to pass arguments to functions as in Forth. Functions have another name in Forth. 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 Forth it has powerful capabilities. In other languages, you can get back only one result. In Forth you can get back several!

All words in Forth 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?

2.5 Making your own words

Of course, every serious language has to have a capability to extend it. So has Forth. 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>). It depends on the Forth you’re using whether it is case- sensitive or not, but usually it isn’t.

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 Forth!

2.6 Adding comment

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

The first form you’ve already encountered. 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 *.
```

Forth 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 Forth 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!

2.7 Text-format of Forth source

Forth source can be simple ASCII-files. 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 Forth 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, Forth won't care and your harddisk supplier either.

However, some Forths still use a special line editor, which works with screens. Screens are usually 1K blocks, divided into 16 lines of 64 characters. Explaining how these kind of editors work goes beyond the scope of this manual. You have to check the documentation of your Forth compiler on that. The files these editors produce are called blockfiles.

2.8 Displaying string constants

Displaying a string is as easy as adding 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 Forth:

```
.( Hello world!)
```

Enter 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?

2.9 Declaring 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 or you'll get pretty strange results. A variable is a word too! And whitespace characters are not allowed. Note that Forth is usually not case-sensitive!

2.10 Using variables

Of course variables are of little use when you can't 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 ?
```

2.11 Built-in variables

Forth has two built-in variables you can use for your own purposes. They are called 'BASE' and '>IN'. 'BASE' controls the radix at run-time, '>IN' is used by 'WORD' and 'PARSE'.

2.12 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 Forth can handle. We'll come to that further on.

2.13 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 shouldn't use a name that already has been taken. A constant is a word too! And whitespace characters are not allowed. Note that Forth is usually not case-sensitive.

2.14 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:

1. BL
2. FALSE
3. PAD
4. TIB
5. TRUE

2.15 Using booleans

Booleans are expressions or values that are either true or false. They are used to conditionally execute parts of your program. In Forth 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 one 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.

2.16 IF-ELSE constructs

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

```
variable var
4 var !

: test
      var @ 5 >
      if ." Greater" cr
      else ." Less or equal" cr
      then
;

test
```

So now our program does the job. It 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'..

2.17 FOR-NEXT constructs

Forth does also have FOR-NEXT constructs. The number of iterations is known in this construct. E.g. let's print the numbers from 1 to 10:

```
: test
      11 1 do i . cr loop
;

test
```

The first number presents the limit. When the limit is goes beyond the limit minus one the loop terminates. The second number presents the initial value of the index. That's where it starts of. 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:

```
: test
      0 0 ?do i . cr loop
;

test
```

'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:

```
: test
      11 1 do i . cr 2 +loop
;

test
```

This one will produce exactly what you asked for. An increment by two. This one will produce all negative numbers from -1 to -11:

```
: test
      -11 -1 do i . cr -1 +loop
;

test
```

Why -11? Because the loop terminates when it reached the limit minus one. And when you're counting downward, that is -11. You can change the step if you want to, e.g.:

```
: test
      32767 1 do i . i +loop
;

test
```

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':

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

test
```

2.18 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 Forth 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 has 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.

2.19 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".

2.20 Infinite loops

In order to make an infinite loop one could write:

```
: test
      begin ." Diamonds are forever" cr 0 until
;

test
```

But there is a nicer way to do just that:

```
: test
      begin ." Diamonds are forever" cr again
;
```

```
test
```

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

2.21 Getting a number from the keyboard

Let's start with "you're not supposed to understand this". If you dig deeper in Forth you'll find out why it works the way it works. But if you define this word in your program it will read a number from the keyboard and put it on the stack. If you haven't entered a valid number, it will prompt you again.

```
S" MAX-N" ENVIRONMENT?          \ query environment
[IF]                            \ if successful
NEGATE 1- CONSTANT (ERROR)      \ create constant (ERROR)
[THEN]

: number            ( a – n)
  0. Rot dup 1+ c@ [char] - = >r count r@ if 1 /string then >number nip
  0= if d>s r> if negate then else r> drop 2drop (error) then ;

:    input#                             ( – n)
     begin
          refill drop bl word number     ( n)
          dup (error) <>                 ( n f)
          dup 0=                         ( n f -f)
          if swap drop then              ( f | n f)
     until ;
```

2.22 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.

3 Arrays and strings

3.1 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:

```
create sixteen 16 cells allot
```

That's it, we're done!

3.2 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 cells + !
```

Which will store '5' in the 0th cell. So storing '7' in the 8th cell

is done like this:

```
7 sixteen 8 cells + !
```

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

```
sixteen 0 cells + @
sixteen 4 cells + @
```

Plain and easy.

3.3 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 'CREATE. Then you specify all its elements. 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.

3.4 Using arrays of constants

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

```
sixteen 0 cells + @
```

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

```
sizes 0 cells + @
```

So I don't think you'll have any problems here.

3.5 Creating strings

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

```
create name 10 chars allot
```

Note that the string variable includes the count byte. That is a special character that tells Forth how long a string is. You usually don't have to add that yourself because Forth 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 count byte. These kind of strings are usually referred to as counted 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:

```
create hello 7 chars allot
```

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:

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

In the next section we will tell you how to get "Hello!" into the string.

3.6 Initializing strings

You can initialize a string with the 'S"' word. You haven't seen this one yet, but we will discuss it in more depth later on. If you want the string to contain your first name use this construction:

```
: place over over >r >r char+ swap chars cmove r> r> c! ;

create name 16 chars allot
s" Hello! " name place
```

The word "PLACE", which is a common word[A], copies the contents of a string constant 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 constant to a string that is too short will result in an error or even worse, corrupt other strings.

[A] Although not part of the ANS-Forth standard.

3.7 Getting the length of a string

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

```
: place over over >r >r char+ swap chars cmove r> r> c! ;

create greeting 32 chars allot   \ 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
```

You usually have nothing to do with the string address. However, it may be required by other words like we will see in the following section. If you just want the bare length of the string you can always define a word like 'length$':

```
: place over over >r >r char+ swap chars cmove r> r> c! ;
: length$ count swap drop ;

create greeting 32 cells allot   \ define string greeting
s" Hello!" greeting place        \ set string to 'Hello!'
greeting length$                 \ get string length
.( String length: ) . cr         \ print the length
```

3.8 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 the string address and the number of characters it has to print. Yes, that are the values that are left on the stack by 'COUNT'! So printing a string means issuing both 'COUNT' and 'TYPE':

```
: place over over >r >r char+ swap chars cmove r> r> c! ;

create greeting 32 cells allot   \ 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$':

```
: place over over >r >r char+ swap chars cmove r> r> c! ;
: print$ count type ;

create greeting 32 cells allot   \ define string greeting
s" Hello!" greeting place        \ set string to 'Hello!'
greeting print$ cr               \ print the string
```

3.9 Copying a string variable

You might want to copy one string variable to another. There is a special word for that, named 'CMOVE'. It takes the two strings and copies a given number of characters from the source to the destination. Let's take a look at this example:

```
: place over over >r >r char+ swap chars cmove r> r> c! ;

create one 16 chars allot        \ define the first string
create two 16 chars allot        \ define the second string

s" Greetings!" one place         \ initialize string one
one dup                          \ save the real address
count                            \ get the length of string one
1+                               \ account for the count byte
swap drop                        \ get the real address
two swap                         \ get the order right
cmove                            \ copy the string
two count type  cr               \ print string two
```

The most difficult part to understand is probably why and how to set up the data for 'CMOVE'. Well, 'CMOVE' wants to see these values on the stack:

```
source destination #chars
```

With the expression:

```
one count
```

We get these data on the stack:

```
source+1 length
```

But the count byte hasn't been accounted for so far. That's why we add:

```
1+
```

So now this parameter has the right value. Now we have to restore the true address of the string and tell 'CMOVE' where to copy the contents of string one to. Initially, we got the correct address. That is why we saved it using:

```
dup
```

Now we're getting rid of the "corrupted" address by issuing:

```
swap drop
```

This is what we got right now:

```
source #chars
```

If we simply add:

```
two
```

The data is still not presented in the right order:

```
source #chars destination
```

So we add the extra 'SWAP' in order to get it right. Of course you may define a word that takes care of all that:

```
: place over over >r >r char+ swap chars cmove r> r> c! ;
: copy$ swap dup count 1+ swap drop rot swap cmove ;

create one 32 chars allot
create two 32 chars allot
s" Greetings!" one place
one two copy$
```

You may wonder why we keep on defining words to make your life easier. Why didn't we simply define these words in the compiler instead of using these hard to understand words? Sure, but I didn't write the standard. However, most Forths allow you to permanently store these words in their vocabulary. Check your documentation for details.

3.10 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:

```
: place over over >r >r char+ swap chars cmove r> r> c! ;
: nextchar dup dup c@ 1- swap char+ c! char+ ;

create one 32 chars allot        \ define string one
s" Hans Bezemer" one place       \ initialize string one
one dup count type cr            \ duplicate and print it
nextchar                         \ move one character forward
dup count type cr                \ duplicate and print it again
nextchar                         \ move one character forward
dup count type cr                \ duplicate and print it again
nextchar                         \ move one character forward
count 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 CHAR+ is usually equivalent to 1+, but Forth was defined to run on unusual hardware too - the CPU of a pocket calculator could be a nibble-machine (4-bit) so each CHAR occupies in fact two addresses. And of course, some Forth systems may treat CHAR to be a 16-bit unicode. If we want to discard the first name at all we could even write:

```
: place over over >r >r char+ swap chars cmove r> r> c! ;

create one 32 chars allot        \ define string one
s" Hans Bezemer" one place       \ initialize string one
one dup c@ 5 -                   \ copy address and get count
swap 5 chars + dup rot swap c!   \ save new count
count 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 no special word for slicing strings. There is a smarter way to handle strings in Forth, which we will discuss later on. But if you desperately need slicing you might want to use a word like this. It works just like 'CMOVE' with an extra parameter:

```
: slice$
      swap                       \ reverse dest and #chars
      over over                  \ copy the dest and #chars
      >r >r >r >r                \ store on the return stack
      +                          \ make address to the source
      r> r>                      \ restore dest and #chars
      char+                      \ make address to destination
      swap cmove                 \ copy the string
      r> r>                      \ restore dest and #chars
      c!                         \ save
;
```

This is another example of "you're not supposed to understand this". You call it with:

```
source index-to-source destination #chars
```

The index-to-source starts counting at one. So this will copy the first name to string "two" and print it:

```
: place over over >r >r char+ swap chars cmove r> r> c! ;

: slice$
      swap                       \ reverse dest and #chars
      over over                  \ copy the dest and #chars
      >r >r >r >r                \ store on the return stack
      +                          \ make address to the source
      r> r>                      \ restore dest and #chars
      char+                      \ make address to destination
      swap cmove                 \ copy the string
      r> r>                      \ restore dest and #chars
      c!                         \ save
;

create one 32 chars allot        \ declare string one
create two 32 chars allot        \ declare string two
s" Hans Bezemer" one place       \ initialize string one
one 1 two 4 slice$               \ slice the first name
two count type cr                \ print string two
```

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

```
: place over over >r >r char+ swap chars cmove r> r> c! ;

: slice$
      swap                       \ reverse dest and #chars
      over over                  \ copy the dest and #chars
      >r >r >r >r                \ store on the return stack
      +                          \ make address to the source
      r> r>                      \ restore dest and #chars
      char+                      \ make address to destination
      swap cmove                 \ copy the string
      r> r>                      \ restore dest and #chars
      c!                         \ save
;
```

```
create one 32 chars allot       \ declare string one
create two 32 chars allot       \ declare string two
s" Hans Bezemer" one place      \ initialize string one
one 6 two 7 slice$              \ slice the first name
two count type cr               \ print string two
```

Since the last name is seven characters long and starts at position six (start counting with one!). Although this is very "Basic" way to slice strings, we can do this kind of string processing the Forth way. It will probably require less stack manipulations.

3.11 Appending strings

There is no standard word in Forth to concatenate strings. As a matter of fact, string manipulation is one of Forths weakest points. But since we are focused here on doing things, we will present you a word which will get the work done.

The word 'APPEND' appends two strings. In this example string "one" holds the first name and "Bezemer" is appended to string "one" to form the full name. Finally string "one" is printed.

```
: place over over >r >r char+ swap chars cmove r> r> c! ;

: append                        ( a1 n2 a2 --)
      over over                 \ duplicate target and count
      >r >r                     \ save them on the return stack
      count chars +             \ calculate offset target
      swap chars move           \ now move the source string
      r> r>                     \ get target and count
      dup >r                    \ duplicate target and save one
      c@ +                      \ calculate new count
      r> c!                     \ get address and store
;

create one 32 chars allot       \ define string one
s" Hans " one place             \ initialize first string
s" Bezemer" one append          \ append 'Bezemer' to string
one count type cr               \ print first string
```

Of course, you can also fetch the string to be appended from a string variable by using 'COUNT'.

3.12 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. But here is a word that can compare two strings.

```
: place over over >r >r char+ swap chars cmove r> r> c! ;

create one 32 chars allot       \ define string one
create two 32 chars allot       \ define string two

: test
  s" H. Bezemer" one place      \ initialize string one
  s" R. 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
;

test
```

Simply pass two strings 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.

3.13 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 Forth 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:

```
: place over over >r >r char+ swap chars cmove r> r> c! ;

create one 32 chars allot       \ 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. And what about leading spaces? Patience, old chap. You've got a lot of ground to cover.

3.14 String constants and string variables

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

There are only two different actions you can do with a string constant. First, you can define one using 's"'. Second, you can print one using '."'.

There are two different ways to represent a string variable in Forth. First, by using just its address, the so-called counted string. Forth relies on the count byte to find the end of the string. Second, by using its address *and* its length. This requires two values.

The word 'TYPE' requires the latter form. Therefore, you have to convert a counted string in order to print it. You can convert an counted string to an "address-count string" with the word 'COUNT'. If you moved a string (by using 'CMOVE') without taking the count byte into account you have to set it yourself.

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

3.15 The count byte

The count byte is used to set the length of a counted string. It has nothing to do with British royalty! It is simply the very first byte of a string, containing the length of the actual string following it.

3.16 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!

3.17 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. Forth 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. Note that 'CHAR' only works with printable characters (ASCII 33 to 127 decimal).

3.18 When to use [CHAR] or CHAR

There is not one, but *two* words for getting the ASCII code of a character, '[CHAR]' and 'CHAR'. Why is that? Well, the complete story is somewhat complex, but one is for use inside colon definitions and one is for use outside colon definitions. And 'CHAR' isn't the only word which is affected. We've put it all together in a neat table for you:

Inside a definition	Outside a definition
."	.(
[CHAR]	CHAR
[']	'

For example, this produces the same results:

```
: Hello ." Hello world" [char] ! emit cr ; Hello
.( Hello world!) char ! emit cr
```

You should also have noticed in the meanwhile that you can't use control structures like DO..LOOP or IF..THEN outside colon definitions. And not only these, others like 'C"" can't be used as well. Real Forth-ers call this "inside a colon definition" thing *compilation mode* and working from the prompt *interpretation mode*. You can do really neat things with it, but that is still beyond you now.

3.19 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, Forth has a more elegant solution for that:

```
10 spaces
```

Which will output ten spaces. Need I say more?

3.20 Fetching individual characters

Take a look at this small program:

```
: place over over >r >r char+ swap chars cmove r> r> c! ;

create one 32 chars allot       \ 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 Forth, 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 Forth always start with zero instead of one, but that is the count byte. So accessing the first character might be done with:

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

This is the complete program:

```
: place over over >r >r char+ swap chars cmove r> r> c! ;

create one 32 chars allot       \ define string one
s" Hans" one place              \ initialize string one
one 2 chars + c@                \ get the second character
emit cr                         \ print it
```

3.21 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:

```
: place over over >r >r char+ swap chars cmove r> r> c! ;

create one 32 chars allot       \ 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:

```
: place over over >r >r char+ swap chars cmove r> r> c! ;

create one 32 chars allot       \ define string one
```

```
s" Hans" one place              \ initialize string one
one 4 chars + c@                \ get the fourth character
```

Remember, we start counting from one! 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!':

```
: place over over >r >r char+ swap chars cmove r> r> c! ;

create one 32 chars allot       \ define string one
s" Hans" one place              \ initialize string one
one 4 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':

```
: place over over >r >r char+ swap chars cmove r> r> c! ;

create one 32 chars allot       \ define string one
char d                          \ we want to store 'd'
s" Hans" one place              \ initialize string one
one 4 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:

```
: place over over >r >r char+ swap chars cmove r> r> c! ;

create one 32 chars allot       ( --)
s" Hans" one place              ( --)
one                             ( a)
4 chars                         ( a i)
+                               ( a+i)
char d                          ( a+i c)
swap                            ( c a+i)
c!                              ( --)
```

Now the second, optimized version:

```
: place over over >r >r char+ swap chars cmove r> r> c! ;

create one 32 chars allot       ( --)
char d                          ( c)
s" Hans" one place              ( c)
one                             ( c a)
4 chars                         ( c a i)
+                               ( c a+i)
c!                              ( --)
```

3.22 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.

Forth 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 256 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?

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. But there is a catch. This program can get you into trouble:

```
64 constant #name               \ length of string
create name #name chars allot   \ define string 'name'

name #name accept               \ input string
name 1+ swap type cr            \ swap count and print
```

Since 64 characters *plus* the count byte add up to 65 characters. You will probably want to use this definition instead:

```
: saccept 1- swap 1+ swap accept ; \ define safe 'ACCEPT'

64 constant #name                 \ length of string
create name #name chars allot     \ define string 'name'

name #name saccept                \ input string
name 1+ swap type cr              \ print string
```

This "safe" version decrements the count so the user input will fit nicely into the string variable. In order to terminate it you write:

```
: saccept 1- swap 1+ swap accept ; \ define safe 'ACCEPT'

64 constant #name                 \ length of string
create name #name chars allot     \ define string 'name'

name dup #name saccept            \ input string
swap c!                           \ set count byte
```

The word 'ACCEPT' always returns the number of characters it received. 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.

3.23 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.

3.24 What is the PAD?

The PAD is short for "scratch-pad". It is a temporary storage area for strings. It is heavily used by Forth 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.

3.25 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 generate any output or do any parsing.

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:

```
: place over over >r >r char+ swap chars cmove r> r> c! ;

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

3.26 Temporary string constants

Hey, haven't we already seen this? Yes, you have.

```
s" This is a string" type cr
```

No 'COUNT'? No. 'S"' leaves its address and its length on the stack, so we can call 'TYPE' right away. Note that this string doesn't last forever. If you wait too long it will be overwritten. It depends on your system how long the string will last.

3.27 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 'WORD', which is a parser. '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 copies the sliced string to PAD and returns its address. 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:

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

```
test
```

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:

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

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

test
```

If '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.

3.28 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 Forth 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.

This implementation 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 'NUMBER' can't convert a string it returns that constant. Forth has its own conversion word called '>NUMBER', but that is a lot harder to use. Let's take a look at this program:

```
S" MAX-N" ENVIRONMENT?          \ query environment
[IF]                            \ if successful
NEGATE 1- CONSTANT (ERROR)      \ create constant (ERROR)
[THEN]

: number 0. Rot dup 1+ c@ [char] - = >r count r@ if 1 /string
  then >number nip 0= if d>s r> if negate then else r> drop
  2drop (error) then ;

: test
      ." Enter a number: "            \ write prompt
      refill drop                     \ enter string
      bl word                         \ parse string
      number dup                      \ 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
;

test
```

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

3.29 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. Forth can do that too. Let's take the previous program and alter it a bit:

```
S" MAX-N" ENVIRONMENT?          \ query environment
[IF]                            \ if successful
NEGATE 1- CONSTANT (ERROR)      \ create constant (ERROR)
[THEN]

: number 0. Rot dup 1+ c@ [char] - = >r count r@ if 1 /string
  then >number nip 0= if d>s r> if negate then else r> drop
  2drop (error) then ;

: test
      ." Enter a number: "            \ write prompt
      refill drop                     \ enter string
      bl word                         \ parse string
      number dup                      \ 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
;

test
```

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

Forth 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:

```
S" MAX-N" ENVIRONMENT?          \ query environment
[IF]                            \ if successful
NEGATE 1- CONSTANT (ERROR)      \ create constant (ERROR)
[THEN]

: number 0. Rot dup 1+ c@ [char] - = >r count r@ if 1 /string
  then >number nip 0= if d>s r> if negate then else r> drop
  2drop (error) then ;

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

: test
  ." Enter a number: "           \ write prompt
  refill drop                    \ enter string
  bl word                        \ parse string
  number dup                     \ 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
;

test
```

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:

```
S" MAX-N" ENVIRONMENT?          \ query environment
[IF]                            \ if successful
NEGATE 1- CONSTANT (ERROR)      \ create constant (ERROR)
[THEN]

: number 0. Rot dup 1+ c@ [char] - = >r count r@ if 1 /string
  then >number nip 0= if d>s r> if negate then else r> drop
  2drop (error) then ;

: test
  ." Enter a number: "             \ write prompt
  refill drop                      \ enter string
  bl word                          \ parse string
  hex                              \ convert hexadecimal
  number dup                       \ 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
;

test
```

'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:

```
S" MAX-N" ENVIRONMENT?          \ query environment
[IF]                            \ if successful
NEGATE 1- CONSTANT (ERROR)      \ create constant (ERROR)
[THEN]

: number 0. Rot dup 1+ c@ [char] - = >r count r@ if 1 /string
  then >number nip 0= if d>s r> if negate then else r> drop
```

```
  2drop (error) then ;

: binary 2 base ! ;

: test
  ." Enter a number: "            \ write prompt
  refill drop                     \ enter string
  bl word                         \ parse string
  binary                          \ convert hexadecimal
  number dup                      \ 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
;

test
```

'BASE' is a predefined variable that enables you to select any radix between 2 and 36. This makes Forth very flexible:

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

However, this won't work:

```
: wont-work hex 02B decimal . cr ;
```

But this will:

```
hex
: will-work 02B decimal . cr ;
```

Why that? Well, 'HEX' will just be compiled, not executed. So when Forth tries to compile "02B", it doesn't recognize it as a hexadecimal number and will try to find word '02B'. Which it can't of course. Note that after "WILL-WORK" has been compiled all numbers following it will still be compiled as hexadecimal numbers. Why? Because 'DECIMAL' is compiled too! You should place a 'DECIMAL' outside the definition in order to reset the radix. BTW, it is always a good idea to add a leading zero to a hexadecimal number. For example, is this a hex number or a word:

```
face
```

3.30 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 Forth 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>d <# #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'. You can forget about the 'S>D' word. Just don't forget to put it there.

This is a slightly different format:

```
: print3# s>d <# # # # #> 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 'S>D', '<#' and '#>' and add 'TYPE CR':

```
s>d <# # # #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 >r abs s>d <# #s r> 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 on the stack twice. First on the returnstack 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 >r abs s>d <# r> sign #s #> type ;
```

But that is still not enough to write "$2000.15" 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# s>d <# :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.

3.31 Converting a number to a string

Since there is no special word in Forth 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 move the string immediately after its creation. So we'll create a word that not only creates the string, but moves it directly to its proper location:

```
: >string >r dup >r abs s>d <# #s r> sign #>
  r@ char+ swap dup >r cmove r> r> c! ;
  ( n a – )
```

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

```
create num$ 16 chars allot
-1024 num$ >string
num$ count type cr
```

4 Stacks and colon definitions

4.1 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. 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 Forth to throw the address of "ADD" on the stack instead of executing "ADD". What you can actually do with it, we will show you in the next section.

4.2 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 Forth 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. We’ll give you a little hint:

```
create subs ’ hello , ’ goodbye ,
```

Does this give you any ideas?

4.3 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    ( Initial value, can’t change)
2 b + value c   ( Initial value, can change)
```

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.

In fact, a value is a variable that behaves in certain aspects like a constant. Why use a value at all? Well, there are situations where a value can help, e.g. when a constant CAN change during execution. It is certainly not a good idea to replace all variables by values.

4.4 The stacks

Forth has 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 definition on the other stack, which is called the Return Stack for obvious reasons.

Then execution continues at the colon-definition. Every colon-defini tion 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 (Forth 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.

4.5 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 Forth there are special words to manipulate stack items in pairs, e.g. "2DUP" (n1 n2 — n1 n2 n1 n2) or "2DROP" (n1 n2 --). In most Forths they are already available, but 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 Forth attempts to return to the wrong address. Avoid this, since this can be very hard bugs to fix.

4.6 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:

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

test
```

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:

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

test
```

You might have noticed (unless you're blind) 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.

4.7 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 a Forth-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.

4.8 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. However, it gives you a pretty good idea how the Return Stack is used by the system.

4.9 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:

```
: 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 Forth 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. Forth shuts down the compiler when encountering ’;’. This is not performed by ’EXIT’.

4.10 How deep is your stack?

You can ask Forth 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            ( Forth reports "3")
```

5 Advanced topics

5.1 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":

```
: test
      1 2 and
      if
            ." I’m here"
      then
;

test
```

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 !              \ 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 !              \ 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 !              \ 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, although the standard allows it. 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 istrue (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:

```
: test
      1 0<>
      2 0<>
      and if
            ." I’m here" cr
      then
;

test
```

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

5.2 Including your own definitions

At a certain point you may have written a lot of definitions you're very fond of. You use them in most of your programs, so before you actually get to the programs you have to work your way through all these standard definitions. Even worse, when you change one of them you have to edit all your programs. Most Forths have a way to permanently include them in the kernel, but if you're not up to that or want your programs to be as portable as possible you can solve this in a better way.

Just put all of your definitions in a single file and start your program with:

```
s" mydefs.fs" included
```

The compiler will now first compile all the definitions in "mydefs.fs" before starting with the main program. We've done exactly the same in the following sections. Most of the code you'll find there uses the Easy4tH extensions, so instead of listing them every single time, we've just included them. Easy4tH has old favorites like "PLACE" and "NUMBER" already available to you.

You have to define the constant "/STRING-SPACE" first in order to use it. A value of 16384 should be fine in most cases. If you get an error, you can always increase it.

5.3 Conditional compilation

This is something which can be very handy when you're designing a Forth program for different environments or even different Forth compilers. Let's say you've written a general ledger program in Forth 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]
65536 constant #Entries
[then]

variable CurrentEntry

create Entries #Entries cells allot
```

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 "[then]". So, in this case the compiler behaves like you've written:

```
256 constant #Entries
variable CurrentEntry
create Entries #Entries cells allot
```

Would you change "DEMO" to false, the compiler would behave as if you wrote:

```
variable CurrentEntry
65536 constant #Entries
create Entries #Entries cells allot
```

The word '[IF]' only works at compile time and is *never* compiled into the object. '[IF]' takes a expression. If this expression is true, the code from '[IF]' until '[ELSE]' is compiled, just as '[IF]' wasn't there. Is this expression is false, everything '[IF]' up to '[ELSE]' 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 this one is nicer and will take up less code:

```
DEMO [IF]
: .Message ." Buy the full version" ;
[ELSE]
: .Message ." No more entries" ;
[THEN]

: CheckIfFull                   ( n – n)
      dup #Entries =            ( n f)
      if                        ( n)
           drop                 ( --)
           .Message
```

```
            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] .. [THEN]' can be nested! Conditional compilation is very powerful and one of the easiest features a language can have.

5.4 Exceptions

You know when you violate the integrity of Forth, 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 Forth, but it takes code and performance, which makes your program less compact and slower.

Well, you can do that too in Forth. And not even that, you can trigger your own errors as well. This simple program triggers an error and exits Forth when you enter a "0":

```
16384 constant /string-space
s" easy4th.fs" included

: input#                            \ get a number
      begin
            refill drop             ( --)
            bl word number          ( n )
            dup (error) <>          ( n f )
            dup 0=                  ( n f -f )
            if swap drop then       ( f | n f )
      until                         ( input routine )
;
                                    \ get a number
                                    \ if non-zero, return it
                                    \ if zero, throw exception
: could-fail                        ( – n)
      input# 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 "COULD-FAIL". "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 Forth won’t exit, but transfers control back to some routine. Let’s do just that:

```
16384 constant /string-space
s" easy4th.fs" included

: input#
      begin
           refill drop               ( --)
           bl word number            ( n )
           dup (error) <>            ( n f )
           dup 0=                    ( n f -f )
           if swap drop then         ( f | n f )
      until                          ( input routine )
;

: could-fail                         ( – n)
      input# dup 0=
      if 1 throw then
;

: do-it                              ( --)
      drop drop could-fail
;

: 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 Forth 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:

```
16384 constant /string-space
s" easy4th.fs" included

: input#
      begin
           refill drop               ( --)
           bl word number            ( n )
           dup (error) <>            ( n f )
           dup 0=                    ( n f -f )
           if swap drop then         ( f | n f )
      until                          ( input routine )
;

: could-fail                         ( – n)
      input# 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)
      input# 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)
      input# 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 Forth do the checking.

We won't even have to set flags in every possible colon-definition, since Forth 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?

5.5 Lookup tables

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:

```
16384 constant /string-space
s" easy4th.fs" included

create MonthTable
      $"  January " ,
      $" February " ,
      $"   March  " ,
      $"   April  " ,
      $"    May   " ,
      $"   June   " ,
      $"   July   " ,
      $"  August  " ,
      $" September" ,
      $"  October " ,
      $" November " ,
      $" December " ,

: Get-Month               ( n -- )
      12 min 1- MonthTable swap cells + @ pad copy [B]  count type cr
;
```

[B]

"COPY" is part of the Easy4tH extensions and will copy

Which does the very same thing and will certainly work faster. Normally, you can't do that this easily in ANS-Forth, but with this primer you can, so use it! 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:

```
: Search-Table            ( n1 a1 n2 n3 – n4 f)
      swap >r             ( n1 a1 n3)
      rot rot             ( n3 n1 a1)
      over over           ( n3 n1 a1 n1 a1)
      0                   ( n3 n1 a1 n1 a1 n2)

      begin               ( n3 n1 a1 n1 a1 n2)
           swap over      ( n3 n1 a1 n1 n2 a1 n2)
           cells +        ( n3 n1 a1 n1 n2 a2)
           @ dup          ( n3 n1 a1 n1 n2 n3 n3)
           0> >r          ( n3 n1 a1 n1 n2 n3)
           rot <>         ( n3 n1 a1 n2 f)
           r@ and         ( n3 n1 a1 n2 f)
      while               ( n3 n1 a1 n2)
           r> drop        ( n3 n1 a1 n2)
           r@ +           ( n3 n1 a1 n2+2)
           >r over over   ( n3 n1 a1 n1 a1)
           r>             ( n3 n1 a1 n1 a1 n2+2)
      repeat              ( n3 n1 a1 n2)

      r@ if
           >r rot r>      ( n1 a1 n3 n2)
           + cells + @    ( n1 n4)
           swap drop      ( n3)
      else
           drop drop drop ( n1)
      then

      r>                  ( n f)
      r> drop             ( n f)
;
```

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 one is the number of fields this table has. And on top of the stack you'll find the field which value it has to return. The first field must be the "index" field. It contains the values which have to be compared. That field has number zero.

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. You can change all that by modifying the line with "0> >r". It returns the value in the appropriate field 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:

```
16384 constant /string-space
s" easy4th.fs" included

0 Constant NULL

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 ,
```

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 MonthTable 2 1 Search-Table ;     ( n1 – n2 f)
```

Now, we define a new "Get-Month" routine:

```
: Get-Month                         ( n --)
      Search-Month                  \ search table

      if                            \ if month is found
           pad copy 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:

```
16384 constant /string-space
s" easy4th.fs" included

0 Constant NULL
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 MonthTable #MonthFields 1 Search-Table ;
```

This enables us to add more fields without ever having to modify "Search-Month" 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 MonthTable #Monthfields 2 Search-Table ;
```

Of course, there is room for even more optimization, but for now we leave it at that. Do you now understand why Forth shouldn't have a CASE construct?

5.6 What DOES> CREATE do?

Let's take a closer look at 'CREATE'. What does 'CREATE' actually do? Well, it takes the string afterward as a name and makes a word out of it. Let's try this:

```
CREATE aname
```

You can even type that at the prompt. It works, it just makes a word. If you don't believe me, type this:

```
aname .
```

Now "ANAME" just wrote out an address. When you're still at the prompt and haven't done anything else, type this:

```
5 ,
```

Right after our definition we just compiled in "5". Is that useful? Can we ever retrieve it? Sure, we just created "ANAME", didn't we? And the address "ANAME" gave is exactly the address where out "5" was compiled! Believe it or not:

```
aname @ .
```

Sure, it answers "5". We can even compile another number:

```
10 ,
```

And retrieve it:

```
aname 1 cells + @ .
```

Looks a lot like an array doesn't it? Well, under the hood Forth is actually doing the same thing. Let's say we want a word that compiles a number and makes a name for it. We could define this:

```
: compilenumber create , ;
```

Now let's use it:

```
10 compilenumber anothername
anothername @ .
```

First 'CREATE' does it's job and and creates the word "ANOTHERNAME". Second, ',' kicks in and compiles the number that is on the stack, just like at our previous example. When we execute, the address of the number is thrown on the stack, so we can retrieve the contents and display them.

Don't you think "COMPILENUMBER" is a bad name. What we actually did was create an initialized variable! So what do you think the word 'VARIABLE' does? Simple:

```
: variable create 1 cells allot ;
```

It creates a name and reserves some space for it! But can't we do anything else than just throw a address. Yes, you can not just define the way Forth compiles a word, but also what it does at runtime. You use 'DOES>' to define it. Let's say we want it to display the contents right away. We already got an address. What next? Sure:

```
@ .
```

That will get the contents of the address and show 'em. Just put them behind 'DOES>':

```
: displaynumber create , does> @ . ;
```

That's it! Let's use it:

```
11 displaynumber anumber
anumber
```

Great, isn't it? Looks a lot like 'CONSTANT', doesn't it? Let's fill you in on that. 'CONSTANT' is defined like this:

```
: constant create , does> @ ;
```

There is really nothing more to it. Yeah, you can do great things with that. Make you own datatypes and such. We'll see more of that later on.

5.7 Multidimensional arrays

We've seen we can define our own datatypes. We can now put it to use and make our lives easier. So far we've had to struggle our way through arrays, defining them like

```
create myarray 10 cells allot
```

Wouldn't it be nice if we could Forth do the work and just had to write:

```
10 array myarray
```

That definition shouldn't be hard to make. Just enter:

```
: array create cells allot ;
```

First we define an entry and then allocate space for it. However we still have to write:

```
10 myarray 5 cells + !
```

when we want to assign a value to the the sixth element (we start counting at zero, remember?). We can simplify that one too. When executed, "MYARRAY" throws it address on the stack. If we throw the index on the stack first, we could calculate the address:

```
   swap               ( addr index)
   cells +            ( addr-of-indexed-element)
```

Just put that behind 'DOES>' like you learned in the previous section. So this is our complete definition:

```
: array
  create              ( n)           \ create an entry
    cells allot       ( --)          \ allocate the number of cells
  does>               ( n a)         \ what to do at runtime
    swap              ( a n)         \ swap the count
    cells +           ( a+n)         \ calculate address
;
```

So now we can write things like this:

```
   5 myarray @ .
```

Which is equivalent to:

```
   myarray 5 cells + @ .
```

And displays the contents of the sixth element of "MYARRAY". You're now probably running to your computer and try to define something that does the same for multidimensional arrays. But you're going to find out pretty quickly that this approach won't work.

Why? Well, it's easy enough to allocate the space for a multidimensional array. Just multiply the number of rows and the number of columns:

```
: 2array create * cells allot ;
```

This allows you to define a twodimensional array like this:

```
16 8 2array myarray
```

However, you're missing something when you want to calculate the address of e.g. the second element in the third row. How many elements are there in a row? The only way to get that information is to store it inside the array itself, just before the allocated space:

```
: 2array      ( n1 n2)
  create      ( n1 n2)
  dup         ( n1 n2 n2)
  ,           ( n1 n2)
  *           ( n1*n2)
  cells allot ( --)
;
```

We just have to remember when we write our 'DOES>' part that the address on the stack points to the "number of elements in a row" we compiled, not to the actual data area. That comes behind that one. Let's first see what syntax we want:

```
1 2 myarray
```

That expression should leave the address of the third element of the second row on the stack. Row comes first, okay? We got these values on the stack:

```
row element address
```

Now, these are the actions we need to take:

- We have to get the "number of elements in a row" at "address".
- We have to multiply that with the row requested.
- We have to add the element requested to that one.
- We have to correct that offset for the "number of elements in a row".

And here is the definition that does it:

```
: 2array
  create              ( n1 n2)        \ create an entry
    dup               ( n1 n2 n2)     \ = rows columns columns
    ,                 ( n1 n2)        \ compile the number of cells in a row
    * cells           ( n1*n2)        \ calculate size
    allot             ( --)           \ allocate the number of cells
  does>               ( n1 n2 a)      \ what to do at runtime
    rot over @        ( n2 a n1 n3)   \ get number of cells in a row
    * rot + 1+        ( a n1*n3+n2+1) \ calculate offset
    cells +           ( a+n1*n3+n2+1) \ calculate address
;
```

5.8 Fixed point calculation

We already learned that if we can't calculate it out 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:

```
31415 CONSTANT PI
10000 CONSTANT 10K         ( scaling constant )
VARIABLE XS                ( square of scaled angle )

: KN  ( n1 n2 – n3, n3=10000-n1*x*x/n2 where x is the angle )
      XS @ SWAP /          ( x*x/n2 )
      NEGATE 10K */        ( -n1*x*x/n2 )
      10K +                ( 10000-n1*x*x/n2 )
;

: (SIN)                    ( x – sine*10K, x in radian*10K )
      DUP DUP 10K */       ( x*x scaled by 10K )
      XS !                 ( save it in XS )
      10K 72 KN            ( last term )
      42 KN 20 KN 6 KN     ( terms 3, 2, and 1 )
      10K */               ( times x )
;

: SIN                      ( degree – sine*10K )
      PI 180 */            ( convert to radian )
      (SIN)                ( compute sine )
;
```

If you enter:

```
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.

```
: /10K. <# # # # # [char] . hold #S #> type cr ;
45 sin /10K.
```

This one will actually print:

```
0.7071
```

But note that Forth internally still works with the scaled number, which is "7071". Another example:

```
: SQRT                     ( n1 – n2, n2**2<=n1 )
      0                    ( initial root )
      SWAP 0               ( set n1 as the limit )
      DO 1 + DUP           ( refresh root )
           2* 1 +          ( 2n+1 )
      +LOOP                ( add 2n+1 to sum, loop if )
;                          ( less than n1, else done )
```

```
: .fp <# # [char] . hold #S #> type cr ;
```

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 scale it by 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:

```
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.

5.9 Recursion

Yes, but can she do recursion? Of course she can! In order to let a colon-definition call itself, you have to use the word 'RECURSE'. Everybody knows how to calculate a factorial. In Forth you can do this by:

```
: factorial           ( n1 – n2)
      dup 2 >
      if
           dup 1-
           recurse *
      then
;

10 factorial . cr
```

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.

5.10 Forward declarations

It doesn't happen very often, but sometimes you have a program where two colon-definitions call each other. There is no special instruction in Forth to do this, like Pascals "FORWARD" keyword, but still it can be done. It even works the same way. Let's say we've got two colon-definitions called "STEP1" and "STEP2". "STEP1" calls "STEP2" and vice versa. First we create a value called "(STEP2)". We assign it the value '-1' since it is highly unlikely, there will ever be a word with that address:

```
-1 value (Step2)
```

Then we use vectored execution to create a forward declaration for "STEP2":

```
: Step2 (Step2) execute ;
```

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)", but this creates a superfluous word.

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 ; to (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 ;).

5.11 This is the end

This is the end of it. If you mastered all we have written about Forth, 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, there is plenty left for you to learn.

If you find any errors in this primer or just want to make a remark or suggestion, you can contact us by sending an email to:

hansoft@bigfoot.com

We do also have a web-site:

http://hansoft.come.to

You will find there lots of documentation and news on 4tH, our own Forth compiler.

Part I. Appendices

Bibliography

ANSI X3/X3J14 (1993).
Draft proposed American National Standrad for Information Systems — Programming Languages — Forth. Global Engineering Documents, 15 Inverness Way East, Englewood, CO 80122-5704, USA, sixth edition, 1993. Document Number: ANSI/IEEE X3.215-1994.
Leo Brodie (1982).
Starting Forth. Prentice Hall International, second edition, 1982.
Leo Brodie (1984).
Thinking Forth. Prentice Hall International, 1984.
Hans Bezemer / Benjamin Hoyt (1997).
Lookup Tables. Forth Dimensions, Volume XIX, Number 3, September 1997 October.

History

Version	Author	Date	Modification
0.1	Hans Bezemer	2001-03-07	Initial document
0.2	Hans Bezemer	2001-03-11	Used ’COMUS’ APPEND and changed ” to $” in section ’Lookup Tables’
0.3	Hans Bezemer	2001-03-25	Changed several things in Easy4tH and fixed some errors in example programs
0.4	Hans Bezemer	2001-04-06	Got rid of SCOPY and added ’What DOES> CREATE do’
0.5	Hans Bezemer	2001-04-25	Added first part of ’Multidimensional arrays’

Easy4tH

```
\ easy4th  V1.0d            A 4tH to ANS Forth interface

\ Typical usage:
\    4096 constant /string-space
\    s" easy4th.fs" included

\ This is an ANS Forth program requiring:
\      1. The word NIP in the Core Ext. word set
\      2. The word /STRING in the String word set
\      3. The word D>S in the Double word set
\      4. The words MS and TIME&DATE in the Facility Ext. word set
\      5. The words [IF] and [THEN] in the Tools Ext. word set.

\ (c) Copyright 1997,2001 Wil Baden, Hans Bezemer.  Permission is granted by the
\ authors to use this software for any application provided this
\ copyright notice is preserved.


\ Uncomment the next line if REFILL does not function properly
\ : refill query cr true ;

\ 4tH datatypes
: ARRAY CREATE CELLS ALLOT ;
: STRING CREATE CHARS ALLOT ;
```

```
: TABLE CREATE ;

\ 4tH constants
S" MAX-N" ENVIRONMENT?          \ query environment
[IF]                            \ if successful
NEGATE 1- CONSTANT (ERROR)      \ create constant (ERROR)
[ELSE]
.( Error: MAX-N undefined) cr
[THEN]

S" MAX-N" ENVIRONMENT?          \ query environment
[IF]                            \ if successful
CONSTANT MAX-N                  \ create constant MAX-N
[ELSE]
.( Error: MAX-N undefined) cr
[THEN]

S" STACK-CELLS" ENVIRONMENT?    \ query environment
[IF]                            \ if successful
CONSTANT STACK-CELLS            \ create constant STACK-CELLS
[ELSE]
.( Error: STACK-CELLS undefined) cr
[THEN]

S" /PAD" ENVIRONMENT?           \ query environment
[IF]                            \ if successful
CONSTANT /PAD                   \ create constant /PAD
[ELSE]
.( Error: /PAD undefined) cr
[THEN]

\ 4tH compiletime words
: [NOT] 0= ;
: [*] * ;
: [+] + ;

\ 4tH wordset
: TH CELLS + ;
: @’ @ ;
: COPY ( a b – b ) >R  DUP C@ 1+ R@ SWAP MOVE  R> ;
: WAIT 1000 * MS ;

: NUMBER             ( a – n)
  0. ROT DUP 1+ C@ [CHAR] - = >R COUNT
  R@ IF 1 /STRING THEN >NUMBER NIP 0=
  IF D>S R> IF NEGATE THEN ELSE R> DROP 2DROP (ERROR) THEN
;

( Reserve  STRING-SPACE  in data-space. )
CREATE STRING-SPACE           /STRING-SPACE CHARS ALLOT
VARIABLE NEXT-STRING          0 NEXT-STRING !

( caddr n addr – )
: PLACE OVER OVER >R >R CHAR+ SWAP CHARS MOVE R> R> C! ;

( "string<">" – caddr )
: $" [CHAR] " PARSE
  DUP 1+ NEXT-STRING @ + /STRING-SPACE >
        ABORT" String Space Exhausted. "
  STRING-SPACE NEXT-STRING @ CHARS + >R
  DUP 1+ NEXT-STRING +!
  R@ PLACE
  R>
;

\ 4tHs Random generator

( Default RNG from the C Standard.  ‘RAND’ has reasonable )
( properties, plus the advantage of being widely used. )
VARIABLE RANDSEED

32767 CONSTANT MAX-RAND

: RAND                                ( – random )
  RANDSEED @ ( random) 1103515245 *   12345 +   DUP RANDSEED !
  16 RSHIFT   MAX-RAND AND
;

: SRAND ( n – ) RANDSEED ! ;  1 SRAND

( Don’t mumble. )
: random        ( – n )        RAND ;

: set-random    ( n – )        SRAND ;

( Mix ’em up. )
```

```
: randomize                                ( – )
 TIME&DATE  12 * + 31 * + 24 * + 60 * + 60 * +  set-random
;

randomize
```

GNU Free Documentation License

GNU Free Documentation License

Version 1.1, March 2000

0. PREAMBLE

The purpose of this License is to make a manual, textbook, or other written document "free" in the sense of freedom: to assure everyone the effective freedom to copy and redistribute it, with or without modifying it, either commercially or noncommercially. Secondarily, this License preserves for the author and publisher a way to get credit for their work, while not being considered responsible for modifications made by others.

This License is a kind of "copyleft", which means that derivative works of the document must themselves be free in the same sense. It complements the GNU General Public License, which is a copyleft license designed for free software.

We have designed this License in order to use it for manuals for free software, because free software needs free documentation: a free program should come with manuals providing the same freedoms that the software does. But this License is not limited to software manuals; it can be used for any textual work, regardless of subject matter or whether it is published as a printed book. We recommend this License principally for works whose purpose is instruction or reference.

1. APPLICABILITY AND DEFINITIONS

This License applies to any manual or other work that contains a notice placed by the copyright holder saying it can be distributed under the terms of this License. The "Document", below, refers to any such manual or work. Any member of the public is a licensee, and is addressed as "you".

A "Modified Version" of the Document means any work containing the Document or a portion of it, either copied verbatim, or with modifications and/or translated into another language.

A "Secondary Section" is a named appendix or a front-matter section of the Document that deals exclusively with the relationship of the publishers or authors of the Document to the Document’s overall subject (or to related matters) and contains nothing that could fall directly within that overall subject. (For example, if the Document is in part a textbook of mathematics, a Secondary Section may not explain any mathematics.) The relationship could be a matter of historical connection with the subject or with related matters, or of legal, commercial, philosophical, ethical or political position regarding them.

The "Invariant Sections" are certain Secondary Sections whose titles are designated, as being those of Invariant Sections, in the notice that says that the Document is released under this License.

The "Cover Texts" are certain short passages of text that are listed, as Front-Cover Texts or Back-Cover Texts, in the notice that says that the Document is released under this License.

A "Transparent" copy of the Document means a machine-readable copy, represented in a format whose specification is available to the general public, whose contents can be viewed and edited directly and straightforwardly with generic text editors or (for images composed of pixels) generic paint programs or (for drawings) some widely available drawing editor, and that is suitable for input to text formatters or for automatic translation to a variety of formats suitable for input to text formatters. A copy made in an otherwise Transparent file format whose markup has been designed to thwart or discourage subsequent modification by readers is not Transparent. A copy that is not "Transparent" is called "Opaque".

Examples of suitable formats for Transparent copies include plain ASCII without markup, Texinfo input format, LaTeX input format, SGML or XML using a publicly available DTD, and standard-conforming simple HTML designed for human modification. Opaque formats include PostScript, PDF, proprietary formats that can be read and edited only by proprietary word processors, SGML or XML for which the DTD and/or processing tools are not generally available, and the machine-generated HTML produced by some word processors for output purposes only.

The "Title Page" means, for a printed book, the title page itself, plus such following pages as are needed to hold, legibly, the material this License requires to appear in the title page. For works in formats which do not have any title page as such, "Title Page" means the text near the most prominent appearance of the work’s title, preceding the beginning of the body of the text.

2. VERBATIM COPYING

You may copy and distribute the Document in any medium, either commercially or noncommercially, provided that this License, the copyright notices, and the license notice saying this License applies to the Document are reproduced in all copies, and that you add no other conditions whatsoever to those of this License. You may not use technical measures to obstruct or control the reading or further copying of the copies you make or distribute. However, you may accept compensation in exchange for copies. If you distribute a large enough number of copies you must also follow the conditions in section 3.

You may also lend copies, under the same conditions stated above, and you may publicly display copies.

3. COPYING IN QUANTITY

If you publish printed copies of the Document numbering more than 100, and the Document's license notice requires Cover Texts, you must enclose the copies in covers that carry, clearly and legibly, all these Cover Texts: Front-Cover Texts on the front cover, and Back-Cover Texts on the back cover. Both covers must also clearly and legibly identify you as the publisher of these copies. The front cover must present the full title with all words of the title equally prominent and visible. You may add other material on the covers in addition. Copying with changes limited to the covers, as long as they preserve the title of the Document and satisfy these conditions, can be treated as verbatim copying in other respects.

If the required texts for either cover are too voluminous to fit legibly, you should put the first ones listed (as many as fit reasonably) on the actual cover, and continue the rest onto adjacent pages.

If you publish or distribute Opaque copies of the Document numbering more than 100, you must either include a machine-readable Transparent copy along with each Opaque copy, or state in or with each Opaque copy a publicly-accessible computer-network location containing a complete Transparent copy of the Document, free of added material, which the general network-using public has access to download anonymously at no charge using public-standard network protocols. If you use the latter option, you must take reasonably prudent steps, when you begin distribution of Opaque copies in quantity, to ensure that this Transparent copy will remain thus accessible at the stated location until at least one year after the last time you distribute an Opaque copy (directly or through your agents or retailers) of that edition to the public.

It is requested, but not required, that you contact the authors of the Document well before redistributing any large number of copies, to give them a chance to provide you with an updated version of the Document.

4. MODIFICATIONS

You may copy and distribute a Modified Version of the Document under the conditions of sections 2 and 3 above, provided that you release the Modified Version under precisely this License, with the Modified Version filling the role of the Document, thus licensing distribution and modification of the Modified Version to whoever possesses a copy of it. In addition, you must do these things in the Modified Version:

A. Use in the Title Page (and on the covers, if any) a title distinct from that of the Document, and from those of previous versions (which should, if there were any, be listed in the History section of the Document). You may use the same title as a previous version if the original publisher of that version gives permission.

B. List on the Title Page, as authors, one or more persons or entities responsible for authorship of the modifications in the Modified Version, together with at least five of the principal authors of the Document (all of its principal authors, if it has less than five).

C. State on the Title page the name of the publisher of the Modified Version, as the publisher.

D. Preserve all the copyright notices of the Document.

E. Add an appropriate copyright notice for your modifications adjacent to the other copyright notices.

F. Include, immediately after the copyright notices, a license notice giving the public permission to use the Modified Version under theterms of this License, in the form shown in the Addendum below.

G. Preserve in that license notice the full lists of Invariant Sections and required Cover Texts given in the Document's license notice.

H. Include an unaltered copy of this License.

I. Preserve the section entitled "History", and its title, and add to it an item stating at least the title, year, new authors, and publisher of the Modified Version as given on the Title Page. If there is no section entitled "History" in the Document, create one stating the title, year, authors, and publisher of the Document as given on its Title Page, then add an item describing the Modified Version as stated in the previous sentence.

J. Preserve the network location, if any, given in the Document for public access to a Transparent copy of the Document, and likewise the network locations given in the Document for previous versions it was based on. These may be placed in the "History" section. You may omit a network location for a work that was published at least four years before the Document itself, or if the original publisher of the version it refers to gives permission.

K. In any section entitled "Acknowledgements" or "Dedications", preserve the section's title, and preserve in the section all the substance and tone of each of the contributor acknowledgements and/or dedications given therein.

L. Preserve all the Invariant Sections of the Document, unaltered in their text and in their titles. Section numbers or the equivalent are not considered part of the section titles.

M. Delete any section entitled "Endorsements". Such a section may not be included in the Modified Version.

N. Do not retitle any existing section as "Endorsements" or to conflict in title with any Invariant Section. If the Modified Version includes new front-matter sections or appendices that qualify as Secondary Sections and contain no material copied from the Document, you may at your option designate some or all of these sections as invariant. To do this, add their titles to the list of Invariant Sections in the Modified Version's license notice. These titles must be distinct from any other section titles.

You may add a section entitled "Endorsements", provided it contains nothing but endorsements of your Modified Version by various parties--for example, statements of peer review or that the text has been approved by an organization as the authoritative definition of a standard.

You may add a passage of up to five words as a Front-Cover Text, and a passage of up to 25 words as a Back-Cover Text, to the end of the list of Cover Texts in the Modified Version. Only one passage of Front-Cover Text and one of Back-Cover Text may be added by (or through arrangements made by) any one entity. If the Document already includes a cover text for the same cover, previously added by you or by arrangement made by the same entity you are acting on behalf of, you may not add another; but you may replace the old one, on explicit permission from the previous publisher that added the old one.

The author(s) and publisher(s) of the Document do not by this License give permission to use their names for publicity for or to assert or imply endorsement of any Modified Version.

5. COMBINING DOCUMENTS

You may combine the Document with other documents released under this License, under the terms defined in section 4 above for modified versions, provided that you include in the combination all of the Invariant Sections of all of the original documents, unmodified, and list them all as Invariant Sections of your combined work in its license notice.

The combined work need only contain one copy of this License, and multiple identical Invariant Sections may be replaced with a single copy. If there are multiple Invariant Sections with the same name but different contents, make the title of each such section unique by adding at the end of it, in parentheses, the name of the original author or publisher of that section if known, or else a unique number.

Make the same adjustment to the section titles in the list of Invariant Sections in the license notice of the combined work.

In the combination, you must combine any sections entitled "History" in the various original documents, forming one section entitled "History"; likewise combine any sections entitled "Acknowledgements", and any sections entitled "Dedications". You must delete all sections entitled "Endorsements."

6. COLLECTIONS OF DOCUMENTS

You may make a collection consisting of the Document and other documents released under this License, and replace the individual copies of this License in the various documents with a single copy that is included in the collection, provided that you follow the rules of this License for verbatim copying of each of the documents in all other respects.

You may extract a single document from such a collection, and distribute it individually under this License, provided you insert a copy of this License into the extracted document, and follow this License in all other respects regarding verbatim copying of that document.

7. AGGREGATION WITH INDEPENDENT WORKS

A compilation of the Document or its derivatives with other separate and independent documents or works, in or on a volume of a storage or distribution medium, does not as a whole count as a Modified Version of the Document, provided no compilation copyright is claimed for the compilation. Such a compilation is called an "aggregate", and this License does not apply to the other self-contained works thus compiled with the Document, on account of their being thus compiled, if they are not themselves derivative works of the Document.

If the Cover Text requirement of section 3 is applicable to these copies of the Document, then if the Document is less than one quarter of the entire aggregate, the Document's Cover Texts may be placed on covers that surround only the Document within the aggregate. Otherwise they must appear on covers around the whole aggregate.

8. TRANSLATION

Translation is considered a kind of modification, so you may distribute translations of the Document under the terms of section 4. Replacing Invariant Sections with translations requires special permission from their copyright holders, but you may include translations of some or all Invariant Sections in addition to the original versions of these Invariant Sections. You may include a translation of this License provided that you also include the original English version of this License. In case of a disagreement between the translation and the original English version of this License, the original English version will prevail.

9. TERMINATION

You may not copy, modify, sublicense, or distribute the Document except as expressly provided for under this License. Any other attempt to copy, modify, sublicense or distribute the Document is void, and will automatically terminate your rights under this License. However, parties who have received copies, or rights, from you under this License will not have their licenses terminated so long as such parties remain in full compliance.

10. FUTURE REVISIONS OF THIS LICENSE

The Free Software Foundation may publish new, revised versions of the GNU Free Documentation License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns. See http://www.gnu.org/copyleft/.

Each version of the License is given a distinguishing version number. If the Document specifies that a particular numbered version of this License "or any later version" applies to it, you have the option of following the terms and conditions either of that specified version or of any later version that has been published (not as a draft) by the Free Software Foundation. If the Document does not specify a version number of this License, you may choose any version ever published (not as a draft) by the Free Software Foundation.

ADDENDUM: How to use this License for your documents

To use this License in a document you have written, include a copy of the License in the document and put the following copyright and license notices just after the title page:

> Copyright (c) YEAR YOUR NAME.
> Permission is granted to copy, distribute and/or modify this document under the terms of the GNU Free Documentation License, Version 1.1 or any later version published by the Free Software Foundation; with the Invariant Sections being LIST THEIR TITLES, with the Front-Cover Texts being LIST, and with the Back-Cover Texts being LIST. A copy of the license is included in the section entitled "GNU Free Documentation License".

If you have no Invariant Sections, write "with no Invariant Sections" instead of saying which ones are invariant. If you have no Front-Cover Texts, write "no Front-Cover Texts" instead of "Front-Cover Texts being LIST"; likewise for Back-Cover Texts.

If your document contains nontrivial examples of program code, we recommend releasing these examples in parallel under your choice of free software license, such as the GNU General Public License, to permit their use in free software.

Document generated by eLyXer 1.1.0 (2010-11-23) on 2010-11-29T10:47:56.288252