Gforth

for version 0.6.1, March 11, 2003

Neal Crook
Anton Ertl
David Kuehling
Bernd Paysan
Jens Wilke

This manual is for Gforth (version 0.6.1, March 11, 2003), a fast and portable implementation of the ANS Forth language

Table of Contents

Preface

This manual documents Gforth. Some introductory material is provided for readers who are unfamiliar with Forth or who are migrating to Gforth from other Forth compilers. However, this manual is primarily a reference manual.

1 Goals of Gforth

The goal of the Gforth Project is to develop a standard model for ANS Forth. This can be split into several subgoals:

- Gforth should conform to the ANS Forth Standard.
- It should be a model, i.e. it should define all the implementation-dependent things.
- It should become standard, i.e. widely accepted and used. This goal is the most difficult one.

To achieve these goals Gforth should be

- Similar to previous models (fig-Forth, F83)
- Powerful. It should provide for all the things that are considered necessary today and even some that are not yet considered necessary.
- Efficient. It should not get the reputation of being exceptionally slow.
- Free.
- Available on many machines/easy to port.

Have we achieved these goals? Gforth conforms to the ANS Forth standard. It may be considered a model, but we have not yet documented which parts of the model are stable and which parts we are likely to change. It certainly has not yet become a de facto standard, but it appears to be quite popular. It has some similarities to and some differences from previous models. It has some powerful features, but not yet everything that we envisioned. We certainly have achieved our execution speed goals (see Section 14.4 [Performance], page 199)[1]. It is free and available on many machines.

[1] However, in 1998 the bar was raised when the major commercial Forth vendors switched to native code compilers.

2 Gforth Environment

Note: ultimately, the Gforth man page will be auto-generated from the material in this chapter.

For related information about the creation of images see Chapter 13 [Image Files], page 187.

2.1 Invoking Gforth

Gforth is made up of two parts; an executable "engine" (named `gforth` or `gforth-fast`) and an image file. To start it, you will usually just say `gforth` – this automatically loads the default image file '`gforth.fi`'. In many other cases the default Gforth image will be invoked like this:

```
gforth [file | -e forth-code] ...
```

This interprets the contents of the files and the Forth code in the order they are given.

In addition to the `gforth` engine, there is also an engine called `gforth-fast`, which is faster, but gives less informative error messages (see Chapter 6 [Error messages], page 164) and may catch some stack underflows later or not at all. You should use it for debugged, performance-critical programs.

Moreover, there is an engine called `gforth-itc`, which is useful in some backwards-compatibility situations (see Section 14.2.2 [Direct or Indirect Threaded?], page 194).

In general, the command line looks like this:

```
gforth[-fast] [engine options] [image options]
```

The engine options must come before the rest of the command line. They are:

`--image-file` *file*
`-i` *file*
: Loads the Forth image *file* instead of the default '`gforth.fi`' (see Chapter 13 [Image Files], page 187).

`--appl-image` *file*
: Loads the image *file* and leaves all further command-line arguments to the image (instead of processing them as engine options). This is useful for building executable application images on Unix, built with `gforthmi --application ...`.

`--path` *path*
`-p` *path*
: Uses *path* for searching the image file and Forth source code files instead of the default in the environment variable `GFORTHPATH` or the path specified at installation time (e.g., '`/usr/local/share/gforth/0.2.0:.`'). A path is given as a list of directories, separated by '`:`' (on Unix) or '`;`' (on other OSs).

`--dictionary-size` *size*
`-m` *size*
: Allocate *size* space for the Forth dictionary space instead of using the default specified in the image (typically 256K). The *size* specification for this and subsequent options consists of an integer and a unit (e.g., `4M`). The unit can be one of `b` (bytes), `e` (element size, in this case Cells), `k` (kilobytes), `M` (Megabytes), `G` (Gigabytes), and `T` (Terabytes). If no unit is specified, `e` is used.

`--data-stack-size` *size*

`-d` *size* Allocate *size* space for the data stack instead of using the default specified in the image (typically 16K).

`--return-stack-size` *size*

`-r` *size* Allocate *size* space for the return stack instead of using the default specified in the image (typically 15K).

`--fp-stack-size` *size*

`-f` *size* Allocate *size* space for the floating point stack instead of using the default specified in the image (typically 15.5K). In this case the unit specifier `e` refers to floating point numbers.

`--locals-stack-size` *size*

`-l` *size* Allocate *size* space for the locals stack instead of using the default specified in the image (typically 14.5K).

`--help`

`-h` Print a message about the command-line options

`--version`

`-v` Print version and exit

`--debug` Print some information useful for debugging on startup.

`--offset-image`

Start the dictionary at a slightly different position than would be used otherwise (useful for creating data-relocatable images, see Section 13.4 [Data-Relocatable Image Files], page 188).

`--no-offset-im`

Start the dictionary at the normal position.

`--clear-dictionary`

Initialize all bytes in the dictionary to 0 before loading the image (see Section 13.4 [Data-Relocatable Image Files], page 188).

`--die-on-signal`

Normally Gforth handles most signals (e.g., the user interrupt SIGINT, or the segmentation violation SIGSEGV) by translating it into a Forth `THROW`. With this option, Gforth exits if it receives such a signal. This option is useful when the engine and/or the image might be severely broken (such that it causes another signal before recovering from the first); this option avoids endless loops in such cases.

`--no-dynamic`

`--dynamic`

Disable or enable dynamic superinstructions with replication (see Section 14.2.3 [Dynamic Superinstructions], page 194).

`--no-super`

Disable dynamic superinstructions, use just dynamic replication; this is useful if you want to patch threaded code (see Section 14.2.3 [Dynamic Superinstructions], page 194).

As explained above, the image-specific command-line arguments for the default image '`gforth.fi`' consist of a sequence of filenames and `-e` *forth-code* options that are interpreted in the sequence in which they are given. The `-e` *forth-code* or `--evaluate` *forth-code* option evaluates the Forth code. This option takes only one argument; if you want to evaluate more Forth words, you have to quote them or use `-e` several times. To exit after processing the command line (instead of entering interactive mode) append `-e bye` to the command line.

If you have several versions of Gforth installed, `gforth` will invoke the version that was installed last. `gforth-`*version* invokes a specific version. If your environment contains the variable `GFORTHPATH`, you may want to override it by using the `--path` option.

Not yet implemented: On startup the system first executes the system initialization file (unless the option `--no-init-file` is given; note that the system resulting from using this option may not be ANS Forth conformant). Then the user initialization file '`.gforth.fs`' is executed, unless the option `--no-rc` is given; this file is searched for in '`.`', then in '`~`', then in the normal path (see above).

2.2 Leaving Gforth

You can leave Gforth by typing `bye` or *`Ctrl-d`* (at the start of a line) or (if you invoked Gforth with the `--die-on-signal` option) *`Ctrl-c`*. When you leave Gforth, all of your definitions and data are discarded. For ways of saving the state of the system before leaving Gforth see Chapter 13 [Image Files], page 187.

`bye` – tools-ext "bye"

Return control to the host operating system (if any).

2.3 Command-line editing

Gforth maintains a history file that records every line that you type to the text interpreter. This file is preserved between sessions, and is used to provide a command-line recall facility; if you type *`Ctrl-P`* repeatedly you can recall successively older commands from this (or previous) session(s). The full list of command-line editing facilities is:

- *`Ctrl-p`* ("previous") (or up-arrow) to recall successively older commands from the history buffer.
- *`Ctrl-n`* ("next") (or down-arrow) to recall successively newer commands from the history buffer.
- *`Ctrl-f`* (or right-arrow) to move the cursor right, non-destructively.
- *`Ctrl-b`* (or left-arrow) to move the cursor left, non-destructively.
- *`Ctrl-h`* (backspace) to delete the character to the left of the cursor, closing up the line.
- *`Ctrl-k`* to delete ("kill") from the cursor to the end of the line.
- *`Ctrl-a`* to move the cursor to the start of the line.
- *`Ctrl-e`* to move the cursor to the end of the line.
- ⟨RET⟩ (*`Ctrl-m`*) or ⟨LFD⟩ (*`Ctrl-j`*) to submit the current line.
- ⟨TAB⟩ to step through all possible full-word completions of the word currently being typed.

- *Ctrl-d* on an empty line line to terminate Gforth (gracefully, using `bye`).
- *Ctrl-x* (or `Ctrl-d` on a non-empty line) to delete the character under the cursor.

When editing, displayable characters are inserted to the left of the cursor position; the line is always in "insert" (as opposed to "overstrike") mode.

On Unix systems, the history file is '`~/.gforth-history`' by default[1]. You can find out the name and location of your history file using:

```
history-file type \ Unix-class systems

history-file type \ Other systems
history-dir  type
```

If you enter long definitions by hand, you can use a text editor to paste them out of the history file into a Forth source file for reuse at a later time.

Gforth never trims the size of the history file, so you should do this periodically, if necessary.

2.4 Environment variables

Gforth uses these environment variables:

- `GFORTHHIST` – (Unix systems only) specifies the directory in which to open/create the history file, '`.gforth-history`'. Default: `$HOME`.
- `GFORTHPATH` – specifies the path used when searching for the gforth image file and for Forth source-code files.
- `GFORTH` – used by '`gforthmi`', See Section 13.5.1 [gforthmi], page 189.
- `GFORTHD` – used by '`gforthmi`', See Section 13.5.1 [gforthmi], page 189.
- `TMP`, `TEMP` - (non-Unix systems only) used as a potential location for the history file.

All the Gforth environment variables default to sensible values if they are not set.

2.5 Gforth files

When you install Gforth on a Unix system, it installs files in these locations by default:

- '`/usr/local/bin/gforth`'
- '`/usr/local/bin/gforthmi`'
- '`/usr/local/man/man1/gforth.1`' - man page.
- '`/usr/local/info`' - the Info version of this manual.
- '`/usr/local/lib/gforth/<version>/...`' - Gforth '`.fi`' files.
- '`/usr/local/share/gforth/<version>/TAGS`' - Emacs TAGS file.
- '`/usr/local/share/gforth/<version>/...`' - Gforth source files.
- '`.../emacs/site-lisp/gforth.el`' - Emacs gforth mode.

You can select different places for installation by using `configure` options (listed with `configure --help`).

[1] i.e. it is stored in the user's home directory.

2.6 Gforth in pipes

Gforth can be used in pipes created elsewhere (described here). It can also create pipes on its own (see Section 5.19.6 [Pipes], page 120).

If you pipe into Gforth, your program should read with `read-file` or `read-line` from `stdin` (see Section 5.17.2 [General files], page 107). `Key` does not recognize the end of input. Words like `accept` echo the input and are therefore usually not useful for reading from a pipe. You have to invoke the Forth program with an OS command-line option, as you have no chance to use the Forth command line (the text interpreter would try to interpret the pipe input).

You can output to a pipe with `type`, `emit`, `cr` etc.

When you write to a pipe that has been closed at the other end, Gforth receives a SIGPIPE signal ("pipe broken"). Gforth translates this into the exception `broken-pipe-error`. If your application does not catch that exception, the system catches it and exits, usually silently (unless you were working on the Forth command line; then it prints an error message and exits). This is usually the desired behaviour.

If you do not like this behaviour, you have to catch the exception yourself, and react to it.

Here's an example of an invocation of Gforth that is usable in a pipe:

```
gforth -e ": foo begin pad dup 10 stdin read-file throw dup while \
 type repeat ; foo bye"
```

This example just copies the input verbatim to the output. A very simple pipe containing this example looks like this:

```
cat startup.fs |
gforth -e ": foo begin pad dup 80 stdin read-file throw dup while \
 type repeat ; foo bye"|
head
```

Pipes involving Gforth's `stderr` output do not work.

2.7 Startup speed

If Gforth is used for CGI scripts or in shell scripts, its startup speed may become a problem. On a 300MHz 21064a under Linux-2.2.13 with glibc-2.0.7, `gforth -e bye` takes about 24.6ms user and 11.3ms system time.

If startup speed is a problem, you may consider the following ways to improve it; or you may consider ways to reduce the number of startups (for example, by using Fast-CGI).

An easy step that influences Gforth startup speed is the use of the '`--no-dynamic`' option; this decreases image loading speed, but increases compile-time and run-time.

Another step to improve startup speed is to statically link Gforth, by building it with `XLDFLAGS=-static`. This requires more memory for the code and will therefore slow down the first invocation, but subsequent invocations avoid the dynamic linking overhead. Another disadvantage is that Gforth won't profit from library upgrades. As a result, `gforth-static -e bye` takes about 17.1ms user and 8.2ms system time.

The next step to improve startup speed is to use a non-relocatable image (see Section 13.3 [Non-Relocatable Image Files], page 188). You can create this image with `gforth -e "savesystem gforthnr.fi bye"` and later use it with `gforth -i gforthnr.fi ...`. This avoids the relocation overhead and a part of the copy-on-write overhead. The disadvantage is that the non-relocatable image does not work if the OS gives Gforth a different address for the dictionary, for whatever reason; so you better provide a fallback on a relocatable image. `gforth-static -i gforthnr.fi -e bye` takes about 15.3ms user and 7.5ms system time.

The final step is to disable dictionary hashing in Gforth. Gforth builds the hash table on startup, which takes much of the startup overhead. You can do this by commenting out the `include hash.fs` in '`startup.fs`' and everything that requires '`hash.fs`' (at the moment '`table.fs`' and '`ekey.fs`') and then doing `make`. The disadvantages are that functionality like `table` and `ekey` is missing and that text interpretation (e.g., compiling) now takes much longer. So, you should only use this method if there is no significant text interpretation to perform (the script should be compiled into the image, amongst other things). `gforth-static -i gforthnrnh.fi -e bye` takes about 2.1ms user and 6.1ms system time.

3 Forth Tutorial

The difference of this chapter from the Introduction (see Chapter 4 [Introduction], page 37) is that this tutorial is more fast-paced, should be used while sitting in front of a computer, and covers much more material, but does not explain how the Forth system works.

This tutorial can be used with any ANS-compliant Forth; any Gforth-specific features are marked as such and you can skip them if you work with another Forth. This tutorial does not explain all features of Forth, just enough to get you started and give you some ideas about the facilities available in Forth. Read the rest of the manual and the standard when you are through this.

The intended way to use this tutorial is that you work through it while sitting in front of the console, take a look at the examples and predict what they will do, then try them out; if the outcome is not as expected, find out why (e.g., by trying out variations of the example), so you understand what's going on. There are also some assignments that you should solve.

This tutorial assumes that you have programmed before and know what, e.g., a loop is.

3.1 Starting Gforth

You can start Gforth by typing its name:

```
gforth
```

That puts you into interactive mode; you can leave Gforth by typing `bye`. While in Gforth, you can edit the command line and access the command line history with cursor keys, similar to bash.

3.2 Syntax

A *word* is a sequence of arbitrary characters (expcept white space). Words are separated by white space. E.g., each of the following lines contains exactly one word:

```
word
!@#$%^&*()
1234567890
5!a
```

A frequent beginner's error is to leave away necessary white space, resulting in an error like '`Undefined word`'; so if you see such an error, check if you have put spaces wherever necessary.

```
." hello, world" \ correct
."hello, world"  \ gives an "Undefined word" error
```

Gforth and most other Forth systems ignore differences in case (they are case-insensitive), i.e., '`word`' is the same as '`Word`'. If your system is case-sensitive, you may have to type all the examples given here in upper case.

3.3 Crash Course

Type

```
0 0 !
here execute
' catch >body 20 erase abort
' (quit) >body 20 erase
```

The last two examples are guaranteed to destroy parts of Gforth (and most other systems), so you better leave Gforth afterwards (if it has not finished by itself). On some systems you may have to kill gforth from outside (e.g., in Unix with `kill`).

Now that you know how to produce crashes (and that there's not much to them), let's learn how to produce meaningful programs.

3.4 Stack

The most obvious feature of Forth is the stack. When you type in a number, it is pushed on the stack. You can display the content of the stack with `.s`.

```
1 2 .s
3 .s
```

`.s` displays the top-of-stack to the right, i.e., the numbers appear in `.s` output as they appeared in the input.

You can print the top of stack element with `.`.

```
1 2 3 . . .
```

In general, words consume their stack arguments (`.s` is an exception).

Assignment:

What does the stack contain after `5 6 7 .`?

3.5 Arithmetics

The words `+`, `-`, `*`, `/`, and `mod` always operate on the top two stack items:

```
2 2 .s
+ .s
.
2 1 - .
7 3 mod .
```

The operands of `-`, `/`, and `mod` are in the same order as in the corresponding infix expression (this is generally the case in Forth).

Parentheses are superfluous (and not available), because the order of the words unambiguously determines the order of evaluation and the operands:

```
3 4 + 5 * .
3 4 5 * + .
```

Assignment:

What are the infix expressions corresponding to the Forth code above? Write `6-7*8+9` in Forth notation[1].

[1] This notation is also known as Postfix or RPN (Reverse Polish Notation).

To change the sign, use `negate`:

```
2 negate .
```

Assignment:
Convert -(-3)*4-5 to Forth.

`/mod` performs both `/` and `mod`.

```
7 3 /mod . .
```

Reference: Section 5.5 [Arithmetic], page 50.

3.6 Stack Manipulation

Stack manipulation words rearrange the data on the stack.

```
1 .s drop .s
1 .s dup .s drop drop .s
1 2 .s over .s drop drop drop
1 2 .s swap .s drop drop
1 2 3 .s rot .s drop drop drop
```

These are the most important stack manipulation words. There are also variants that manipulate twice as many stack items:

```
1 2 3 4 .s 2swap .s 2drop 2drop
```

Two more stack manipulation words are:

```
1 2 .s nip .s drop
1 2 .s tuck .s 2drop drop
```

Assignment:
Replace `nip` and `tuck` with combinations of other stack manipulation words.

```
Given:          How do you get:
1 2 3           3 2 1
1 2 3           1 2 3 2
1 2 3           1 2 3 3
1 2 3           1 3 3
1 2 3           2 1 3
1 2 3 4         4 3 2 1
1 2 3           1 2 3 1 2 3
1 2 3 4         1 2 3 4 1 2
1 2 3
1 2 3           1 2 3 4
1 2 3           1 3
```

```
5 dup * .
```

Assignment:
Write 17^3 and 17^4 in Forth, without writing `17` more than once. Write a piece of Forth code that expects two numbers on the stack (*a* and *b*, with *b* on top) and computes `(a-b)(a+1)`.

Reference: Section 5.6 [Stack Manipulation], page 56.

3.7 Using files for Forth code

While working at the Forth command line is convenient for one-line examples and short one-off code, you probably want to store your source code in files for convenient editing and persistence. You can use your favourite editor (Gforth includes Emacs support, see Chapter 12 [Emacs and Gforth], page 184) to create *file.fs* and use

```
s" file.fs" included
```

to load it into your Forth system. The file name extension I use for Forth files is '`.fs`'.

You can easily start Gforth with some files loaded like this:

```
gforth file1.fs file2.fs
```

If an error occurs during loading these files, Gforth terminates, whereas an error during `INCLUDED` within Gforth usually gives you a Gforth command line. Starting the Forth system every time gives you a clean start every time, without interference from the results of earlier tries.

I often put all the tests in a file, then load the code and run the tests with

```
gforth code.fs tests.fs -e bye
```

(often by performing this command with *C-x C-e* in Emacs). The `-e bye` ensures that Gforth terminates afterwards so that I can restart this command without ado.

The advantage of this approach is that the tests can be repeated easily every time the program ist changed, making it easy to catch bugs introduced by the change.

Reference: Section 5.17.1 [Forth source files], page 106.

3.8 Comments

```
\ That's a comment; it ends at the end of the line
( Another comment; it ends here: )  .s
```

`\` and `(` are ordinary Forth words and therefore have to be separated with white space from the following text.

```
\This gives an "Undefined word" error
```

The first `)` ends a comment started with `(`, so you cannot nest `(`-comments; and you cannot comment out text containing a `)` with `( ... )`[2].

I use `\`-comments for descriptive text and for commenting out code of one or more line; I use `(`-comments for describing the stack effect, the stack contents, or for commenting out sub-line pieces of code.

The Emacs mode '`gforth.el`' (see Chapter 12 [Emacs and Gforth], page 184) supports these uses by commenting out a region with *C-x* `\`, uncommenting a region with *C-u C-x* `\`, and filling a `\`-commented region with *M-q*.

Reference: Section 5.3 [Comments], page 50.

[2] therefore it's a good idea to avoid `)` in word names.

3.9 Colon Definitions

are similar to procedures and functions in other programming languages.

```
: squared ( n -- n^2 )
   dup * ;
5 squared .
7 squared .
```

: starts the colon definition; its name is `squared`. The following comment describes its stack effect. The words `dup *` are not executed, but compiled into the definition. `;` ends the colon definition.

The newly-defined word can be used like any other word, including using it in other definitions:

```
: cubed ( n -- n^3 )
   dup squared * ;
-5 cubed .
: fourth-power ( n -- n^4 )
   squared squared ;
3 fourth-power .
```

Assignment:
Write colon definitions for `nip`, `tuck`, `negate`, and `/mod` in terms of other Forth words, and check if they work (hint: test your tests on the originals first). Don't let the '`redefined`'-Messages spook you, they are just warnings.

Reference: Section 5.9.5 [Colon Definitions], page 75.

3.10 Decompilation

You can decompile colon definitions with `see`:

```
see squared
see cubed
```

In Gforth `see` shows you a reconstruction of the source code from the executable code. Informations that were present in the source, but not in the executable code, are lost (e.g., comments).

You can also decompile the predefined words:

```
see .
see +
```

3.11 Stack-Effect Comments

By convention the comment after the name of a definition describes the stack effect: The part in from of the '`--`' describes the state of the stack before the execution of the definition, i.e., the parameters that are passed into the colon definition; the part behind the '`--`' is the state of the stack after the execution of the definition, i.e., the results of the definition. The stack comment only shows the top stack items that the definition accesses and/or changes.

You should put a correct stack effect on every definition, even if it is just `( -- )`. You should also add some descriptive comment to more complicated words (I usually do this in the lines following `:`). If you don't do this, your code becomes unreadable (because you have to work through every definition before you can undertsand any).

Assignment:

The stack effect of `swap` can be written like this: `x1 x2 -- x2 x1`. Describe the stack effect of `-`, `drop`, `dup`, `over`, `rot`, `nip`, and `tuck`. Hint: When you are done, you can compare your stack effects to those in this manual (see [Word Index], page 221).

Sometimes programmers put comments at various places in colon definitions that describe the contents of the stack at that place (stack comments); i.e., they are like the first part of a stack-effect comment. E.g.,

```
: cubed ( n -- n^3 )
   dup squared  ( n n^2 ) * ;
```

In this case the stack comment is pretty superfluous, because the word is simple enough. If you think it would be a good idea to add such a comment to increase readability, you should also consider factoring the word into several simpler words (see Section 3.13 [Factoring], page 15), which typically eliminates the need for the stack comment; however, if you decide not to refactor it, then having such a comment is better than not having it.

The names of the stack items in stack-effect and stack comments in the standard, in this manual, and in many programs specify the type through a type prefix, similar to Fortran and Hungarian notation. The most frequent prefixes are:

`n` signed integer

`u` unsigned integer

`c` character

`f` Boolean flags, i.e. `false` or `true`.

`a-addr,a-`
Cell-aligned address

`c-addr,c-`
Char-aligned address (note that a Char may have two bytes in Windows NT)

`xt` Execution token, same size as Cell

`w,x` Cell, can contain an integer or an address. It usually takes 32, 64 or 16 bits (depending on your platform and Forth system). A cell is more commonly known as machine word, but the term *word* already means something different in Forth.

`d` signed double-cell integer

`ud` unsigned double-cell integer

`r` Float (on the FP stack)

You can find a more complete list in Section 5.1 [Notation], page 48.

Assignment:

Write stack-effect comments for all definitions you have written up to now.

3.12 Types

In Forth the names of the operations are not overloaded; so similar operations on different types need different names; e.g., `+` adds integers, and you have to use `f+` to add floating-point numbers. The following prefixes are often used for related operations on different types:

`(none)` signed integer

`u` unsigned integer

`c` character

`d` signed double-cell integer

`ud, du` unsigned double-cell integer

`2` two cells (not-necessarily double-cell numbers)

`m, um` mixed single-cell and double-cell operations

`f` floating-point (note that in stack comments '`f`' represents flags, and '`r`' represents FP numbers).

If there are no differences between the signed and the unsigned variant (e.g., for `+`), there is only the prefix-less variant.

Forth does not perform type checking, neither at compile time, nor at run time. If you use the wrong oeration, the data are interpreted incorrectly:

```
-1 u.
```

If you have only experience with type-checked languages until now, and have heard how important type-checking is, don't panic! In my experience (and that of other Forthers), type errors in Forth code are usually easy to find (once you get used to it), the increased vigilance of the programmer tends to catch some harder errors in addition to most type errors, and you never have to work around the type system, so in most situations the lack of type-checking seems to be a win (projects to add type checking to Forth have not caught on).

3.13 Factoring

If you try to write longer definitions, you will soon find it hard to keep track of the stack contents. Therefore, good Forth programmers tend to write only short definitions (e.g., three lines). The art of finding meaningful short definitions is known as factoring (as in factoring polynomials).

Well-factored programs offer additional advantages: smaller, more general words, are easier to test and debug and can be reused more and better than larger, specialized words.

So, if you run into difficulties with stack management, when writing code, try to define meaningful factors for the word, and define the word in terms of those. Even if a factor contains only two words, it is often helpful.

Good factoring is not easy, and it takes some practice to get the knack for it; but even experienced Forth programmers often don't find the right solution right away, but only when rewriting the program. So, if you don't come up with a good solution immediately, keep trying, don't despair.

3.14 Designing the stack effect

In other languages you can use an arbitrary order of parameters for a function; and since there is only one result, you don't have to deal with the order of results, either.

In Forth (and other stack-based languages, e.g., Postscript) the parameter and result order of a definition is important and should be designed well. The general guideline is to design the stack effect such that the word is simple to use in most cases, even if that complicates the implementation of the word. Some concrete rules are:

- Words consume all of their parameters (e.g., `.`).
- If there is a convention on the order of parameters (e.g., from mathematics or another programming language), stick with it (e.g., `-`).
- If one parameter usually requires only a short computation (e.g., it is a constant), pass it on the top of the stack. Conversely, parameters that usually require a long sequence of code to compute should be passed as the bottom (i.e., first) parameter. This makes the code easier to read, because reader does not need to keep track of the bottom item through a long sequence of code (or, alternatively, through stack manipulations). E.g., `!` (store, see Section 5.7 [Memory], page 58) expects the address on top of the stack because it is usually simpler to compute than the stored value (often the address is just a variable).
- Similarly, results that are usually consumed quickly should be returned on the top of stack, whereas a result that is often used in long computations should be passed as bottom result. E.g., the file words like `open-file` return the error code on the top of stack, because it is usually consumed quickly by `throw`; moreover, the error code has to be checked before doing anything with the other results.

These rules are just general guidelines, don't lose sight of the overall goal to make the words easy to use. E.g., if the convention rule conflicts with the computation-length rule, you might decide in favour of the convention if the word will be used rarely, and in favour of the computation-length rule if the word will be used frequently (because with frequent use the cost of breaking the computation-length rule would be quite high, and frequent use makes it easier to remember an unconventional order).

3.15 Local Variables

You can define local variables (*locals*) in a colon definition:

```
: swap { a b -- b a }
  b a ;
1 2 swap .s 2drop
```

(If your Forth system does not support this syntax, include '`compat/anslocals.fs`' first).

In this example `{ a b -- b a }` is the locals definition; it takes two cells from the stack, puts the top of stack in `b` and the next stack element in `a`. `--` starts a comment ending with `}`. After the locals definition, using the name of the local will push its value on the stack. You can leave the comment part (`-- b a`) away:

```
: swap ( x1 x2 -- x2 x1 )
  { a b } b a ;
```

In Gforth you can have several locals definitions, anywhere in a colon definition; in contrast, in a standard program you can have only one locals definition per colon definition, and that locals definition must be outside any controll structure.

With locals you can write slightly longer definitions without running into stack trouble. However, I recommend trying to write colon definitions without locals for exercise purposes to help you gain the essential factoring skills.

Assignment:
Rewrite your definitions until now with locals

Reference: Section 5.20 [Locals], page 121.

3.16 Conditional execution

In Forth you can use control structures only inside colon definitions. An `if`-structure looks like this:

```
: abs ( n1 -- +n2 )
    dup 0 < if
        negate
    endif ;
5 abs .
-5 abs .
```

`if` takes a flag from the stack. If the flag is non-zero (true), the following code is performed, otherwise execution continues after the `endif` (or `else`). `<` compares the top two stack elements and prioduces a flag:

```
1 2 < .
2 1 < .
1 1 < .
```

Actually the standard name for `endif` is `then`. This tutorial presents the examples using `endif`, because this is often less confusing for people familiar with other programming languages where `then` has a different meaning. If your system does not have `endif`, define it with

```
: endif postpone then ; immediate
```

You can optionally use an `else`-part:

```
: min ( n1 n2 -- n )
  2dup < if
    drop
  else
    nip
  endif ;
2 3 min .
3 2 min .
```

Assignment:
Write `min` without `else`-part (hint: what's the definition of `nip`?).

Reference: Section 5.8.1 [Selection], page 64.

3.17 Flags and Comparisons

In a false-flag all bits are clear (0 when interpreted as integer). In a canonical true-flag all bits are set (-1 as a twos-complement signed integer); in many contexts (e.g., `if`) any non-zero value is treated as true flag.

```
false .
true .
true hex u. decimal
```

Comparison words produce canonical flags:

```
1 1 = .
1 0= .
0 1 < .
0 0 < .
-1 1 u< . \ type error, u< interprets -1 as large unsigned number
-1 1 < .
```

Gforth supports all combinations of the prefixes `0 u d d0 du f f0` (or none) and the comparisons `= <> < > <= >=`. Only a part of these combinations are standard (for details see the standard, Section 5.5.4 [Numeric comparison], page 52, Section 5.5.6 [Floating Point], page 54 or [Word Index], page 221).

You can use `and or xor invert` can be used as operations on canonical flags. Actually they are bitwise operations:

```
1 2 and .
1 2 or .
1 3 xor .
1 invert .
```

You can convert a zero/non-zero flag into a canonical flag with `0<>` (and complement it on the way with `0=`).

```
1 0= .
1 0<> .
```

You can use the all-bits-set feature of canonical flags and the bitwise operation of the Boolean operations to avoid `if`s:

```
: foo ( n1 -- n2 )
  0= if
    14
  else
    0
  endif ;
0 foo .
1 foo .

: foo ( n1 -- n2 )
  0= 14 and ;
0 foo .
1 foo .
```

Assignment:

Write `min` without `if`.

For reference, see Section 5.4 [Boolean Flags], page 50, Section 5.5.4 [Numeric comparison], page 52, and Section 5.5.3 [Bitwise operations], page 52.

3.18 General Loops

The endless loop is the most simple one:

```
: endless ( -- )
  0 begin
    dup . 1+
  again ;
endless
```

Terminate this loop by pressing *Ctrl-C* (in Gforth). `begin` does nothing at run-time, `again` jumps back to `begin`.

A loop with one exit at any place looks like this:

```
: log2 ( +n1 -- n2 )
\ logarithmus dualis of n1>0, rounded down to the next integer
  assert( dup 0> )
  2/ 0 begin
    over 0> while
      1+ swap 2/ swap
  repeat
  nip ;
7 log2 .
8 log2 .
```

At run-time `while` consumes a flag; if it is 0, execution continues behind the `repeat`; if the flag is non-zero, execution continues behind the `while`. `Repeat` jumps back to `begin`, just like `again`.

In Forth there are many combinations/abbreviations, like `1+`. However, `2/` is not one of them; it shifts its argument right by one bit (arithmetic shift right):

```
-5 2 / .
-5 2/ .
```

`assert(` is no standard word, but you can get it on systems other then Gforth by including '`compat/assert.fs`'. You can see what it does by trying

```
0 log2 .
```

Here's a loop with an exit at the end:

```
: log2 ( +n1 -- n2 )
\ logarithmus dualis of n1>0, rounded down to the next integer
  assert( dup 0 > )
  -1 begin
    1+ swap 2/ swap
    over 0 <=
  until
  nip ;
```

`Until` consumes a flag; if it is non-zero, execution continues at the `begin`, otherwise after the `until`.

Assignment:
Write a definition for computing the greatest common divisor.

Reference: Section 5.8.2 [Simple Loops], page 65.

3.19 Counted loops

```
: ^ ( n1 u -- n )
\ n = the uth power of u1
  1 swap 0 u+do
    over *
  loop
  nip ;
3 2 ^ .
4 3 ^ .
```

`U+do` (from '`compat/loops.fs`', if your Forth system doesn't have it) takes two numbers of the stack `( u3 u4 -- )`, and then performs the code between `u+do` and `loop` for `u3-u4` times (or not at all, if `u3-u4<0`).

You can see the stack effect design rules at work in the stack effect of the loop start words: Since the start value of the loop is more frequently constant than the end value, the start value is passed on the top-of-stack.

You can access the counter of a counted loop with `i`:

```
: fac ( u -- u! )
  1 swap 1+ 1 u+do
    i *
  loop ;
5 fac .
7 fac .
```

There is also `+do`, which expects signed numbers (important for deciding whether to enter the loop).

Assignment:
Write a definition for computing the nth Fibonacci number.

You can also use increments other than 1:

```
: up2 ( n1 n2 -- )
  +do
    i .
  2 +loop ;
10 0 up2

: down2 ( n1 n2 -- )
  -do
    i .
  2 -loop ;
0 10 down2
```

Reference: Section 5.8.3 [Counted Loops], page 66.

3.20 Recursion

Usually the name of a definition is not visible in the definition; but earlier definitions are usually visible:

```
1 0 / . \ "Floating-point unidentified fault" in Gforth on most platforms
: / ( n1 n2 -- n )
  dup 0= if
    -10 throw \ report division by zero
  endif
  /          \ old version
;
1 0 /
```

For recursive definitions you can use `recursive` (non-standard) or `recurse`:

```
: fac1 ( n -- n! ) recursive
 dup 0> if
   dup 1- fac1 *
 else
   drop 1
 endif ;
7 fac1 .

: fac2 ( n -- n! )
 dup 0> if
   dup 1- recurse *
 else
   drop 1
 endif ;
8 fac2 .
```

Assignment:
Write a recursive definition for computing the nth Fibonacci number.

Reference (including indirect recursion): See Section 5.8.5 [Calls and returns], page 70.

3.21 Leaving definitions or loops

`EXIT` exits the current definition right away. For every counted loop that is left in this way, an `UNLOOP` has to be performed before the `EXIT`:

```
: ...
 ... u+do
   ... if
     ... unloop exit
   endif
   ...
 loop
 ... ;
```

`LEAVE` leaves the innermost counted loop right away:

```
: ...
```

```
   ... u+do
     ... if
       ... leave
     endif
     ...
   loop
   ... ;
```

Reference: Section 5.8.5 [Calls and returns], page 70, Section 5.8.3 [Counted Loops], page 66.

3.22 Return Stack

In addition to the data stack Forth also has a second stack, the return stack; most Forth systems store the return addresses of procedure calls there (thus its name). Programmers can also use this stack:

```
: foo ( n1 n2 -- )
 .s
 >r .s
 r@ .
 >r .s
 r@ .
 r> .
 r@ .
 r> . ;
1 2 foo
```

`>r` takes an element from the data stack and pushes it onto the return stack; conversely, `r>` moves an elementm from the return to the data stack; `r@` pushes a copy of the top of the return stack on the return stack.

Forth programmers usually use the return stack for storing data temporarily, if using the data stack alone would be too complex, and factoring and locals are not an option:

```
: 2swap ( x1 x2 x3 x4 -- x3 x4 x1 x2 )
 rot >r rot r> ;
```

The return address of the definition and the loop control parameters of counted loops usually reside on the return stack, so you have to take all items, that you have pushed on the return stack in a colon definition or counted loop, from the return stack before the definition or loop ends. You cannot access items that you pushed on the return stack outside some definition or loop within the definition of loop.

If you miscount the return stack items, this usually ends in a crash:

```
: crash ( n -- )
  >r ;
5 crash
```

You cannot mix using locals and using the return stack (according to the standard; Gforth has no problem). However, they solve the same problems, so this shouldn't be an issue.

Assignment:

Can you rewrite any of the definitions you wrote until now in a better way using the return stack?

Reference: Section 5.6.3 [Return stack], page 57.

3.23 Memory

You can create a global variable `v` with

```
variable v ( -- addr )
```

`v` pushes the address of a cell in memory on the stack. This cell was reserved by `variable`. You can use `!` (store) to store values into this cell and `@` (fetch) to load the value from the stack into memory:

```
v .
5 v ! .s
v @ .
```

You can see a raw dump of memory with `dump`:

```
v 1 cells .s dump
```

`Cells ( n1 -- n2 )` gives you the number of bytes (or, more generally, address units (aus)) that `n1 cells` occupy. You can also reserve more memory:

```
create v2 20 cells allot
v2 20 cells dump
```

creates a word `v2` and reserves 20 uninitialized cells; the address pushed by `v2` points to the start of these 20 cells. You can use address arithmetic to access these cells:

```
3 v2 5 cells + !
v2 20 cells dump
```

You can reserve and initialize memory with `,`:

```
create v3
  5 , 4 , 3 , 2 , 1 ,
v3 @ .
v3 cell+ @ .
v3 2 cells + @ .
v3 5 cells dump
```

Assignment:

Write a definition `vsum ( addr u -- n )` that computes the sum of `u` cells, with the first of these cells at `addr`, the next one at `addr cell+` etc.

You can also reserve memory without creating a new word:

```
here 10 cells allot .
here .
```

`Here` pushes the start address of the memory area. You should store it somewhere, or you will have a hard time finding the memory area again.

`Allot` manages dictionary memory. The dictionary memory contains the system's data structures for words etc. on Gforth and most other Forth systems. It is managed like a stack: You can free the memory that you have just `allot`ed with

```
-10 cells allot
here .
```

Note that you cannot do this if you have created a new word in the meantime (because then your `allot`ed memory is no longer on the top of the dictionary “stack”).

Alternatively, you can use `allocate` and `free` which allow freeing memory in any order:

```
10 cells allocate throw .s
20 cells allocate throw .s
swap
free throw
free throw
```

The `throw`s deal with errors (e.g., out of memory).

And there is also a garbage collector (`http://www.complang.tuwien.ac.at/forth/garbage-collection.zip`), which eliminates the need to `free` memory explicitly.

Reference: Section 5.7 [Memory], page 58.

3.24 Characters and Strings

On the stack characters take up a cell, like numbers. In memory they have their own size (one 8-bit byte on most systems), and therefore require their own words for memory access:

```
create v4
  104 c, 97 c, 108 c, 108 c, 111 c,
v4 4 chars + c@ .
v4 5 chars dump
```

The preferred representation of strings on the stack is `addr u-count`, where `addr` is the address of the first character and `u-count` is the number of characters in the string.

```
v4 5 type
```

You get a string constant with

```
s" hello, world" .s
type
```

Make sure you have a space between `s"` and the string; `s"` is a normal Forth word and must be delimited with white space (try what happens when you remove the space).

However, this interpretive use of `s"` is quite restricted: the string exists only until the next call of `s"` (some Forth systems keep more than one of these strings, but usually they still have a limited lifetime).

```
s" hello," s" world" .s
type
type
```

You can also use `s"` in a definition, and the resulting strings then live forever (well, for as long as the definition):

```
: foo s" hello," s" world" ;
foo .s
type
type
```

Assignment:

> `Emit ( c -- )` types `c` as character (not a number). Implement `type ( addr u -- )`.

Reference: Section 5.7.6 [Memory Blocks], page 63.

3.25 Alignment

On many processors cells have to be aligned in memory, if you want to access them with `@` and `!` (and even if the processor does not require alignment, access to aligned cells is faster).

`Create` aligns `here` (i.e., the place where the next allocation will occur, and that the `create`d word points to). Likewise, the memory produced by `allocate` starts at an aligned address. Adding a number of `cells` to an aligned address produces another aligned address.

However, address arithmetic involving `char+` and `chars` can create an address that is not cell-aligned. `Aligned ( addr -- a-addr )` produces the next aligned address:

```
v3 char+ aligned .s @ .
v3 char+ .s @ .
```

Similarly, `align` advances `here` to the next aligned address:

```
create v5 97 c,
here .
align here .
1000 ,
```

Note that you should use aligned addresses even if your processor does not require them, if you want your program to be portable.

Reference: Section 5.7.5 [Address arithmetic], page 61.

3.26 Files

This section gives a short introduction into how to use files inside Forth. It's broken up into five easy steps:

1. Opened an ASCII text file for input
2. Opened a file for output
3. Read input file until string matched (or some other condition matched)
4. Wrote some lines from input (modified or not) to output
5. Closed the files.

3.26.1 Open file for input

```
s" foo.in"  r/o open-file throw Value fd-in
```

3.26.2 Create file for output

```
s" foo.out" w/o create-file throw Value fd-out
```

The available file modes are r/o for read-only access, r/w for read-write access, and w/o for write-only access. You could open both files with r/w, too, if you like. All file words return error codes; for most applications, it's best to pass there error codes with `throw` to the outer error handler.

If you want words for opening and assigning, define them as follows:

```
0 Value fd-in
0 Value fd-out
: open-input ( addr u -- )  r/o open-file throw to fd-in ;
: open-output ( addr u -- )  w/o create-file throw to fd-out ;
```

Usage example:

```
s" foo.in" open-input
s" foo.out" open-output
```

3.26.3 Scan file for a particular line

```
256 Constant max-line
Create line-buffer  max-line 2 + allot

: scan-file ( addr u -- )
  begin
      line-buffer max-line fd-in read-line throw
  while
         >r 2dup line-buffer r> compare 0=
     until
  else
     drop
  then
  2drop ;
```

`read-line ( addr u1 fd -- u2 flag ior )` reads up to u1 bytes into the buffer at addr, and returns the number of bytes read, a flag that is false when the end of file is reached, and an error code.

`compare ( addr1 u1 addr2 u2 -- n )` compares two strings and returns zero if both strings are equal. It returns a positive number if the first string is lexically greater, a negative if the second string is lexically greater.

We haven't seen this loop here; it has two exits. Since the `while` exits with the number of bytes read on the stack, we have to clean up that separately; that's after the `else`.

Usage example:

```
s" The text I search is here" scan-file
```

3.26.4 Copy input to output

```
: copy-file ( -- )
  begin
      line-buffer max-line fd-in read-line throw
  while
      line-buffer swap fd-out write-file throw
  repeat ;
```

3.26.5 Close files

```
fd-in close-file throw
fd-out close-file throw
```

Likewise, you can put that into definitions, too:

```
: close-input ( -- )  fd-in close-file throw ;
: close-output ( -- )  fd-out close-file throw ;
```

Assignment:

How could you modify `copy-file` so that it copies until a second line is matched? Can you write a program that extracts a section of a text file, given the line that starts and the line that terminates that section?

3.27 Interpretation and Compilation Semantics and Immediacy

When a word is compiled, it behaves differently from being interpreted. E.g., consider `+`:

```
1 2 + .
: foo + ;
```

These two behaviours are known as compilation and interpretation semantics. For normal words (e.g., `+`), the compilation semantics is to append the interpretation semantics to the currently defined word (`foo` in the example above). I.e., when `foo` is executed later, the interpretation semantics of `+` (i.e., adding two numbers) will be performed.

However, there are words with non-default compilation semantics, e.g., the control-flow words like `if`. You can use `immediate` to change the compilation semantics of the last defined word to be equal to the interpretation semantics:

```
: [FOO] ( -- )
 5 . ; immediate

[FOO]
: bar ( -- )
  [FOO] ;
bar
see bar
```

Two conventions to mark words with non-default compilation semnatics are names with brackets (more frequently used) and to write them all in upper case (less frequently used).

In Gforth (and many other systems) you can also remove the interpretation semantics with `compile-only` (the compilation semantics is derived from the original interpretation semantics):

```
: flip ( -- )
 6 . ; compile-only \ but not immediate
flip

: flop ( -- )
 flip ;
flop
```

In this example the interpretation semantics of `flop` is equal to the original interpretation semantics of `flip`.

The text interpreter has two states: in interpret state, it performs the interpretation semantics of words it encounters; in compile state, it performs the compilation semantics of these words.

Among other things, `:` switches into compile state, and `;` switches back to interpret state. They contain the factors `]` (switch to compile state) and `[` (switch to interpret state), that do nothing but switch the state.

```
: xxx ( -- )
  [ 5 . ]
;

xxx
see xxx
```

These brackets are also the source of the naming convention mentioned above.

Reference: Section 5.10 [Interpretation and Compilation Semantics], page 85.

3.28 Execution Tokens

`' word` gives you the execution token (XT) of a word. The XT is a cell representing the interpretation semantics of a word. You can execute this semantics with `execute`:

```
' + .s
1 2 rot execute .
```

The XT is similar to a function pointer in C. However, parameter passing through the stack makes it a little more flexible:

```
: map-array ( ... addr u xt -- ... )
\ executes xt ( ... x -- ... ) for every element of the array starting
\ at addr and containing u elements
  { xt }
  cells over + swap ?do
    i @ xt execute
  1 cells +loop ;

create a 3 , 4 , 2 , -1 , 4 ,
a 5 ' . map-array .s
0 a 5 ' + map-array .
s" max-n" environment? drop .s
a 5 ' min map-array .
```

You can use map-array with the XTs of words that consume one element more than they produce. In theory you can also use it with other XTs, but the stack effect then depends on the size of the array, which is hard to understand.

Since XTs are cell-sized, you can store them in memory and manipulate them on the stack like other cells. You can also compile the XT into a word with `compile,`:

```
: foo1 ( n1 n2 -- n )
   [ ' + compile, ] ;
```

```
see foo
```

This is non-standard, because `compile,` has no compilation semantics in the standard, but it works in good Forth systems. For the broken ones, use

```
: [compile,] compile, ; immediate

: foo1 ( n1 n2 -- n )
   [ ' + ] [compile,] ;
see foo
```

`'` is a word with default compilation semantics; it parses the next word when its interpretation semantics are executed, not during compilation:

```
: foo ( -- xt )
  ' ;
see foo
: bar ( ... "word" -- ... )
  ' execute ;
see bar
1 2 bar + .
```

You often want to parse a word during compilation and compile its XT so it will be pushed on the stack at run-time. `[']` does this:

```
: xt-+ ( -- xt )
  ['] + ;
see xt-+
1 2 xt-+ execute .
```

Many programmers tend to see `'` and the word it parses as one unit, and expect it to behave like `[']` when compiled, and are confused by the actual behaviour. If you are, just remember that the Forth system just takes `'` as one unit and has no idea that it is a parsing word (attempts to convenience programmers in this issue have usually resulted in even worse pitfalls, see `State`-smartness—Why it is evil and How to Exorcise it (`http://www.complang.tuwien.ac.at/papers/ertl98.ps.gz`)).

Note that the state of the interpreter does not come into play when creating and executing XTs. I.e., even when you execute `'` in compile state, it still gives you the interpretation semantics. And whatever that state is, `execute` performs the semantics represented by the XT (i.e., for XTs produced with `'` the interpretation semantics).

Reference: Section 5.11 [Tokens for Words], page 87.

3.29 Exceptions

`throw ( n -- )` causes an exception unless n is zero.

```
100 throw .s
0 throw .s
```

`catch ( ... xt -- ... n )` behaves similar to `execute`, but it catches exceptions and pushes the number of the exception on the stack (or 0, if the xt executed without exception). If there was an exception, the stacks have the same depth as when entering `catch`:

```
.s
3 0 ' / catch .s
3 2 ' / catch .s
```

Assignment:

Try the same with `execute` instead of `catch`.

`Throw` always jumps to the dynamically next enclosing `catch`, even if it has to leave several call levels to achieve this:

```
: foo 100 throw ;
: foo1 foo ." after foo" ;
: bar ['] foo1 catch ;
bar .
```

It is often important to restore a value upon leaving a definition, even if the definition is left through an exception. You can ensure this like this:

```
: ...
   save-x
   ['] word-changing-x catch ( ... n )
   restore-x
   ( ... n ) throw ;
```

Gforth provides an alternative syntax in addition to `catch`: `try` ... `recover` ... `endtry`. If the code between `try` and `recover` has an exception, the stack depths are restored, the exception number is pushed on the stack, and the code between `recover` and `endtry` is performed. E.g., the definition for `catch` is

```
: catch ( x1 .. xn xt -- y1 .. ym 0 / z1 .. zn error ) \ exception
  try
    execute 0
  recover
    nip
  endtry ;
```

The equivalent to the restoration code above is

```
: ...
  save-x
  try
    word-changing-x 0
  recover endtry
  restore-x
  throw ;
```

This works if `word-changing-x` does not change the stack depth, otherwise you should add some code between `recover` and `endtry` to balance the stack.

Reference: Section 5.8.6 [Exception Handling], page 70.

3.30 Defining Words

`:`, `create`, and `variable` are definition words: They define other words. `Constant` is another definition word:

```
5 constant foo
foo .
```

You can also use the prefixes `2` (double-cell) and `f` (floating point) with `variable` and `constant`.

You can also define your own defining words. E.g.:

```
: variable ( "name" -- )
  create 0 , ;
```

You can also define defining words that create words that do something other than just producing their address:

```
: constant ( n "name" -- )
  create ,
does> ( -- n )
  ( addr ) @ ;

5 constant foo
foo .
```

The definition of `constant` above ends at the `does>`; i.e., `does>` replaces `;`, but it also does something else: It changes the last defined word such that it pushes the address of the body of the word and then performs the code after the `does>` whenever it is called.

In the example above, `constant` uses `,` to store 5 into the body of `foo`. When `foo` executes, it pushes the address of the body onto the stack, then (in the code after the `does>`) fetches the 5 from there.

The stack comment near the `does>` reflects the stack effect of the defined word, not the stack effect of the code after the `does>` (the difference is that the code expects the address of the body that the stack comment does not show).

You can use these definition words to do factoring in cases that involve (other) definition words. E.g., a field offset is always added to an address. Instead of defining

```
2 cells constant offset-field1
```

and using this like

```
( addr ) offset-field1 +
```

you can define a definition word

```
: simple-field ( n "name" -- )
  create ,
does> ( n1 -- n1+n )
  ( addr ) @ + ;
```

Definition and use of field offsets now look like this:

```
2 cells simple-field field1
create mystruct 4 cells allot
mystruct .s field1 .s drop
```

If you want to do something with the word without performing the code after the `does>`, you can access the body of a `create`d word with `>body ( xt -- addr )`:

```
: value ( n "name" -- )
  create ,
does> ( -- n1 )
  @ ;
: to ( n "name" -- )
  ' >body ! ;

5 value foo
foo .
```

```
7 to foo
foo .
```

Assignment:

Define `defer ( "name" -- )`, which creates a word that stores an XT (at the start the XT of `abort`), and upon execution `execute`s the XT. Define `is ( xt "name" -- )` that stores `xt` into `name`, a word defined with `defer`. Indirect recursion is one application of `defer`.

Reference: Section 5.9.8 [User-defined Defining Words], page 76.

3.31 Arrays and Records

Forth has no standard words for defining data structures such as arrays and records (structs in C terminology), but you can build them yourself based on address arithmetic. You can also define words for defining arrays and records (see Section 3.30 [Defining Words], page 30).

One of the first projects a Forth newcomer sets out upon when learning about defining words is an array defining word (possibly for n-dimensional arrays). Go ahead and do it, I did it, too; you will learn something from it. However, don't be disappointed when you later learn that you have little use for these words (inappropriate use would be even worse). I have not yet found a set of useful array words yet; the needs are just too diverse, and named, global arrays (the result of naive use of defining words) are often not flexible enough (e.g., consider how to pass them as parameters). Another such project is a set of words to help dealing with strings.

On the other hand, there is a useful set of record words, and it has been defined in '`compat/struct.fs`'; these words are predefined in Gforth. They are explained in depth elsewhere in this manual (see see Section 5.21 [Structures], page 128). The `simple-field` example above is simplified variant of fields in this package.

3.32 `POSTPONE`

You can compile the compilation semantics (instead of compiling the interpretation semantics) of a word with `POSTPONE`:

```
: MY-+ ( Compilation: -- ; Run-time of compiled code: n1 n2 -- n )
 POSTPONE + ; immediate
: foo ( n1 n2 -- n )
 MY-+ ;
1 2 foo .
see foo
```

During the definition of `foo` the text interpreter performs the compilation semantics of `MY-+`, which performs the compilation semantics of `+`, i.e., it compiles `+` into `foo`.

This example also displays separate stack comments for the compilation semantics and for the stack effect of the compiled code. For words with default compilation semantics these stack effects are usually not displayed; the stack effect of the compilation semantics is always `( -- )` for these words, the stack effect for the compiled code is the stack effect of the interpretation semantics.

Note that the state of the interpreter does not come into play when performing the compilation semantics in this way. You can also perform it interpretively, e.g.:

```
: foo2 ( n1 n2 -- n )
 [ MY-+ ] ;
1 2 foo .
see foo
```

However, there are some broken Forth systems where this does not always work, and therefore this practice was been declared non-standard in 1999.

Here is another example for using `POSTPONE`:

```
: MY-- ( Compilation: -- ; Run-time of compiled code: n1 n2 -- n )
 POSTPONE negate POSTPONE + ; immediate compile-only
: bar ( n1 n2 -- n )
  MY-- ;
2 1 bar .
see bar
```

You can define `ENDIF` in this way:

```
: ENDIF ( Compilation: orig -- )
  POSTPONE then ; immediate
```

Assignment:
Write `MY-2DUP` that has compilation semantics equivalent to `2dup`, but compiles `over over`.

3.33 Literal

You cannot `POSTPONE` numbers:

```
: [FOO] POSTPONE 500 ; immediate
```

Instead, you can use `LITERAL (compilation: n --; run-time: -- n )`:

```
: [FOO] ( compilation: --; run-time: -- n )
  500 POSTPONE literal ; immediate

: flip [FOO] ;
flip .
see flip
```

`LITERAL` consumes a number at compile-time (when it's compilation semantics are executed) and pushes it at run-time (when the code it compiled is executed). A frequent use of `LITERAL` is to compile a number computed at compile time into the current word:

```
: bar ( -- n )
  [ 2 2 + ] literal ;
see bar
```

Assignment:
Write `]L` which allows writing the example above as `: bar ( -- n ) [ 2 2 + ]L ;`

3.34 Advanced macros

Reconsider `map-array` from Section 3.28 [Execution Tokens], page 28. It frequently performs `execute`, a relatively expensive operation in some Forth implementations. You can use `compile,` and `POSTPONE` to eliminate these `executes` and produce a word that contains the word to be performed directly:

```
: compile-map-array ( compilation: xt -- ; run-time: ... addr u -- ... )
\ at run-time, execute xt ( ... x -- ... ) for each element of the
\ array beginning at addr and containing u elements
  { xt }
  POSTPONE cells POSTPONE over POSTPONE + POSTPONE swap POSTPONE ?do
    POSTPONE i POSTPONE @ xt compile,
  1 cells POSTPONE literal POSTPONE +loop ;

: sum-array ( addr u -- n )
 0 rot rot [ ' + compile-map-array ] ;
see sum-array
a 5 sum-array .
```

You can use the full power of Forth for generating the code; here's an example where the code is generated in a loop:

```
: compile-vmul-step ( compilation: n --; run-time: n1 addr1 -- n2 addr2 )
\ n2=n1+(addr1)*n, addr2=addr1+cell
  POSTPONE tuck POSTPONE @
  POSTPONE literal POSTPONE * POSTPONE +
  POSTPONE swap POSTPONE cell+ ;

: compile-vmul ( compilation: addr1 u -- ; run-time: addr2 -- n )
\ n=v1*v2 (inner product), where the v_i are represented as addr_i u
  0 postpone literal postpone swap
  [ ' compile-vmul-step compile-map-array ]
  postpone drop ;
see compile-vmul

: a-vmul ( addr -- n )
\ n=a*v, where v is a vector that's as long as a and starts at addr
 [ a 5 compile-vmul ] ;
see a-vmul
a a-vmul .
```

This example uses `compile-map-array` to show off, but you could also use `map-array` instead (try it now!).

You can use this technique for efficient multiplication of large matrices. In matrix multiplication, you multiply every line of one matrix with every column of the other matrix. You can generate the code for one line once, and use it for every column. The only downside of this technique is that it is cumbersome to recover the memory consumed by the generated code when you are done (and in more complicated cases it is not possible portably).

3.35 Compilation Tokens

This section is Gforth-specific. You can skip it.

`' word compile,` compiles the interpretation semantics. For words with default compilation semantics this is the same as performing the compilation semantics. To represent the compilation semantics of other words (e.g., words like `if` that have no interpretation semantics), Gforth has the concept of a compilation token (CT, consisting of two cells), and words `comp'` and `[comp']`. You can perform the compilation semantics represented by a CT with `execute`:

```
: foo2 ( n1 n2 -- n )
   [ comp' + execute ] ;
see foo
```

You can compile the compilation semantics represented by a CT with `postpone,`:

```
: foo3 ( -- )
  [ comp' + postpone, ] ;
see foo3
```

`[ comp' word postpone, ]` is equivalent to `POSTPONE word`. `comp'` is particularly useful for words that have no interpretation semantics:

```
' if
comp' if .s 2drop
```

Reference: Section 5.11 [Tokens for Words], page 87.

3.36 Wordlists and Search Order

The dictionary is not just a memory area that allows you to allocate memory with `allot`, it also contains the Forth words, arranged in several wordlists. When searching for a word in a wordlist, conceptually you start searching at the youngest and proceed towards older words (in reality most systems nowadays use hash-tables); i.e., if you define a word with the same name as an older word, the new word shadows the older word.

Which wordlists are searched in which order is determined by the search order. You can display the search order with `order`. It displays first the search order, starting with the wordlist searched first, then it displays the wordlist that will contain newly defined words.

You can create a new, empty wordlist with `wordlist ( -- wid )`:

```
wordlist constant mywords
```

`Set-current ( wid -- )` sets the wordlist that will contain newly defined words (the *current* wordlist):

```
mywords set-current
order
```

Gforth does not display a name for the wordlist in `mywords` because this wordlist was created anonymously with `wordlist`.

You can get the current wordlist with `get-current ( -- wid)`. If you want to put something into a specific wordlist without overall effect on the current wordlist, this typically looks like this:

```
get-current mywords set-current ( wid )
create someword
( wid ) set-current
```

You can write the search order with `set-order ( wid1 .. widn n -- )` and read it with `get-order ( -- wid1 .. widn n )`. The first searched wordlist is topmost.

```
get-order mywords swap 1+ set-order
order
```

Yes, the order of wordlists in the output of `order` is reversed from stack comments and the output of `.s` and thus unintuitive.

Assignment:

Define `>order ( wid -- )` with adds `wid` as first searched wordlist to the search order. Define `previous ( -- )`, which removes the first searched wordlist from the search order. Experiment with boundary conditions (you will see some crashes or situations that are hard or impossible to leave).

The search order is a powerful foundation for providing features similar to Modula-2 modules and C++ namespaces. However, trying to modularize programs in this way has disadvantages for debugging and reuse/factoring that overcome the advantages in my experience (I don't do huge projects, though). These disadvantages are not so clear in other languages/programming environments, because these languages are not so strong in debugging and reuse.

Reference: Section 5.15 [Word Lists], page 100.

4 An Introduction to ANS Forth

The difference of this chapter from the Tutorial (see Chapter 3 [Tutorial], page 9) is that it is slower-paced in its examples, but uses them to dive deep into explaining Forth internals (not covered by the Tutorial). Apart from that, this chapter covers far less material. It is suitable for reading without using a computer.

The primary purpose of this manual is to document Gforth. However, since Forth is not a widely-known language and there is a lack of up-to-date teaching material, it seems worthwhile to provide some introductory material. For other sources of Forth-related information, see Appendix C [Forth-related information], page 206.

The examples in this section should work on any ANS Forth; the output shown was produced using Gforth. Each example attempts to reproduce the exact output that Gforth produces. If you try out the examples (and you should), what you should type is shown *`like this`* and Gforth's response is shown `like this`. The single exception is that, where the example shows ⟨RET⟩ it means that you should press the "carriage return" key. Unfortunately, some output formats for this manual cannot show the difference between *`this`* and `this` which will make trying out the examples harder (but not impossible).

Forth is an unusual language. It provides an interactive development environment which includes both an interpreter and compiler. Forth programming style encourages you to break a problem down into many small fragments (*factoring*), and then to develop and test each fragment interactively. Forth advocates assert that breaking the edit-compile-test cycle used by conventional programming languages can lead to great productivity improvements.

4.1 Introducing the Text Interpreter

When you invoke the Forth image, you will see a startup banner printed and nothing else (if you have Gforth installed on your system, try invoking it now, by typing *`gforth`*⟨RET⟩). Forth is now running its command line interpreter, which is called the *Text Interpreter* (also known as the *Outer Interpreter*). (You will learn a lot about the text interpreter as you read through this chapter, for more detail see Section 5.13 [The Text Interpreter], page 93).

Although it's not obvious, Forth is actually waiting for your input. Type a number and press the ⟨RET⟩ key:

```
45⟨RET⟩  ok
```

Rather than give you a prompt to invite you to input something, the text interpreter prints a status message *after* it has processed a line of input. The status message in this case (" `ok`" followed by carriage-return) indicates that the text interpreter was able to process all of your input successfully. Now type something illegal:

```
qwer341⟨RET⟩
:1: Undefined word
qwer341
^^^^^^^
$400D2BA8 Bounce
$400DBDA8 no.extensions
```

The exact text, other than the "Undefined word" may differ slightly on your system, but the effect is the same; when the text interpreter detects an error, it discards any remaining text on a line, resets certain internal state and prints an error message. For a detailed description of error messages see Chapter 6 [Error messages], page 164.

The text interpreter waits for you to press carriage-return, and then processes your input line. Starting at the beginning of the line, it breaks the line into groups of characters separated by spaces. For each group of characters in turn, it makes two attempts to do something:

- It tries to treat it as a command. It does this by searching a *name dictionary*. If the group of characters matches an entry in the name dictionary, the name dictionary provides the text interpreter with information that allows the text interpreter perform some actions. In Forth jargon, we say that the group of characters names a *word*, that the dictionary search returns an *execution token (xt)* corresponding to the *definition* of the word, and that the text interpreter executes the xt. Often, the terms *word* and *definition* are used interchangeably.
- If the text interpreter fails to find a match in the name dictionary, it tries to treat the group of characters as a number in the current number base (when you start up Forth, the current number base is base 10). If the group of characters legitimately represents a number, the text interpreter pushes the number onto a stack (we'll learn more about that in the next section).

If the text interpreter is unable to do either of these things with any group of characters, it discards the group of characters and the rest of the line, then prints an error message. If the text interpreter reaches the end of the line without error, it prints the status message " `ok`" followed by carriage-return.

This is the simplest command we can give to the text interpreter:

```
⟨RET⟩  ok
```

The text interpreter did everything we asked it to do (nothing) without an error, so it said that everything is " `ok`". Try a slightly longer command:

```
12 dup fred dup⟨RET⟩
:1: Undefined word
12 dup fred dup
       ^^^^
$400D2BA8 Bounce
$400DBDA8 no.extensions
```

When you press the carriage-return key, the text interpreter starts to work its way along the line:

- When it gets to the space after the `2`, it takes the group of characters `12` and looks them up in the name dictionary[1]. There is no match for this group of characters in the name dictionary, so it tries to treat them as a number. It is able to do this successfully, so it puts the number, 12, "on the stack" (whatever that means).
- The text interpreter resumes scanning the line and gets the next group of characters, `dup`. It looks it up in the name dictionary and (you'll have to take my word for this) finds it, and executes the word `dup` (whatever that means).

[1] We can't tell if it found them or not, but assume for now that it did not

- Once again, the text interpreter resumes scanning the line and gets the group of characters `fred`. It looks them up in the name dictionary, but can't find them. It tries to treat them as a number, but they don't represent any legal number.

At this point, the text interpreter gives up and prints an error message. The error message shows exactly how far the text interpreter got in processing the line. In particular, it shows that the text interpreter made no attempt to do anything with the final character group, `dup`, even though we have good reason to believe that the text interpreter would have no problem looking that word up and executing it a second time.

4.2 Stacks, postfix notation and parameter passing

In procedural programming languages (like C and Pascal), the building-block of programs is the *function* or *procedure*. These functions or procedures are called with *explicit parameters*. For example, in C we might write:

```
total = total + new_volume(length,height,depth);
```

where new_volume is a function-call to another piece of code, and total, length, height and depth are all variables. length, height and depth are parameters to the function-call.

In Forth, the equivalent of the function or procedure is the *definition* and parameters are implicitly passed between definitions using a shared stack that is visible to the programmer. Although Forth does support variables, the existence of the stack means that they are used far less often than in most other programming languages. When the text interpreter encounters a number, it will place (*push*) it on the stack. There are several stacks (the actual number is implementation-dependent ...) and the particular stack used for any operation is implied unambiguously by the operation being performed. The stack used for all integer operations is called the *data stack* and, since this is the stack used most commonly, references to "the data stack" are often abbreviated to "the stack".

The stacks have a last-in, first-out (LIFO) organisation. If you type:

1 2 3⟨RET⟩ `ok`

Then this instructs the text interpreter to placed three numbers on the (data) stack. An analogy for the behaviour of the stack is to take a pack of playing cards and deal out the ace (1), 2 and 3 into a pile on the table. The 3 was the last card onto the pile ("last-in") and if you take a card off the pile then, unless you're prepared to fiddle a bit, the card that you take off will be the 3 ("first-out"). The number that will be first-out of the stack is called the *top of stack*, which is often abbreviated to *TOS*.

To understand how parameters are passed in Forth, consider the behaviour of the definition `+` (pronounced "plus"). You will not be surprised to learn that this definition performs addition. More precisely, it adds two number together and produces a result. Where does it get the two numbers from? It takes the top two numbers off the stack. Where does it place the result? On the stack. You can act-out the behaviour of `+` with your playing cards like this:

- Pick up two cards from the stack on the table
- Stare at them intently and ask yourself "what *is* the sum of these two numbers"
- Decide that the answer is 5
- Shuffle the two cards back into the pack and find a 5

- Put a 5 on the remaining ace that's on the table.

If you don't have a pack of cards handy but you do have Forth running, you can use the definition `.s` to show the current state of the stack, without affecting the stack. Type:

```
clearstack 1 2 3⟨RET⟩ ok
.s⟨RET⟩ <3> 1 2 3  ok
```

The text interpreter looks up the word `clearstack` and executes it; it tidies up the stack and removes any entries that may have been left on it by earlier examples. The text interpreter pushes each of the three numbers in turn onto the stack. Finally, the text interpreter looks up the word `.s` and executes it. The effect of executing `.s` is to print the "`<3>`" (the total number of items on the stack) followed by a list of all the items on the stack; the item on the far right-hand side is the TOS.

You can now type:

```
+ .s⟨RET⟩ <2> 1 5  ok
```

which is correct; there are now 2 items on the stack and the result of the addition is 5.

If you're playing with cards, try doing a second addition: pick up the two cards, work out that their sum is 6, shuffle them into the pack, look for a 6 and place that on the table. You now have just one item on the stack. What happens if you try to do a third addition? Pick up the first card, pick up the second card – ah! There is no second card. This is called a *stack underflow* and consitutes an error. If you try to do the same thing with Forth it often reports an error (probably a Stack Underflow or an Invalid Memory Address error).

The opposite situation to a stack underflow is a *stack overflow*, which simply accepts that there is a finite amount of storage space reserved for the stack. To stretch the playing card analogy, if you had enough packs of cards and you piled the cards up on the table, you would eventually be unable to add another card; you'd hit the ceiling. Gforth allows you to set the maximum size of the stacks. In general, the only time that you will get a stack overflow is because a definition has a bug in it and is generating data on the stack uncontrollably.

There's one final use for the playing card analogy. If you model your stack using a pack of playing cards, the maximum number of items on your stack will be 52 (I assume you didn't use the Joker). The maximum *value* of any item on the stack is 13 (the King). In fact, the only possible numbers are positive integer numbers 1 through 13; you can't have (for example) 0 or 27 or 3.52 or -2. If you change the way you think about some of the cards, you can accommodate different numbers. For example, you could think of the Jack as representing 0, the Queen as representing -1 and the King as representing -2. Your *range* remains unchanged (you can still only represent a total of 13 numbers) but the numbers that you can represent are -2 through 10.

In that analogy, the limit was the amount of information that a single stack entry could hold, and Forth has a similar limit. In Forth, the size of a stack entry is called a *cell*. The actual size of a cell is implementation dependent and affects the maximum value that a stack entry can hold. A Standard Forth provides a cell size of at least 16-bits, and most desktop systems use a cell size of 32-bits.

Forth does not do any type checking for you, so you are free to manipulate and combine stack items in any way you wish. A convenient way of treating stack items is as 2's complement signed integers, and that is what Standard words like `+` do. Therefore you can type:

```
-5 12 + .s⟨RET⟩ <1> 7  ok
```

If you use numbers and definitions like + in order to turn Forth into a great big pocket calculator, you will realise that it's rather different from a normal calculator. Rather than typing 2 + 3 = you had to type 2 3 + (ignore the fact that you had to use .s to see the result). The terminology used to describe this difference is to say that your calculator uses *Infix Notation* (parameters and operators are mixed) whilst Forth uses *Postfix Notation* (parameters and operators are separate), also called *Reverse Polish Notation*.

Whilst postfix notation might look confusing to begin with, it has several important advantages:

- it is unambiguous
- it is more concise
- it fits naturally with a stack-based system

To examine these claims in more detail, consider these sums:

```
6 + 5 * 4 =
4 * 5 + 6 =
```

If you're just learning maths or your maths is very rusty, you will probably come up with the answer 44 for the first and 26 for the second. If you are a bit of a whizz at maths you will remember the *convention* that multiplication takes precendence over addition, and you'd come up with the answer 26 both times. To explain the answer 26 to someone who got the answer 44, you'd probably rewrite the first sum like this:

```
6 + (5 * 4) =
```

If what you really wanted was to perform the addition before the multiplication, you would have to use parentheses to force it.

If you did the first two sums on a pocket calculator you would probably get the right answers, unless you were very cautious and entered them using these keystroke sequences:

6 + 5 = * 4 = 4 * 5 = + 6 =

Postfix notation is unambiguous because the order that the operators are applied is always explicit; that also means that parentheses are never required. The operators are *active* (the act of quoting the operator makes the operation occur) which removes the need for "=".

The sum 6 + 5 * 4 can be written (in postfix notation) in two equivalent ways:

```
6 5 4 * +      or:
5 4 * 6 +
```

An important thing that you should notice about this notation is that the *order* of the numbers does not change; if you want to subtract 2 from 10 you type 10 2 -.

The reason that Forth uses postfix notation is very simple to explain: it makes the implementation extremely simple, and it follows naturally from using the stack as a mechanism for passing parameters. Another way of thinking about this is to realise that all Forth definitions are *active*; they execute as they are encountered by the text interpreter. The result of this is that the syntax of Forth is trivially simple.

4.3 Your first Forth definition

Until now, the examples we've seen have been trivial; we've just been using Forth as a bigger-than-pocket calculator. Also, each calculation we've shown has been a "one-off" – to repeat it we'd need to type it in again[2] In this section we'll see how to add new words to Forth's vocabulary.

The easiest way to create a new word is to use a *colon definition*. We'll define a few and try them out before worrying too much about how they work. Try typing in these examples; be careful to copy the spaces accurately:

```
: add-two 2 + . ;
: greet ." Hello and welcome" ;
: demo 5 add-two ;
```

Now try them out:

```
greet⟨RET⟩ Hello and welcome  ok
greet greet⟨RET⟩ Hello and welcomeHello and welcome  ok
4 add-two⟨RET⟩ 6  ok
demo⟨RET⟩ 7  ok
9 greet demo add-two⟨RET⟩ Hello and welcome7 11  ok
```

The first new thing that we've introduced here is the pair of words `:` and `;`. These are used to start and terminate a new definition, respectively. The first word after the `:` is the name for the new definition.

As you can see from the examples, a definition is built up of words that have already been defined; Forth makes no distinction between definitions that existed when you started the system up, and those that you define yourself.

The examples also introduce the words `.` (dot), `."` (dot-quote) and `dup` (dewp). Dot takes the value from the top of the stack and displays it. It's like `.s` except that it only displays the top item of the stack and it is destructive; after it has executed, the number is no longer on the stack. There is always one space printed after the number, and no spaces before it. Dot-quote defines a string (a sequence of characters) that will be printed when the word is executed. The string can contain any printable characters except `"`. A `"` has a special function; it is not a Forth word but it acts as a delimiter (the way that delimiters work is described in the next section). Finally, `dup` duplicates the value at the top of the stack. Try typing `5 dup .s` to see what it does.

We already know that the text interpreter searches through the dictionary to locate names. If you've followed the examples earlier, you will already have a definition called `add-two`. Lets try modifying it by typing in a new definition:

```
: add-two dup . ." + 2 =" 2 + . ;⟨RET⟩ redefined add-two  ok
```

Forth recognised that we were defining a word that already exists, and printed a message to warn us of that fact. Let's try out the new definition:

```
9 add-two⟨RET⟩ 9 + 2 =11  ok
```

All that we've actually done here, though, is to create a new definition, with a particular name. The fact that there was already a definition with the same name did not make

[2] That's not quite true. If you press the up-arrow key on your keyboard you should be able to scroll back to any earlier command, edit it and re-enter it.

any difference to the way that the new definition was created (except that Forth printed a warning message). The old definition of add-two still exists (try `demo` again to see that this is true). Any new definition will use the new definition of `add-two`, but old definitions continue to use the version that already existed at the time that they were `compiled`.

Before you go on to the next section, try defining and redefining some words of your own.

4.4 How does that work?

Now we're going to take another look at the definition of `add-two` from the previous section. From our knowledge of the way that the text interpreter works, we would have expected this result when we tried to define `add-two`:

```
: add-two 2 + . ;⟨RET⟩
  ^^^^^^^
Error: Undefined word
```

The reason that this didn't happen is bound up in the way that `:` works. The word `:` does two special things. The first special thing that it does prevents the text interpreter from ever seeing the characters `add-two`. The text interpreter uses a variable called `>IN` (pronounced "to-in") to keep track of where it is in the input line. When it encounters the word `:` it behaves in exactly the same way as it does for any other word; it looks it up in the name dictionary, finds its xt and executes it. When `:` executes, it looks at the input buffer, finds the word `add-two` and advances the value of `>IN` to point past it. It then does some other stuff associated with creating the new definition (including creating an entry for `add-two` in the name dictionary). When the execution of `:` completes, control returns to the text interpreter, which is oblivious to the fact that it has been tricked into ignoring part of the input line.

Words like `:` – words that advance the value of `>IN` and so prevent the text interpreter from acting on the whole of the input line – are called *parsing words*.

The second special thing that `:` does is change the value of a variable called `state`, which affects the way that the text interpreter behaves. When Gforth starts up, `state` has the value 0, and the text interpreter is said to be *interpreting*. During a colon definition (started with `:`), `state` is set to -1 and the text interpreter is said to be *compiling*.

In this example, the text interpreter is compiling when it processes the string "`2 + . ;`". It still breaks the string down into character sequences in the same way. However, instead of pushing the number `2` onto the stack, it lays down (*compiles*) some magic into the definition of `add-two` that will make the number `2` get pushed onto the stack when `add-two` is *executed*. Similarly, the behaviours of `+` and `.` are also compiled into the definition.

One category of words don't get compiled. These so-called *immediate words* get executed (performed *now*) regardless of whether the text interpreter is interpreting or compiling. The word `;` is an immediate word. Rather than being compiled into the definition, it executes. Its effect is to terminate the current definition, which includes changing the value of `state` back to 0.

When you execute `add-two`, it has a *run-time effect* that is exactly the same as if you had typed `2 + .` ⟨RET⟩ outside of a definition.

In Forth, every word or number can be described in terms of two properties:

- Its *interpretation semantics* describe how it will behave when the text interpreter encounters it in *interpret* state. The interpretation semantics of a word are represented by an *execution token*.
- Its *compilation semantics* describe how it will behave when the text interpreter encounters it in *compile* state. The compilation semantics of a word are represented in an implementation-dependent way; Gforth uses a *compilation token*.

Numbers are always treated in a fixed way:

- When the number is *interpreted*, its behaviour is to push the number onto the stack.
- When the number is *compiled*, a piece of code is appended to the current definition that pushes the number when it runs. (In other words, the compilation semantics of a number are to postpone its interpretation semantics until the run-time of the definition that it is being compiled into.)

Words don't behave in such a regular way, but most have *default semantics* which means that they behave like this:

- The *interpretation semantics* of the word are to do something useful.
- The *compilation semantics* of the word are to append its *interpretation semantics* to the current definition (so that its run-time behaviour is to do something useful).

The actual behaviour of any particular word can be controlled by using the words `immediate` and `compile-only` when the word is defined. These words set flags in the name dictionary entry of the most recently defined word, and these flags are retrieved by the text interpreter when it finds the word in the name dictionary.

A word that is marked as *immediate* has compilation semantics that are identical to its interpretation semantics. In other words, it behaves like this:

- The *interpretation semantics* of the word are to do something useful.
- The *compilation semantics* of the word are to do something useful (and actually the same thing); i.e., it is executed during compilation.

Marking a word as *compile-only* prohibits the text interpreter from performing the interpretation semantics of the word directly; an attempt to do so will generate an error. It is never necessary to use `compile-only` (and it is not even part of ANS Forth, though it is provided by many implementations) but it is good etiquette to apply it to a word that will not behave correctly (and might have unexpected side-effects) in interpret state. For example, it is only legal to use the conditional word `IF` within a definition. If you forget this and try to use it elsewhere, the fact that (in Gforth) it is marked as `compile-only` allows the text interpreter to generate a helpful error message rather than subjecting you to the consequences of your folly.

This example shows the difference between an immediate and a non-immediate word:

```
: show-state state @ . ;
: show-state-now show-state ; immediate
: word1 show-state ;
: word2 show-state-now ;
```

The word `immediate` after the definition of `show-state-now` makes that word an immediate word. These definitions introduce a new word: `@` (pronounced "fetch"). This word fetches the value of a variable, and leaves it on the stack. Therefore, the behaviour of `show-state` is to print a number that represents the current value of `state`.

When you execute `word1`, it prints the number 0, indicating that the system is interpreting. When the text interpreter compiled the definition of `word1`, it encountered `show-state` whose compilation semantics are to append its interpretation semantics to the current definition. When you execute `word1`, it performs the interpretation semantics of `show-state`. At the time that `word1` (and therefore `show-state`) are executed, the system is interpreting.

When you pressed ⟨RET⟩ after entering the definition of `word2`, you should have seen the number -1 printed, followed by " `ok`". When the text interpreter compiled the definition of `word2`, it encountered `show-state-now`, an immediate word, whose compilation semantics are therefore to perform its interpretation semantics. It is executed straight away (even before the text interpreter has moved on to process another group of characters; the `;` in this example). The effect of executing it are to display the value of `state` *at the time that the definition of* `word2` *is being defined*. Printing -1 demonstrates that the system is compiling at this time. If you execute `word2` it does nothing at all.

Before leaving the subject of immediate words, consider the behaviour of `."` in the definition of `greet`, in the previous section. This word is both a parsing word and an immediate word. Notice that there is a space between `."` and the start of the text `Hello and welcome`, but that there is no space between the last letter of `welcome` and the `"` character. The reason for this is that `."` is a Forth word; it must have a space after it so that the text interpreter can identify it. The `"` is not a Forth word; it is a *delimiter*. The examples earlier show that, when the string is displayed, there is neither a space before the `H` nor after the `e`. Since `."` is an immediate word, it executes at the time that `greet` is defined. When it executes, its behaviour is to search forward in the input line looking for the delimiter. When it finds the delimiter, it updates `>IN` to point past the delimiter. It also compiles some magic code into the definition of `greet`; the xt of a run-time routine that prints a text string. It compiles the string `Hello and welcome` into memory so that it is available to be printed later. When the text interpreter gains control, the next word it finds in the input stream is `;` and so it terminates the definition of `greet`.

4.5 Forth is written in Forth

When you start up a Forth compiler, a large number of definitions already exist. In Forth, you develop a new application using bottom-up programming techniques to create new definitions that are defined in terms of existing definitions. As you create each definition you can test and debug it interactively.

If you have tried out the examples in this section, you will probably have typed them in by hand; when you leave Gforth, your definitions will be lost. You can avoid this by using a text editor to enter Forth source code into a file, and then loading code from the file using `include` (see Section 5.17.1 [Forth source files], page 106). A Forth source file is processed by the text interpreter, just as though you had typed it in by hand[3].

Gforth also supports the traditional Forth alternative to using text files for program entry (see Section 5.18 [Blocks], page 109).

[3] Actually, there are some subtle differences – see Section 5.13 [The Text Interpreter], page 93.

In common with many, if not most, Forth compilers, most of Gforth is actually written in Forth. All of the '`.fs`' files in the installation directory[4] are Forth source files, which you can study to see examples of Forth programming.

Gforth maintains a history file that records every line that you type to the text interpreter. This file is preserved between sessions, and is used to provide a command-line recall facility. If you enter long definitions by hand, you can use a text editor to paste them out of the history file into a Forth source file for reuse at a later time (for more information see Section 2.3 [Command-line editing], page 5).

4.6 Review - elements of a Forth system

To summarise this chapter:

- Forth programs use *factoring* to break a problem down into small fragments called *words* or *definitions*.
- Forth program development is an interactive process.
- The main command loop that accepts input, and controls both interpretation and compilation, is called the *text interpreter* (also known as the *outer interpreter*).
- Forth has a very simple syntax, consisting of words and numbers separated by spaces or carriage-return characters. Any additional syntax is imposed by *parsing words*.
- Forth uses a stack to pass parameters between words. As a result, it uses postfix notation.
- To use a word that has previously been defined, the text interpreter searches for the word in the *name dictionary*.
- Words have *interpretation semantics* and *compilation semantics*.
- The text interpreter uses the value of `state` to select between the use of the *interpretation semantics* and the *compilation semantics* of a word that it encounters.
- The relationship between the *interpretation semantics* and *compilation semantics* for a word depend upon the way in which the word was defined (for example, whether it is an *immediate* word).
- Forth definitions can be implemented in Forth (called *high-level definitions*) or in some other way (usually a lower-level language and as a result often called *low-level definitions*, *code definitions* or *primitives*).
- Many Forth systems are implemented mainly in Forth.

4.7 Where To Go Next

Amazing as it may seem, if you have read (and understood) this far, you know almost all the fundamentals about the inner workings of a Forth system. You certainly know enough to be able to read and understand the rest of this manual and the ANS Forth document, to learn more about the facilities that Forth in general and Gforth in particular provide. Even scarier, you know almost enough to implement your own Forth system. However, that's not a good idea just yet... better to try writing some programs in Gforth.

---

[4] For example, '`/usr/local/share/gforth...`'

Forth has such a rich vocabulary that it can be hard to know where to start in learning it. This section suggests a few sets of words that are enough to write small but useful programs. Use the word index in this document to learn more about each word, then try it out and try to write small definitions using it. Start by experimenting with these words:

- Arithmetic: `+ - * / /MOD */ ABS INVERT`
- Comparison: `MIN MAX =`
- Logic: `AND OR XOR NOT`
- Stack manipulation: `DUP DROP SWAP OVER`
- Loops and decisions: `IF ELSE ENDIF ?DO I LOOP`
- Input/Output: `. ." EMIT CR KEY`
- Defining words: `: ; CREATE`
- Memory allocation words: `ALLOT ,`
- Tools: `SEE WORDS .S MARKER`

When you have mastered those, go on to:

- More defining words: `VARIABLE CONSTANT VALUE TO CREATE DOES>`
- Memory access: `@ !`

When you have mastered these, there's nothing for it but to read through the whole of this manual and find out what you've missed.

4.8 Exercises

TODO: provide a set of programming excercises linked into the stuff done already and into other sections of the manual. Provide solutions to all the exercises in a .fs file in the distribution.

5 Forth Words

5.1 Notation

The Forth words are described in this section in the glossary notation that has become a de-facto standard for Forth texts:

word *Stack effect* *wordset* *pronunciation*

Description

word The name of the word.

Stack effect

The stack effect is written in the notation *before* `--` *after*, where *before* and *after* describe the top of stack entries before and after the execution of the word. The rest of the stack is not touched by the word. The top of stack is rightmost, i.e., a stack sequence is written as it is typed in. Note that Gforth uses a separate floating point stack, but a unified stack notation. Also, return stack effects are not shown in *stack effect*, but in *Description*. The name of a stack item describes the type and/or the function of the item. See below for a discussion of the types.

All words have two stack effects: A compile-time stack effect and a run-time stack effect. The compile-time stack-effect of most words is – . If the compile-time stack-effect of a word deviates from this standard behaviour, or the word does other unusual things at compile time, both stack effects are shown; otherwise only the run-time stack effect is shown.

pronunciation

How the word is pronounced.

wordset The ANS Forth standard is divided into several word sets. A standard system need not support all of them. Therefore, in theory, the fewer word sets your program uses the more portable it will be. However, we suspect that most ANS Forth systems on personal machines will feature all word sets. Words that are not defined in ANS Forth have `gforth` or `gforth-internal` as word set. `gforth` describes words that will work in future releases of Gforth; `gforth-internal` words are more volatile. Environmental query strings are also displayed like words; you can recognize them by the `environment` in the word set field.

Description

A description of the behaviour of the word.

The type of a stack item is specified by the character(s) the name starts with:

`f` Boolean flags, i.e. `false` or `true`.

`c` Char

`w` Cell, can contain an integer or an address

`n` signed integer

`u` unsigned integer

`d` double sized signed integer

`ud` double sized unsigned integer

`r` Float (on the FP stack)

`a-` Cell-aligned address

`c-` Char-aligned address (note that a Char may have two bytes in Windows NT)

`f-` Float-aligned address

`df-` Address aligned for IEEE double precision float

`sf-` Address aligned for IEEE single precision float

`xt` Execution token, same size as Cell

`wid` Word list ID, same size as Cell

`ior, wior` I/O result code, cell-sized. In Gforth, you can `throw` iors.

`f83name` Pointer to a name structure

`"` string in the input stream (not on the stack). The terminating character is a blank by default. If it is not a blank, it is shown in `<>` quotes.

5.2 Case insensitivity

Gforth is case-insensitive; you can enter definitions and invoke Standard words using upper, lower or mixed case (however, see Section 8.1.1 [Implementation-defined options], page 167).

ANS Forth only *requires* implementations to recognise Standard words when they are typed entirely in upper case. Therefore, a Standard program must use upper case for all Standard words. You can use whatever case you like for words that you define, but in a Standard program you have to use the words in the same case that you defined them.

Gforth supports case sensitivity through `table`s (case-sensitive wordlists, see Section 5.15 [Word Lists], page 100).

Two people have asked how to convert Gforth to be case-sensitive; while we think this is a bad idea, you can change all wordlists into tables like this:

```
' table-find forth-wordlist wordlist-map !
```

Note that you now have to type the predefined words in the same case that we defined them, which are varying. You may want to convert them to your favourite case before doing this operation (I won't explain how, because if you are even contemplating doing this, you'd better have enough knowledge of Forth systems to know this already).

5.3 Comments

Forth supports two styles of comment; the traditional *in-line* comment, `(` and its modern cousin, the *comment to end of line*; `\`.

`(` *compilation 'ccc<close-paren>' – ; run-time –* core,file "paren"

Comment, usually till the next `)`: parse and discard all subsequent characters in the parse area until ")" is encountered. During interactive input, an end-of-line also acts as a comment terminator. For file input, it does not; if the end-of-file is encountered whilst parsing for the ")" delimiter, Gforth will generate a warning.

`\` *compilation 'ccc<newline>' – ; run-time –* core-ext,block-ext "backslash"

Comment till the end of the line if `BLK` contains 0 (i.e., while not loading a block), parse and discard the remainder of the parse area. Otherwise, parse and discard all subsequent characters in the parse area corresponding to the current line.

`\G` *compilation 'ccc<newline>' – ; run-time –* gforth "backslash-gee"

Equivalent to `\` but used as a tag to annotate definition comments into documentation.

5.4 Boolean Flags

A Boolean flag is cell-sized. A cell with all bits clear represents the flag `false` and a flag with all bits set represents the flag `true`. Words that check a flag (for example, `IF`) will treat a cell that has *any* bit set as `true`.

`true` *– f* core-ext "true"

`Constant` *– f* is a cell with all bits set.

`false` *– f* core-ext "false"

`Constant` *– f* is a cell with all bits clear.

`on` *a-addr –* gforth "on"

Set the (value of the) variable at *a-addr* to `true`.

`off` *a-addr –* gforth "off"

Set the (value of the) variable at *a-addr* to `false`.

5.5 Arithmetic

Forth arithmetic is not checked, i.e., you will not hear about integer overflow on addition or multiplication, you may hear about division by zero if you are lucky. The operator is written after the operands, but the operands are still in the original order. I.e., the infix `2-1` corresponds to `2 1 -`. Forth offers a variety of division operators. If you perform division with potentially negative operands, you do not want to use `/` or `/mod` with its undefined behaviour, but rather `fm/mod` or `sm/mod` (probably the former, see Section 5.5.5 [Mixed precision], page 53).

5.5.1 Single precision

By default, numbers in Forth are single-precision integers that are one cell in size. They can be signed or unsigned, depending upon how you treat them. For the rules used by the text interpreter for recognising single-precision integers see Section 5.13.2 [Number Conversion], page 96.

These words are all defined for signed operands, but some of them also work for unsigned numbers: `+`, `1+`, `-`, `1-`, `*`.

`+` *n1 n2 – n* core "plus"

`1+` *n1 – n2* core "one-plus"

`-` *n1 n2 – n* core "minus"

`1-` *n1 – n2* core "one-minus"

`*` *n1 n2 – n* core "star"

`/` *n1 n2 – n* core "slash"

`mod` *n1 n2 – n* core "mod"

`/mod` *n1 n2 – n3 n4* core "slash-mod"

`negate` *n1 – n2* core "negate"

`abs` *n – u* core "abs"

`min` *n1 n2 – n* core "min"

`max` *n1 n2 – n* core "max"

`FLOORED` *– f* environment "FLOORED"

True if `/` etc. perform floored division

5.5.2 Double precision

For the rules used by the text interpreter for recognising double-precision integers, see Section 5.13.2 [Number Conversion], page 96.

A double precision number is represented by a cell pair, with the most significant cell at the TOS. It is trivial to convert an unsigned single to a double: simply push a `0` onto the TOS. Since numbers are represented by Gforth using 2's complement arithmetic, converting a signed single to a (signed) double requires sign-extension across the most significant cell. This can be achieved using `s>d`. The moral of the story is that you cannot convert a number without knowing whether it represents an unsigned or a signed number.

These words are all defined for signed operands, but some of them also work for unsigned numbers: `d+`, `d-`.

`s>d` *n – d* core "s-to-d"

`d>s` *d – n* double "d-to-s"

`d+` *d1 d2 – d* double "d-plus"

`d-` *d1 d2 – d* double "d-minus"

`dnegate` *d1 – d2* double "d-negate"

`dabs` *d – ud* double "d-abs"

`dmin` *d1 d2 – d* double "d-min"

`dmax` *d1 d2 – d* double "d-max"

5.5.3 Bitwise operations

`and` *w1 w2* – *w* core "and"

`or` *w1 w2* – *w* core "or"

`xor` *w1 w2* – *w* core "x-or"

`invert` *w1* – *w2* core "invert"

`lshift` *u1 n* – *u2* core "l-shift"

`rshift` *u1 n* – *u2* core "r-shift"

Logical shift right by *n* bits.

`2*` *n1* – *n2* core "two-star"

Shift left by 1; also works on unsigned numbers

`d2*` *d1* – *d2* double "d-two-star"

Shift left by 1; also works on unsigned numbers

`2/` *n1* – *n2* core "two-slash"

Arithmetic shift right by 1. For signed numbers this is a floored division by 2 (note that `/` not necessarily floors).

`d2/` *d1* – *d2* double "d-two-slash"

Arithmetic shift right by 1. For signed numbers this is a floored division by 2.

5.5.4 Numeric comparison

Note that the words that compare for equality (`= <> 0= 0<> d= d<> d0= d0<>`) work for for both signed and unsigned numbers.

`<` *n1 n2* – *f* core "less-than"

`<=` *n1 n2* – *f* gforth "less-or-equal"

`<>` *n1 n2* – *f* core-ext "not-equals"

`=` *n1 n2* – *f* core "equals"

`>` *n1 n2* – *f* core "greater-than"

`>=` *n1 n2* – *f* gforth "greater-or-equal"

`0<` *n* – *f* core "zero-less-than"

`0<=` *n* – *f* gforth "zero-less-or-equal"

`0<>` *n* – *f* core-ext "zero-not-equals"

`0=` *n* – *f* core "zero-equals"

`0>` *n* – *f* core-ext "zero-greater-than"

`0>=` *n* – *f* gforth "zero-greater-or-equal"

`u<` *u1 u2* – *f* core "u-less-than"

`u<=` *u1 u2* – *f* gforth "u-less-or-equal"

`u>` *u1 u2* – *f* core-ext "u-greater-than"

`u>=` *u1 u2* – *f* gforth "u-greater-or-equal" `within` *u1 u2 u3 – f* core-ext "within"

u2=<u1<u3 or: u3=<u2 and u1 is not in [u3,u2). This works for unsigned and signed numbers (but not a mixture). Another way to think about this word is to consider the numbers as a circle (wrapping around from `max-u` to 0 for unsigned, and from `max-n` to min-n for signed numbers); now consider the range from u2 towards increasing numbers up to and excluding u3 (giving an empty range if u2=u3); if u1 is in this range, `within` returns true.

`d<` *d1 d2 – f* double "d-less-than"

`d<=` *d1 d2 – f* gforth "d-less-or-equal"

`d<>` *d1 d2 – f* gforth "d-not-equals"

`d=` *d1 d2 – f* double "d-equals"

`d>` *d1 d2 – f* gforth "d-greater-than"

`d>=` *d1 d2 – f* gforth "d-greater-or-equal"

`d0<` *d – f* double "d-zero-less-than"

`d0<=` *d – f* gforth "d-zero-less-or-equal"

`d0<>` *d – f* gforth "d-zero-not-equals"

`d0=` *d – f* double "d-zero-equals"

`d0>` *d – f* gforth "d-zero-greater-than"

`d0>=` *d – f* gforth "d-zero-greater-or-equal"

`du<` *ud1 ud2 – f* double-ext "d-u-less-than"

`du<=` *ud1 ud2 – f* gforth "d-u-less-or-equal"

`du>` *ud1 ud2 – f* gforth "d-u-greater-than"

`du>=` *ud1 ud2 – f* gforth "d-u-greater-or-equal"

5.5.5 Mixed precision

`m+` *d1 n – d2* double "m-plus"

`*/` *n1 n2 n3 – n4* core "star-slash"

n4=(n1*n2)/n3, with the intermediate result being double.

`*/mod` *n1 n2 n3 – n4 n5* core "star-slash-mod"

n1*n2=n3*n5+n4, with the intermediate result (n1*n2) being double.

`m*` *n1 n2 – d* core "m-star"

`um*` *u1 u2 – ud* core "u-m-star"

`m*/` *d1 n2 u3 – dquot* double "m-star-slash"

dquot=(d1*n2)/u3, with the intermediate result being triple-precision. In ANS Forth u3 can only be a positive signed number.

`um/mod` *ud u1 – u2 u3* core "u-m-slash-mod"

ud=u3*u1+u2, u1>u2>=0

`fm/mod` *d1 n1 – n2 n3* core "f-m-slash-mod"

Floored division: $d1 = n3*n1+n2$, $n1>n2>=0$ or $0>=n2>n1$.

`sm/rem` *d1 n1 – n2 n3* core "s-m-slash-rem"

Symmetric division: $d1 = n3*n1+n2$, sign($n2$)=sign($d1$) or 0.

5.5.6 Floating Point

For the rules used by the text interpreter for recognising floating-point numbers see Section 5.13.2 [Number Conversion], page 96.

Gforth has a separate floating point stack, but the documentation uses the unified notation.[1]

Floating point numbers have a number of unpleasant surprises for the unwary (e.g., floating point addition is not associative) and even a few for the wary. You should not use them unless you know what you are doing or you don't care that the results you get are totally bogus. If you want to learn about the problems of floating point numbers (and how to avoid them), you might start with *David Goldberg, What Every Computer Scientist Should Know About Floating-Point Arithmetic (*`http://www.validgh.com/goldberg/paper.ps`*), ACM Computing Surveys 23(1):5−48, March 1991.*

`d>f` *d – r* float "d-to-f"

`f>d` *r – d* float "f-to-d"

`f+` *r1 r2 – r3* float "f-plus"

`f-` *r1 r2 – r3* float "f-minus"

`f*` *r1 r2 – r3* float "f-star"

`f/` *r1 r2 – r3* float "f-slash"

`fnegate` *r1 – r2* float "f-negate"

`fabs` *r1 – r2* float-ext "f-abs"

`fmax` *r1 r2 – r3* float "f-max"

`fmin` *r1 r2 – r3* float "f-min"

`floor` *r1 – r2* float "floor"

Round towards the next smaller integral value, i.e., round toward negative infinity.

`fround` *r1 – r2* gforth "f-round"

Round to the nearest integral value.

`f**` *r1 r2 – r3* float-ext "f-star-star"

r3 is *r1* raised to the *r2*th power.

`fsqrt` *r1 – r2* float-ext "f-square-root"

`fexp` *r1 – r2* float-ext "f-e-x-p"

`fexpm1` *r1 – r2* float-ext "f-e-x-p-m-one"

$r2 = e^{**} r1 - 1$

`fln` *r1 – r2* float-ext "f-l-n"

`flnp1` *r1 – r2* float-ext "f-l-n-p-one"

$r2 = \ln(r1+1)$

`flog` *r1 – r2* float-ext "f-log"

The decimal logarithm.

---

[1] It's easy to generate the separate notation from that by just separating the floating-point numbers out: e.g. `( n r1 u r2 -- r3 )` becomes `( n u -- ) ( F: r1 r2 -- r3 )`.

`falog` *r1* – *r2* float-ext "f-a-log"

r2=10***r1*

`f2*` *r1* – *r2* gforth "f2*"

Multiply *r1* by 2.0e0

`f2/` *r1* – *r2* gforth "f2/"

Multiply *r1* by 0.5e0

`1/f` *r1* – *r2* gforth "1/f"

Divide 1.0e0 by *r1*.

`precision` – *u* float-ext "precision"

u is the number of significant digits currently used by `F.` `FE.` and `FS.`

`set-precision` *u* – float-ext "set-precision"

Set the number of significant digits currently used by `F.` `FE.` and `FS.` to *u*.

Angles in floating point operations are given in radians (a full circle has 2 pi radians).

`fsin` *r1* – *r2* float-ext "f-sine"

`fcos` *r1* – *r2* float-ext "f-cos"

`fsincos` *r1* – *r2 r3* float-ext "f-sine-cos"

r2=sin(*r1*), *r3*=cos(*r1*)

`ftan` *r1* – *r2* float-ext "f-tan"

`fasin` *r1* – *r2* float-ext "f-a-sine"

`facos` *r1* – *r2* float-ext "f-a-cos"

`fatan` *r1* – *r2* float-ext "f-a-tan"

`fatan2` *r1 r2* – *r3* float-ext "f-a-tan-two"

r1/*r2*=tan(*r3*). ANS Forth does not require, but probably intends this to be the inverse of `fsincos`. In gforth it is.

`fsinh` *r1* – *r2* float-ext "f-cinch"

`fcosh` *r1* – *r2* float-ext "f-cosh"

`ftanh` *r1* – *r2* float-ext "f-tan-h"

`fasinh` *r1* – *r2* float-ext "f-a-cinch"

`facosh` *r1* – *r2* float-ext "f-a-cosh"

`fatanh` *r1* – *r2* float-ext "f-a-tan-h"

`pi` – *r* gforth "pi"

`Fconstant` – *r* is the value pi; the ratio of a circle's area to its diameter.

One particular problem with floating-point arithmetic is that comparison for equality often fails when you would expect it to succeed. For this reason approximate equality is often preferred (but you still have to know what you are doing). Also note that IEEE NaNs may compare differently from what you might expect. The comparison words are:

`f~rel` *r1 r2 r3* – *flag* gforth "f~rel"

Approximate equality with relative error: |r1-r2|<r3*|r1+r2|.

`f~abs` *r1 r2 r3* – *flag* gforth "f~abs"

Approximate equality with absolute error: |r1-r2|<r3.

`f~` *r1 r2 r3 – flag* float-ext “f-proximate”

ANS Forth medley for comparing r1 and r2 for equality: r3>0: `f~abs`; r3=0: bitwise comparison; r3<0: `fnegate f~rel`.

`f=` *r1 r2 – f* gforth “f-equals”

`f<>` *r1 r2 – f* gforth “f-not-equals”

`f<` *r1 r2 – f* float “f-less-than”

`f<=` *r1 r2 – f* gforth “f-less-or-equal”

`f>` *r1 r2 – f* gforth “f-greater-than”

`f>=` *r1 r2 – f* gforth “f-greater-or-equal”

`f0<` *r – f* float “f-zero-less-than”

`f0<=` *r – f* gforth “f-zero-less-or-equal”

`f0<>` *r – f* gforth “f-zero-not-equals”

`f0=` *r – f* float “f-zero-equals”

`f0>` *r – f* gforth “f-zero-greater-than”

`f0>=` *r – f* gforth “f-zero-greater-or-equal”

5.6 Stack Manipulation

Gforth maintains a number of separate stacks:

- A data stack (also known as the *parameter stack*) – for characters, cells, addresses, and double cells.
- A floating point stack – for holding floating point (FP) numbers.
- A return stack – for holding the return addresses of colon definitions and other (non-FP) data.
- A locals stack – for holding local variables.

5.6.1 Data stack

`drop` *w –* core “drop”

`nip` *w1 w2 – w2* core-ext “nip”

`dup` *w – w w* core “dupe”

`over` *w1 w2 – w1 w2 w1* core “over”

`tuck` *w1 w2 – w2 w1 w2* core-ext “tuck”

`swap` *w1 w2 – w2 w1* core “swap”

`pick` *u – w* core-ext “pick”

Actually the stack effect is `x0 ... xu u -- x0 ... xu x0` .

`rot` *w1 w2 w3 – w2 w3 w1* core “rote”

`-rot` *w1 w2 w3 – w3 w1 w2* gforth “not-rote”

`?dup` *w – w* core “question-dupe”

Actually the stack effect is: `( w -- 0 | w w )`. It performs a `dup` if w is nonzero.

`roll` *x0 x1 .. xn n – x1 .. xn x0* core-ext “roll”

`2drop` *w1 w2 –* core “two-drop”

`2nip` *w1 w2 w3 w4 – w3 w4* gforth "two-nip"

`2dup` *w1 w2 – w1 w2 w1 w2* core "two-dupe"

`2over` *w1 w2 w3 w4 – w1 w2 w3 w4 w1 w2* core "two-over"

`2tuck` *w1 w2 w3 w4 – w3 w4 w1 w2 w3 w4* gforth "two-tuck"

`2swap` *w1 w2 w3 w4 – w3 w4 w1 w2* core "two-swap"

`2rot` *w1 w2 w3 w4 w5 w6 – w3 w4 w5 w6 w1 w2* double-ext "two-rote"

5.6.2 Floating point stack

Whilst every sane Forth has a separate floating-point stack, it is not strictly required; an ANS Forth system could theoretically keep floating-point numbers on the data stack. As an additional difficulty, you don't know how many cells a floating-point number takes. It is reportedly possible to write words in a way that they work also for a unified stack model, but we do not recommend trying it. Instead, just say that your program has an environmental dependency on a separate floating-point stack.

`floating-stack` *– n* environment "floating-stack"

n is non-zero, showing that Gforth maintains a separate floating-point stack of depth *n*.

`fdrop` *r –* float "f-drop"

`fnip` *r1 r2 – r2* gforth "f-nip"

`fdup` *r – r r* float "f-dupe"

`fover` *r1 r2 – r1 r2 r1* float "f-over"

`ftuck` *r1 r2 – r2 r1 r2* gforth "f-tuck"

`fswap` *r1 r2 – r2 r1* float "f-swap"

`fpick` *u – r* gforth "fpick"

Actually the stack effect is `r0 ... ru u -- r0 ... ru r0` .

`frot` *r1 r2 r3 – r2 r3 r1* float "f-rote"

5.6.3 Return stack

A Forth system is allowed to keep local variables on the return stack. This is reasonable, as local variables usually eliminate the need to use the return stack explicitly. So, if you want to produce a standard compliant program and you are using local variables in a word, forget about return stack manipulations in that word (refer to the standard document for the exact rules).

`>r` *w – R:w* core "to-r"

`r>` *R:w – w* core "r-from"

`r@` *– w ; R: w – w* core "r-fetch"

`rdrop` *R:w –* gforth "rdrop"

`2>r` *w1 w2 – R:w1 R:w2* core-ext "two-to-r"

`2r>` *R:w1 R:w2 – w1 w2* core-ext "two-r-from"

`2r@` *R:w1 R:w2 – R:w1 R:w2 w1 w2* core-ext "two-r-fetch"

`2rdrop` *R:w1 R:w2 –* gforth "two-r-drop"

5.6.4 Locals stack

Gforth uses an extra locals stack. It is described, along with the reasons for its existence, in Section 5.20.1.4 [Locals implementation], page 125.

5.6.5 Stack pointer manipulation

`sp0` – *a-addr* gforth "sp0"

`User` variable – initial value of the data stack pointer.

`sp@` – *a-addr* gforth "sp-fetch"

`sp!` *a-addr* – gforth "sp-store"

`fp0` – *a-addr* gforth "fp0"

`User` variable – initial value of the floating-point stack pointer.

`fp@` – *f-addr* gforth "fp-fetch"

`fp!` *f-addr* – gforth "fp-store"

`rp0` – *a-addr* gforth "rp0"

`User` variable – initial value of the return stack pointer.

`rp@` – *a-addr* gforth "rp-fetch"

`rp!` *a-addr* – gforth "rp-store"

`lp0` – *a-addr* gforth "lp0"

`User` variable – initial value of the locals stack pointer.

`lp@` – *addr* gforth "lp-fetch"

`lp!` *c-addr* – gforth "lp-store"

5.7 Memory

In addition to the standard Forth memory allocation words, there is also a garbage collector (`http://www.complang.tuwien.ac.at/forth/garbage-collection.zip`).

5.7.1 ANS Forth and Gforth memory models

ANS Forth considers a Forth system as consisting of several address spaces, of which only *data space* is managed and accessible with the memory words. Memory not necessarily in data space includes the stacks, the code (called code space) and the headers (called name space). In Gforth everything is in data space, but the code for the primitives is usually read-only.

Data space is divided into a number of areas: The (data space portion of the) dictionary[2], the heap, and a number of system-allocated buffers.

In ANS Forth data space is also divided into contiguous regions. You can only use address arithmetic within a contiguous region, not between them. Usually each allocation gives you one contiguous region, but the dictionary allocation words have additional rules (see Section 5.7.2 [Dictionary allocation], page 59).

---

[2] Sometimes, the term *dictionary* is used to refer to the search data structure embodied in word lists and headers, because it is used for looking up names, just as you would in a conventional dictionary.

Gforth provides one big address space, and address arithmetic can be performed between any addresses. However, in the dictionary headers or code are interleaved with data, so almost the only contiguous data space regions there are those described by ANS Forth as contiguous; but you can be sure that the dictionary is allocated towards increasing addresses even between contiguous regions. The memory order of allocations in the heap is platform-dependent (and possibly different from one run to the next).

5.7.2 Dictionary allocation

Dictionary allocation is a stack-oriented allocation scheme, i.e., if you want to deallocate X, you also deallocate everything allocated after X.

The allocations using the words below are contiguous and grow the region towards increasing addresses. Other words that allocate dictionary memory of any kind (i.e., defining words including `:noname`) end the contiguous region and start a new one.

In ANS Forth only `create`d words are guaranteed to produce an address that is the start of the following contiguous region. In particular, the cell allocated by `variable` is not guaranteed to be contiguous with following `allot`ed memory.

You can deallocate memory by using `allot` with a negative argument (with some restrictions, see `allot`). For larger deallocations use `marker`.

`here` – *addr* core "here"

Return the address of the next free location in data space.

`unused` – *u* core-ext "unused"

Return the amount of free space remaining (in address units) in the region addressed by `here`.

`allot` *n* – core "allot"

Reserve *n* address units of data space without initialization. *n* is a signed number, passing a negative *n* releases memory. In ANS Forth you can only deallocate memory from the current contiguous region in this way. In Gforth you can deallocate anything in this way but named words. The system does not check this restriction.

`c,` *c* – core "c-comma"

Reserve data space for one char and store *c* in the space.

`f,` *f* – gforth "f,"

Reserve data space for one floating-point number and store *f* in the space.

`,` *w* – core "comma"

Reserve data space for one cell and store *w* in the space.

`2,` *w1 w2* – gforth "2,"

Reserve data space for two cells and store the double *w1 w2* there, *w2* first (lower address).

Memory accesses have to be aligned (see Section 5.7.5 [Address arithmetic], page 61). So of course you should allocate memory in an aligned way, too. I.e., before allocating allocating a cell, `here` must be cell-aligned, etc. The words below align `here` if it is not already. Basically it is only already aligned for a type, if the last allocation was a multiple of the size of this type and if `here` was aligned for this type before.

After freshly `create`ing a word, `here` is `align`ed in ANS Forth (`maxalign`ed in Gforth).

`align` – core "align"

If the data-space pointer is not aligned, reserve enough space to align it.

`falign` – float "f-align"

If the data-space pointer is not float-aligned, reserve enough space to align it.

`sfalign` – float-ext "s-f-align"

If the data-space pointer is not single-float-aligned, reserve enough space to align it.

`dfalign` – float-ext "d-f-align"

If the data-space pointer is not double-float-aligned, reserve enough space to align it.

`maxalign` – gforth "maxalign"

Align data-space pointer for all alignment requirements.

`cfalign` – gforth "cfalign"

Align data-space pointer for code field requirements (i.e., such that the corresponding body is maxaligned).

5.7.3 Heap allocation

Heap allocation supports deallocation of allocated memory in any order. Dictionary allocation is not affected by it (i.e., it does not end a contiguous region). In Gforth, these words are implemented using the standard C library calls malloc(), free() and resize().

The memory region produced by one invocation of `allocate` or `resize` is internally contiguous. There is no contiguity between such a region and any other region (including others allocated from the heap).

`allocate` *u – a-addr wior* memory "allocate"

Allocate *u* address units of contiguous data space. The initial contents of the data space is undefined. If the allocation is successful, *a-addr* is the start address of the allocated region and *wior* is 0. If the allocation fails, *a-addr* is undefined and *wior* is a non-zero I/O result code.

`free` *a-addr – wior* memory "free"

Return the region of data space starting at *a-addr* to the system. The region must originally have been obtained using `allocate` or `resize`. If the operational is successful, *wior* is 0. If the operation fails, *wior* is a non-zero I/O result code.

`resize` *a-addr1 u – a-addr2 wior* memory "resize"

Change the size of the allocated area at *a-addr1* to *u* address units, possibly moving the contents to a different area. *a-addr2* is the address of the resulting area. If the operation is successful, *wior* is 0. If the operation fails, *wior* is a non-zero I/O result code. If *a-addr1* is 0, Gforth's (but not the Standard) `resize` `allocate`s *u* address units.

5.7.4 Memory Access

`@` *a-addr – w* core "fetch"

w is the cell stored at *a_addr*.

`!` *w a-addr –* core "store"

Store *w* into the cell at *a-addr*.

`+!` *n a-addr* – core "plus-store"

Add *n* to the cell at *a-addr*.

`c@` *c-addr* – *c* core "c-fetch"

c is the char stored at *c_addr*.

`c!` *c c-addr* – core "c-store"

Store *c* into the char at *c-addr*.

`2@` *a-addr* – *w1 w2* core "two-fetch"

w2 is the content of the cell stored at *a-addr*, *w1* is the content of the next cell.

`2!` *w1 w2 a-addr* – core "two-store"

Store *w2* into the cell at *c-addr* and *w1* into the next cell.

`f@` *f-addr* – *r* float "f-fetch"

r is the float at address *f-addr*.

`f!` *r f-addr* – float "f-store"

Store *r* into the float at address *f-addr*.

`sf@` *sf-addr* – *r* float-ext "s-f-fetch"

Fetch the single-precision IEEE floating-point value *r* from the address *sf-addr*.

`sf!` *r sf-addr* – float-ext "s-f-store"

Store *r* as single-precision IEEE floating-point value to the address *sf-addr*.

`df@` *df-addr* – *r* float-ext "d-f-fetch"

Fetch the double-precision IEEE floating-point value *r* from the address *df-addr*.

`df!` *r df-addr* – float-ext "d-f-store"

Store *r* as double-precision IEEE floating-point value to the address *df-addr*.

5.7.5 Address arithmetic

Address arithmetic is the foundation on which you can build data structures like arrays, records (see Section 5.21 [Structures], page 128) and objects (see Section 5.22 [Object-oriented Forth], page 132).

ANS Forth does not specify the sizes of the data types. Instead, it offers a number of words for computing sizes and doing address arithmetic. Address arithmetic is performed in terms of address units (aus); on most systems the address unit is one byte. Note that a character may have more than one au, so `chars` is no noop (on platforms where it is a noop, it compiles to nothing).

The basic address arithmetic words are `+` and `-`. E.g., if you have the address of a cell, perform `1 cells +`, and you will have the address of the next cell.

In ANS Forth you can perform address arithmetic only within a contiguous region, i.e., if you have an address into one region, you can only add and subtract such that the result is still within the region; you can only subtract or compare addresses from within the same contiguous region. Reasons: several contiguous regions can be arranged in memory in any way; on segmented systems addresses may have unusual representations, such that address arithmetic only works within a region. Gforth provides a few more guarantees (linear address space, dictionary grows upwards), but in general I have found it easy to stay within contiguous regions (exception: computing and comparing to the address just beyond the end of an array).

ANS Forth also defines words for aligning addresses for specific types. Many computers require that accesses to specific data types must only occur at specific addresses; e.g., that cells may only be accessed at addresses divisible by 4. Even if a machine allows unaligned accesses, it can usually perform aligned accesses faster.

For the performance-conscious: alignment operations are usually only necessary during the definition of a data structure, not during the (more frequent) accesses to it.

ANS Forth defines no words for character-aligning addresses. This is not an oversight, but reflects the fact that addresses that are not char-aligned have no use in the standard and therefore will not be created.

ANS Forth guarantees that addresses returned by `CREATE`d words are cell-aligned; in addition, Gforth guarantees that these addresses are aligned for all purposes.

Note that the ANS Forth word `char` has nothing to do with address arithmetic.

`chars` *n1 – n2* core "chars"

n2 is the number of address units of *n1* chars.""

`char+` *c-addr1 – c-addr2* core "char-plus"

`1 chars +`.

`cells` *n1 – n2* core "cells"

n2 is the number of address units of *n1* cells.

`cell+` *a-addr1 – a-addr2* core "cell-plus"

`1 cells +`

`cell` *– u* gforth "cell"

`Constant` – `1 cells`

`aligned` *c-addr – a-addr* core "aligned"

a-addr is the first aligned address greater than or equal to *c-addr*.

`floats` *n1 – n2* float "floats"

n2 is the number of address units of *n1* floats.

`float+` *f-addr1 – f-addr2* float "float-plus"

`1 floats +`.

`float` *– u* gforth "float"

`Constant` – the number of address units corresponding to a floating-point number.

`faligned` *c-addr – f-addr* float "f-aligned"

f-addr is the first float-aligned address greater than or equal to *c-addr*.

`sfloats` *n1 – n2* float-ext "s-floats"

n2 is the number of address units of *n1* single-precision IEEE floating-point numbers.

`sfloat+` *sf-addr1 – sf-addr2* float-ext "s-float-plus"

`1 sfloats +`.

`sfaligned` *c-addr – sf-addr* float-ext "s-f-aligned"

sf-addr is the first single-float-aligned address greater than or equal to *c-addr*.

`dfloats` *n1* – *n2* float-ext "d-floats"

n2 is the number of address units of *n1* double-precision IEEE floating-point numbers.

`dfloat+` *df-addr1* – *df-addr2* float-ext "d-float-plus"

`1 dfloats +`.

`dfaligned` *c-addr* – *df-addr* float-ext "d-f-aligned"

df-addr is the first double-float-aligned address greater than or equal to *c-addr*.

`maxaligned` *addr1* – *addr2* gforth "maxaligned"

addr2 is the first address after *addr1* that satisfies all alignment restrictions. maxaligned"

`cfaligned` *addr1* – *addr2* gforth "cfaligned"

addr2 is the first address after *addr1* that is aligned for a code field (i.e., such that the corresponding body is maxaligned).

`ADDRESS-UNIT-BITS` – *n* environment "ADDRESS-UNIT-BITS"

Size of one address unit, in bits.

5.7.6 Memory Blocks

Memory blocks often represent character strings; For ways of storing character strings in memory see Section 5.19.3 [String Formats], page 117. For other string-processing words see Section 5.19.4 [Displaying characters and strings], page 117.

A few of these words work on address unit blocks. In that case, you usually have to insert `CHARS` before the word when working on character strings. Most words work on character blocks, and expect a char-aligned address.

When copying characters between overlapping memory regions, use `chars move` or choose carefully between `cmove` and `cmove>`.

`move` *c-from c-to ucount* – core "move"

Copy the contents of *ucount* aus at *c-from* to *c-to*. `move` works correctly even if the two areas overlap.

`erase` *addr u* – core-ext "erase"

Clear all bits in *u* aus starting at *addr*.

`cmove` *c-from c-to u* – string "c-move"

Copy the contents of *ucount* characters from data space at *c-from* to *c-to*. The copy proceeds `char`-by-`char` from low address to high address; i.e., for overlapping areas it is safe if *c-to*=`<`*c-from*.

`cmove>` *c-from c-to u* – string "c-move-up"

Copy the contents of *ucount* characters from data space at *c-from* to *c-to*. The copy proceeds `char`-by-`char` from high address to low address; i.e., for overlapping areas it is safe if *c-to*`>`=*c-from*.

`fill` *c-addr u c* – core "fill"

Store *c* in *u* chars starting at *c-addr*.

`blank` *c-addr u* – string "blank"

Store the space character into *u* chars starting at *c-addr*.

`compare` *c-addr1 u1 c-addr2 u2 – n* string "compare"

Compare two strings lexicographically. If they are equal, *n* is 0; if the first string is smaller, *n* is -1; if the first string is larger, *n* is 1. Currently this is based on the machine's character comparison. In the future, this may change to consider the current locale and its collation order.

`str=` *c-addr1 u1 c-addr2 u2 – f* gforth "str="

`str<` *c-addr1 u1 c-addr2 u2 – f* gforth "str<"

`string-prefix?` *c-addr1 u1 c-addr2 u2 – f* gforth "string-prefix?"

Is *c-addr2 u2* a prefix of *c-addr1 u1*?

`search` *c-addr1 u1 c-addr2 u2 – c-addr3 u3 flag* string "search"

Search the string specified by *c-addr1, u1* for the string specified by *c-addr2, u2*. If *flag* is true: match was found at *c-addr3* with *u3* characters remaining. If *flag* is false: no match was found; *c-addr3, u3* are equal to *c-addr1, u1*.

`-trailing` *c_addr u1 – c_addr u2* string "dash-trailing"

Adjust the string specified by *c-addr, u1* to remove all trailing spaces. *u2* is the length of the modified string.

`/string` *c-addr1 u1 n – c-addr2 u2* string "slash-string"

Adjust the string specified by *c-addr1, u1* to remove *n* characters from the start of the string.

`bounds` *addr u – addr+u addr* gforth "bounds"

Given a memory block represented by starting address *addr* and length *u* in aus, produce the end address *addr+u* and the start address in the right order for `u+do` or `?do`.

5.8 Control Structures

Control structures in Forth cannot be used interpretively, only in a colon definition[3]. We do not like this limitation, but have not seen a satisfying way around it yet, although many schemes have been proposed.

5.8.1 Selection

```
flag
IF
  code
ENDIF
```

If *flag* is non-zero (as far as `IF` etc. are concerned, a cell with any bit set represents truth) *code* is executed.

```
flag
IF
  code1
ELSE
  code2
```

[3] To be precise, they have no interpretation semantics (see Section 5.10 [Interpretation and Compilation Semantics], page 85).

```
ENDIF
```

If *flag* is true, *code1* is executed, otherwise *code2* is executed.

You can use `THEN` instead of `ENDIF`. Indeed, `THEN` is standard, and `ENDIF` is not, although it is quite popular. We recommend using `ENDIF`, because it is less confusing for people who also know other languages (and is not prone to reinforcing negative prejudices against Forth in these people). Adding `ENDIF` to a system that only supplies `THEN` is simple:

```
: ENDIF   POSTPONE then ; immediate
```

[According to *Webster's New Encyclopedic Dictionary*, *then (adv.)* has the following meanings:

> ... 2b: following next after in order ... 3d: as a necessary consequence (if you were there, then you saw them).

Forth's `THEN` has the meaning 2b, whereas `THEN` in Pascal and many other programming languages has the meaning 3d.]

Gforth also provides the words `?DUP-IF` and `?DUP-0=-IF`, so you can avoid using `?dup`. Using these alternatives is also more efficient than using `?dup`. Definitions in ANS Forth for `ENDIF`, `?DUP-IF` and `?DUP-0=-IF` are provided in '`compat/control.fs`'.

```
n
CASE
  n1 OF code1 ENDOF
  n2 OF code2 ENDOF
  ...
  ( n ) default-code ( n )
ENDCASE
```

Executes the first *codei*, where the *ni* is equal to *n*. If no *ni* matches, the optional *default-code* is executed. The optional default case can be added by simply writing the code after the last `ENDOF`. It may use *n*, which is on top of the stack, but must not consume it.

Programming style note: To keep the code understandable, you should ensure that on all paths through a selection construct the stack is changed in the same way (wrt. number and types of stack items consumed and pushed).

5.8.2 Simple Loops

```
BEGIN
  code1
  flag
WHILE
  code2
REPEAT
```

code1 is executed and *flag* is computed. If it is true, *code2* is executed and the loop is restarted; If *flag* is false, execution continues after the `REPEAT`.

```
BEGIN
  code
  flag
UNTIL
```

code is executed. The loop is restarted if `flag` is false.

Programming style note: To keep the code understandable, a complete iteration of the loop should not change the number and types of the items on the stacks.

```
BEGIN
  code
AGAIN
```

This is an endless loop.

5.8.3 Counted Loops

The basic counted loop is:

```
limit start
?DO
  body
LOOP
```

This performs one iteration for every integer, starting from *start* and up to, but excluding *limit*. The counter, or *index*, can be accessed with `i`. For example, the loop:

```
10 0 ?DO
  i .
LOOP
```

prints `0 1 2 3 4 5 6 7 8 9`

The index of the innermost loop can be accessed with `i`, the index of the next loop with `j`, and the index of the third loop with `k`.

`i` *R:n – R:n n* core "i"

`j` *R:n R:d1 – n R:n R:d1* core "j"

`k` *R:n R:d1 R:d2 – n R:n R:d1 R:d2* gforth "k"

The loop control data are kept on the return stack, so there are some restrictions on mixing return stack accesses and counted loop words. In particuler, if you put values on the return stack outside the loop, you cannot read them inside the loop[4]. If you put values on the return stack within a loop, you have to remove them before the end of the loop and before accessing the index of the loop.

There are several variations on the counted loop:

- `LEAVE` leaves the innermost counted loop immediately; execution continues after the associated `LOOP` or `NEXT`. For example:

  ```
  10 0 ?DO  i DUP . 3 = IF LEAVE THEN LOOP
  ```

  prints `0 1 2 3`
- `UNLOOP` prepares for an abnormal loop exit, e.g., via `EXIT`. `UNLOOP` removes the loop control parameters from the return stack so `EXIT` can get to its return address. For example:

  ```
  : demo 10 0 ?DO i DUP . 3 = IF UNLOOP EXIT THEN LOOP ." Done" ;
  ```

  prints `0 1 2 3`

[4] well, not in a way that is portable.

- If *start* is greater than *limit*, a `?DO` loop is entered (and `LOOP` iterates until they become equal by wrap-around arithmetic). This behaviour is usually not what you want. Therefore, Gforth offers `+DO` and `U+DO` (as replacements for `?DO`), which do not enter the loop if *start* is greater than *limit*; `+DO` is for signed loop parameters, `U+DO` for unsigned loop parameters.
- `?DO` can be replaced by `DO`. `DO` always enters the loop, independent of the loop parameters. Do not use `DO`, even if you know that the loop is entered in any case. Such knowledge tends to become invalid during maintenance of a program, and then the `DO` will make trouble.
- `LOOP` can be replaced with *n* `+LOOP`; this updates the index by *n* instead of by 1. The loop is terminated when the border between *limit-1* and *limit* is crossed. E.g.:

  ```
  4 0 +DO  i .  2 +LOOP
  ```

  prints `0 2`

  ```
  4 1 +DO  i .  2 +LOOP
  ```

  prints `1 3`
- The behaviour of *n* `+LOOP` is peculiar when *n* is negative:

  ```
  -1 0 ?DO  i .  -1 +LOOP
  ```

  prints `0 -1`

  ```
  0 0 ?DO  i .  -1 +LOOP
  ```

  prints nothing.

  Therefore we recommend avoiding *n* `+LOOP` with negative *n*. One alternative is *u* `-LOOP`, which reduces the index by *u* each iteration. The loop is terminated when the border between *limit+1* and *limit* is crossed. Gforth also provides `-DO` and `U-DO` for down-counting loops. E.g.:

  ```
  -2 0 -DO  i .  1 -LOOP
  ```

  prints `0 -1`

  ```
  -1 0 -DO  i .  1 -LOOP
  ```

  prints `0`

  ```
  0 0 -DO  i .  1 -LOOP
  ```

  prints nothing.

Unfortunately, `+DO`, `U+DO`, `-DO`, `U-DO` and `-LOOP` are not defined in ANS Forth. However, an implementation for these words that uses only standard words is provided in '`compat/loops.fs`'.

Another counted loop is:

```
n
FOR
  body
NEXT
```

This is the preferred loop of native code compiler writers who are too lazy to optimize `?DO` loops properly. This loop structure is not defined in ANS Forth. In Gforth, this loop iterates *n+1* times; `i` produces values starting with *n* and ending with 0. Other Forth systems may behave differently, even if they support `FOR` loops. To avoid problems, don't use `FOR` loops.

5.8.4 Arbitrary control structures

ANS Forth permits and supports using control structures in a non-nested way. Information about incomplete control structures is stored on the control-flow stack. This stack may be implemented on the Forth data stack, and this is what we have done in Gforth.

An *orig* entry represents an unresolved forward branch, a *dest* entry represents a backward branch target. A few words are the basis for building any control structure possible (except control structures that need storage, like calls, coroutines, and backtracking).

`IF` *compilation – orig ; run-time f –* core "IF"

`AHEAD` *compilation – orig ; run-time –* tools-ext "AHEAD"

`THEN` *compilation orig – ; run-time –* core "THEN"

`BEGIN` *compilation – dest ; run-time –* core "BEGIN"

`UNTIL` *compilation dest – ; run-time f –* core "UNTIL"

`AGAIN` *compilation dest – ; run-time –* core-ext "AGAIN"

`CS-PICK` *... u – ... destu* tools-ext "c-s-pick"

`CS-ROLL` *destu/origu .. dest0/orig0 u – .. dest0/orig0 destu/origu* tools-ext "c-s-roll"

The Standard words `CS-PICK` and `CS-ROLL` allow you to manipulate the control-flow stack in a portable way. Without them, you would need to know how many stack items are occupied by a control-flow entry (many systems use one cell. In Gforth they currently take three, but this may change in the future).

Some standard control structure words are built from these words:

`ELSE` *compilation orig1 – orig2 ; run-time f –* core "ELSE"

`WHILE` *compilation dest – orig dest ; run-time f –* core "WHILE"

`REPEAT` *compilation orig dest – ; run-time –* core "REPEAT"

Gforth adds some more control-structure words:

`ENDIF` *compilation orig – ; run-time –* gforth "ENDIF"

`?DUP-IF` *compilation – orig ; run-time n – n|* gforth "question-dupe-if"

This is the preferred alternative to the idiom `"?DUP IF"`, since it can be better handled by tools like stack checkers. Besides, it's faster.

`?DUP-0=-IF` *compilation – orig ; run-time n – n|* gforth "question-dupe-zero-equals-if"

Counted loop words constitute a separate group of words:

`?DO` *compilation – do-sys ; run-time w1 w2 – | loop-sys* core-ext "question-do"

`+DO` *compilation – do-sys ; run-time n1 n2 – | loop-sys* gforth "plus-do"

`U+DO` *compilation – do-sys ; run-time u1 u2 – | loop-sys* gforth "u-plus-do"

`-DO` *compilation – do-sys ; run-time n1 n2 – | loop-sys* gforth "minus-do"

`U-DO` *compilation – do-sys ; run-time u1 u2 – | loop-sys* gforth "u-minus-do"

`DO` *compilation – do-sys ; run-time w1 w2 – loop-sys* core "DO"

`FOR` *compilation – do-sys ; run-time u – loop-sys* gforth "FOR"

`LOOP` *compilation do-sys – ; run-time loop-sys1 – | loop-sys2* core "LOOP"

`+LOOP` *compilation do-sys – ; run-time loop-sys1 n – | loop-sys2* core "plus-loop"

`-LOOP` *compilation do-sys – ; run-time loop-sys1 u – | loop-sys2* gforth "minus-loop"

`NEXT` *compilation do-sys – ; run-time loop-sys1 – | loop-sys2* gforth "NEXT"

`LEAVE` *compilation – ; run-time loop-sys –* core "LEAVE"

`?LEAVE` *compilation – ; run-time f | f loop-sys –* gforth "question-leave"

`unloop` *R:w1 R:w2 –* core "unloop"

`DONE` *compilation orig – ; run-time –* gforth "DONE"

The standard does not allow using `CS-PICK` and `CS-ROLL` on *do-sys*. Gforth allows it, but it's your job to ensure that for every `?DO` etc. there is exactly one `UNLOOP` on any path through the definition (`LOOP` etc. compile an `UNLOOP` on the fall-through path). Also, you have to ensure that all `LEAVE`s are resolved (by using one of the loop-ending words or `DONE`).

Another group of control structure words are:

`case` *compilation – case-sys ; run-time –* core-ext "case"

`endcase` *compilation case-sys – ; run-time x –* core-ext "end-case"

`of` *compilation – of-sys ; run-time x1 x2 – |x1* core-ext "of"

`endof` *compilation case-sys1 of-sys – case-sys2 ; run-time –* core-ext "end-of"

case-sys and *of-sys* cannot be processed using `CS-PICK` and `CS-ROLL`.

5.8.4.1 Programming Style

In order to ensure readability we recommend that you do not create arbitrary control structures directly, but define new control structure words for the control structure you want and use these words in your program. For example, instead of writing:

```
BEGIN
  ...
IF [ 1 CS-ROLL ]
  ...
AGAIN THEN
```

we recommend defining control structure words, e.g.,

```
: WHILE ( DEST -- ORIG DEST )
 POSTPONE IF
 1 CS-ROLL ; immediate

: REPEAT ( orig dest -- )
 POSTPONE AGAIN
 POSTPONE THEN ; immediate
```

and then using these to create the control structure:

```
BEGIN
  ...
WHILE
  ...
REPEAT
```

That's much easier to read, isn't it? Of course, `REPEAT` and `WHILE` are predefined, so in this example it would not be necessary to define them.

5.8.5 Calls and returns

A definition can be called simply be writing the name of the definition to be called. Normally a definition is invisible during its own definition. If you want to write a directly recursive definition, you can use `recursive` to make the current definition visible, or `recurse` to call the current definition directly.

`recursive` *compilation – ; run-time –* gforth "recursive"

Make the current definition visible, enabling it to call itself recursively.

`recurse` *compilation – ; run-time ?? – ??* core "recurse"

Call the current definition.

> Programming style note: I prefer using `recursive` to `recurse`, because calling the definition by name is more descriptive (if the name is well-chosen) than the somewhat cryptic `recurse`. E.g., in a quicksort implementation, it is much better to read (and think) "now sort the partitions" than to read "now do a recursive call".

For mutual recursion, use `Defer`red words, like this:

```
Defer foo

: bar ( ... -- ... )
 ... foo ... ;

:noname ( ... -- ... )
 ... bar ... ;
IS foo
```

Deferred words are discussed in more detail in Section 5.9.9 [Deferred words], page 82.

The current definition returns control to the calling definition when the end of the definition is reached or `EXIT` is encountered.

`EXIT` *compilation – ; run-time nest-sys –* core "EXIT"

Return to the calling definition; usually used as a way of forcing an early return from a definition. Before `EXIT`ing you must clean up the return stack and `UNLOOP` any outstanding `?DO`...`LOOP`s.

`;s` *R:w –* gforth "semis"

The primitive compiled by `EXIT`.

5.8.6 Exception Handling

If a word detects an error condition that it cannot handle, it can `throw` an exception. In the simplest case, this will terminate your program, and report an appropriate error.

`throw` *y1 .. ym nerror – y1 .. ym / z1 .. zn error* exception "throw"

If *nerror* is 0, drop it and continue. Otherwise, transfer control to the next dynamically enclosing exception handler, reset the stacks accordingly, and push *nerror*.

`Throw` consumes a cell-sized error number on the stack. There are some predefined error numbers in ANS Forth (see '`errors.fs`'). In Gforth (and most other systems) you can use the iors produced by various words as error numbers (e.g., a typical use of `allocate` is `allocate throw`). Gforth also provides the word `exception` to define your own error numbers (with decent error reporting); an ANS Forth version of this word (but without the error messages) is available in `compat/except.fs`. And finally, you can use your own error numbers (anything outside the range -4095..0), but won't get nice error messages, only numbers. For example, try:

```
-10 throw                    \ ANS defined
-267 throw                   \ system defined
s" my error" exception throw \ user defined
7 throw                      \ arbitrary number
```

`exception` *addr u – n* gforth "exception"

n is a previously unused `throw` value in the range (-4095...-256). Consecutive calls to `exception` return consecutive decreasing numbers. Gforth uses the string *addr u* as an error message.

A common idiom to `THROW` a specific error if a flag is true is this:

```
( flag ) 0<> errno and throw
```

Your program can provide exception handlers to catch exceptions. An exception handler can be used to correct the problem, or to clean up some data structures and just throw the exception to the next exception handler. Note that `throw` jumps to the dynamically innermost exception handler. The system's exception handler is outermost, and just prints an error and restarts command-line interpretation (or, in batch mode (i.e., while processing the shell command line), leaves Gforth).

The ANS Forth way to catch exceptions is `catch`:

`catch` *... xt – ... n* exception "catch"

The most common use of exception handlers is to clean up the state when an error happens. E.g.,

```
base  >r hex \ actually the hex should be inside foo, or we h
['] foo catch ( nerror|0 )
r> base !
( nerror|0 ) throw \ pass it on
```

A use of `catch` for handling the error `myerror` might look like this:

```
['] foo catch
CASE
  myerror OF ... ( do something about it ) ENDOF
  dup throw \ default: pass other errors on, do nothing on non-errors
ENDCASE
```

Having to wrap the code into a separate word is often cumbersome, therefore Gforth provides an alternative syntax:

```
TRY
  code1
RECOVER     \ optional
  code2 \ optional
ENDTRY
```

This performs *Code1*. If *code1* completes normally, execution continues after the `endtry`. If *Code1* throws, the stacks are reset to the state during `try`, the throw value is pushed on the data stack, and execution constinues at *code2*, and finally falls through the `endtry` into the following code.

`try` *compilation – orig ; run-time –* gforth "try"

`recover` *compilation orig1 – orig2 ; run-time –* gforth "recover"

`endtry` *compilation orig – ; run-time –* gforth "endtry"

The cleanup example from above in this syntax:

```
base  >r TRY
  hex foo \ now the hex is placed correctly
  0       \ value for throw
RECOVER ENDTRY
r> base ! throw
```

And here's the error handling example:

```
TRY
  foo
RECOVER
  CASE
    myerror OF ... ( do something about it ) ENDOF
    throw \ pass other errors on
  ENDCASE
ENDTRY
```

Programming style note: As usual, you should ensure that the stack depth is statically known at the end: either after the `throw` for passing on errors, or after the `ENDTRY` (or, if you use `catch`, after the end of the selection construct for handling the error).

There are two alternatives to `throw`: `Abort"` is conditional and you can provide an error message. `Abort` just produces an "Aborted" error.

The problem with these words is that exception handlers cannot differentiate between different `abort"`s; they just look like `-2 throw` to them (the error message cannot be accessed by standard programs). Similar `abort` looks like `-1 throw` to exception handlers.

`ABORT"` *compilation 'ccc"' – ; run-time f –* core,exception-ext "abort-quote"

If any bit of *f* is non-zero, perform the function of `-2 throw`, displaying the string *ccc* if there is no exception frame on the exception stack.

`abort` *?? – ??* core,exception-ext "abort"

`-1 throw`.

5.9 Defining Words

Defining words are used to extend Forth by creating new entries in the dictionary.

5.9.1 CREATE

Defining words are used to create new entries in the dictionary. The simplest defining word is `CREATE`. `CREATE` is used like this:

```
CREATE new-word1
```

`CREATE` is a parsing word, i.e., it takes an argument from the input stream (`new-word1` in our example). It generates a dictionary entry for `new-word1`. When `new-word1` is executed, all that it does is leave an address on the stack. The address represents the value of the data space pointer (`HERE`) at the time that `new-word1` was defined. Therefore, `CREATE` is a way of associating a name with the address of a region of memory.

`Create` "*name*" – core "Create"

Note that in ANS Forth guarantees only for `create` that its body is in dictionary data space (i.e., where `here`, `allot` etc. work, see Section 5.7.2 [Dictionary allocation], page 59). Also, in ANS Forth only `create`d words can be modified with `does>` (see Section 5.9.8 [User-defined Defining Words], page 76). And in ANS Forth `>body` can only be applied to `create`d words.

By extending this example to reserve some memory in data space, we end up with something like a *variable*. Here are two different ways to do it:

```
CREATE new-word2 1 cells allot  \ reserve 1 cell - initial value undefined
CREATE new-word3 4 ,            \ reserve 1 cell and initialise it (to 4)
```

The variable can be examined and modified using `@` ("fetch") and `!` ("store") like this:

```
new-word2 @ .      \ get address, fetch from it and display
1234 new-word2 !   \ new value, get address, store to it
```

A similar mechanism can be used to create arrays. For example, an 80-character text input buffer:

```
CREATE text-buf 80 chars allot

text-buf 0 chars c@ \ the 1st character (offset 0)
text-buf 3 chars c@ \ the 4th character (offset 3)
```

You can build arbitrarily complex data structures by allocating appropriate areas of memory. For further discussions of this, and to learn about some Gforth tools that make it easier, See Section 5.21 [Structures], page 128.

5.9.2 Variables

The previous section showed how a sequence of commands could be used to generate a variable. As a final refinement, the whole code sequence can be wrapped up in a defining word (pre-empting the subject of the next section), making it easier to create new variables:

```
: myvariableX ( "name" -- a-addr ) CREATE 1 cells allot ;
: myvariable0 ( "name" -- a-addr ) CREATE 0 , ;

myvariableX foo \ variable foo starts off with an unknown value
myvariable0 joe \ whilst joe is initialised to 0

45 3 * foo !   \ set foo to 135
1234 joe !     \ set joe to 1234
3 joe +!       \ increment joe by 3.. to 1237
```

Not surprisingly, there is no need to define `myvariable`, since Forth already has a definition `Variable`. ANS Forth does not guarantee that a `Variable` is initialised when it is created (i.e., it may behave like `myvariableX`). In contrast, Gforth's `Variable` initialises the variable to 0 (i.e., it behaves exactly like `myvariable0`). Forth also provides `2Variable` and `fvariable` for double and floating-point variables, respectively – they are initialised to 0. and 0e in Gforth. If you use a `Variable` to store a boolean, you can use `on` and `off` to toggle its state.

`Variable` "*name*" – core "Variable"

`2Variable` "*name*" – double "two-variable"

`fvariable` "*name*" – float "f-variable"

The defining word `User` behaves in the same way as `Variable`. The difference is that it reserves space in *user (data) space* rather than normal data space. In a Forth system that has a multi-tasker, each task has its own set of user variables.

`User` "*name*" – gforth "User"

5.9.3 Constants

`Constant` allows you to declare a fixed value and refer to it by name. For example:

```
12 Constant INCHES-PER-FOOT
3E+08 fconstant SPEED-O-LIGHT
```

A `Variable` can be both read and written, so its run-time behaviour is to supply an address through which its current value can be manipulated. In contrast, the value of a `Constant` cannot be changed once it has been declared[5] so it's not necessary to supply the address – it is more efficient to return the value of the constant directly. That's exactly what happens; the run-time effect of a constant is to put its value on the top of the stack (You can find one way of implementing `Constant` in Section 5.9.8 [User-defined Defining Words], page 76).

Forth also provides `2Constant` and `fconstant` for defining double and floating-point constants, respectively.

`Constant` *w* "*name*" – core "Constant"

Define a constant *name* with value *w*.

name execution: – *w*

`2Constant` *w1 w2* "*name*" – double "two-constant"

`fconstant` *r* "*name*" – float "f-constant"

Constants in Forth behave differently from their equivalents in other programming languages. In other languages, a constant (such as an EQU in assembler or a #define in C) only exists at compile-time; in the executable program the constant has been translated into an absolute number and, unless you are using a symbolic debugger, it's impossible to know what abstract thing that number represents. In Forth a constant has an entry in the header space and remains there after the code that uses it has been defined. In fact, it must remain in the dictionary since it has run-time duties to perform. For example:

```
12 Constant INCHES-PER-FOOT
: FEET-TO-INCHES ( n1 -- n2 ) INCHES-PER-FOOT * ;
```

When `FEET-TO-INCHES` is executed, it will in turn execute the xt associated with the constant `INCHES-PER-FOOT`. If you use `see` to decompile the definition of `FEET-TO-INCHES`, you can see that it makes a call to `INCHES-PER-FOOT`. Some Forth compilers attempt to optimise constants by in-lining them where they are used. You can force Gforth to in-line a constant like this:

```
: FEET-TO-INCHES ( n1 -- n2 ) [ INCHES-PER-FOOT ] LITERAL * ;
```

[5] Well, often it can be – but not in a Standard, portable way. It's safer to use a `Value` (read on).

If you use `see` to decompile *this* version of `FEET-TO-INCHES`, you can see that `INCHES-PER-FOOT` is no longer present. To understand how this works, read Section 5.13.3 [Interpret/Compile states], page 97, and Section 5.12.1 [Literals], page 89.

In-lining constants in this way might improve execution time fractionally, and can ensure that a constant is now only referenced at compile-time. However, the definition of the constant still remains in the dictionary. Some Forth compilers provide a mechanism for controlling a second dictionary for holding transient words such that this second dictionary can be deleted later in order to recover memory space. However, there is no standard way of doing this.

5.9.4 Values

A `Value` behaves like a `Constant`, but it can be changed. `TO` is a parsing word that changes a `Values`. In Gforth (not in ANS Forth) you can access (and change) a `value` also with `>body`.

Here are some examples:

```
12 Value APPLES     \ Define APPLES with an initial value of 12
34 TO APPLES        \ Change the value of APPLES. TO is a parsing word
1 ' APPLES >body +! \ Increment APPLES.  Non-standard usage.
APPLES              \ puts 35 on the top of the stack.
```

`Value` *w* "*name*" – core-ext "Value"

`TO` *w* "*name*" – core-ext "TO"

5.9.5 Colon Definitions

```
: name ( ... -- ... )
    word1 word2 word3 ;
```

Creates a word called `name` that, upon execution, executes `word1 word2 word3`. `name` is a *(colon) definition*.

The explanation above is somewhat superficial. For simple examples of colon definitions see Section 4.3 [Your first definition], page 42. For an in-depth discussion of some of the issues involved, See Section 5.10 [Interpretation and Compilation Semantics], page 85.

`:` "*name*" – *colon-sys* core "colon"

`;` *compilation colon-sys – ; run-time nest-sys* core "semicolon"

5.9.6 Anonymous Definitions

Sometimes you want to define an *anonymous word*; a word without a name. You can do this with:

`:noname` – *xt colon-sys* core-ext "colon-no-name"

This leaves the execution token for the word on the stack after the closing `;`. Here's an example in which a deferred word is initialised with an `xt` from an anonymous colon definition:

```
Defer deferred
:noname ( ... -- ... )
  ... ;
```

```
IS deferred
```

Gforth provides an alternative way of doing this, using two separate words:

`noname` – gforth "noname"

The next defined word will be anonymous. The defining word will leave the input stream alone. The xt of the defined word will be given by `lastxt`.

`lastxt` – *xt* gforth "lastxt"

xt is the execution token of the last word defined.

The previous example can be rewritten using `noname` and `lastxt`:

```
Defer deferred
noname : ( ... -- ... )
  ... ;
lastxt IS deferred
```

`noname` works with any defining word, not just `:`.

`lastxt` also works when the last word was not defined as `noname`. It does not work for combined words, though. It also has the useful property that is is valid as soon as the header for a definition has been built. Thus:

```
lastxt . : foo [ lastxt . ] ; ' foo .
```

prints 3 numbers; the last two are the same.

5.9.7 Supplying the name of a defined word

By default, a defining word takes the name for the defined word from the input stream. Sometimes you want to supply the name from a string. You can do this with:

`nextname` *c-addr u* – gforth "nextname"

The next defined word will have the name *c-addr u*; the defining word will leave the input stream alone.

For example:

```
s" foo" nextname create
```

is equivalent to:

```
create foo
```

`nextname` works with any defining word.

5.9.8 User-defined Defining Words

You can create a new defining word by wrapping defining-time code around an existing defining word and putting the sequence in a colon definition.

For example, suppose that you have a word `stats` that gathers statistics about colon definitions given the *xt* of the definition, and you want every colon definition in your application to make a call to `stats`. You can define and use a new version of `:` like this:

```
: stats ( xt -- ) DUP ." (Gathering statistics for " . ." )"
  ... ;  \ other code

: my: : lastxt postpone literal ['] stats compile, ;
```

```
my: foo + - ;
```

When `foo` is defined using `my:` these steps occur:

- `my:` is executed.
- The `:` within the definition (the one between `my:` and `lastxt`) is executed, and does just what it always does; it parses the input stream for a name, builds a dictionary header for the name `foo` and switches `state` from interpret to compile.
- The word `lastxt` is executed. It puts the *xt* for the word that is being defined – `foo` – onto the stack.
- The code that was produced by `postpone literal` is executed; this causes the value on the stack to be compiled as a literal in the code area of `foo`.
- The code `['] stats` compiles a literal into the definition of `my:`. When `compile,` is executed, that literal – the execution token for `stats` – is layed down in the code area of `foo` , following the literal[6].
- At this point, the execution of `my:` is complete, and control returns to the text interpreter. The text interpreter is in compile state, so subsequent text `+ -` is compiled into the definition of `foo` and the `;` terminates the definition as always.

You can use `see` to decompile a word that was defined using `my:` and see how it is different from a normal `:` definition. For example:

```
: bar + - ;  \ like foo but using : rather than my:
see bar
: bar
  + - ;
see foo
: foo
  107645672 stats + - ;

\ use ' stats . to show that 107645672 is the xt for stats
```

You can use techniques like this to make new defining words in terms of *any* existing defining word.

If you want the words defined with your defining words to behave differently from words defined with standard defining words, you can write your defining word like this:

```
: def-word ( "name" -- )
    CREATE code1
DOES> ( ... -- ... )
    code2 ;

def-word name
```

This fragment defines a *defining word* `def-word` and then executes it. When `def-word` executes, it `CREATE`s a new word, `name`, and executes the code *code1*. The code *code2* is not executed at this time. The word `name` is sometimes called a *child* of `def-word`.

[6] Strictly speaking, the mechanism that `compile,` uses to convert an *xt* into something in the code area is implementation-dependent. A threaded implementation might spit out the execution token directly whilst another implementation might spit out a native code sequence.

When you execute `name`, the address of the body of `name` is put on the data stack and *code2* is executed (the address of the body of `name` is the address `HERE` returns immediately after the `CREATE`, i.e., the address a `create`d word returns by default).

You can use `def-word` to define a set of child words that behave similarly; they all have a common run-time behaviour determined by *code2*. Typically, the *code1* sequence builds a data area in the body of the child word. The structure of the data is common to all children of `def-word`, but the data values are specific – and private – to each child word. When a child word is executed, the address of its private data area is passed as a parameter on TOS to be used and manipulated[7] by *code2*.

The two fragments of code that make up the defining words act (are executed) at two completely separate times:

- At *define time*, the defining word executes *code1* to generate a child word
- At *child execution time*, when a child word is invoked, *code2* is executed, using parameters (data) that are private and specific to the child word.

Another way of understanding the behaviour of `def-word` and `name` is to say that, if you make the following definitions:

```
: def-word1 ( "name" -- )
    CREATE code1 ;

: action1 ( ... -- ... )
    code2 ;

def-word1 name1
```

Then using `name1 action1` is equivalent to using `name`.

The classic example is that you can define `CONSTANT` in this way:

```
: CONSTANT ( w "name" -- )
    CREATE ,
DOES> ( -- w )
    @ ;
```

When you create a constant with `5 CONSTANT five`, a set of define-time actions take place; first a new word `five` is created, then the value 5 is laid down in the body of `five` with `,`. When `five` is executed, the address of the body is put on the stack, and `@` retrieves the value 5. The word `five` has no code of its own; it simply contains a data field and a pointer to the code that follows `DOES>` in its defining word. That makes words created in this way very compact.

The final example in this section is intended to remind you that space reserved in `CREATE`d words is *data* space and therefore can be both read and written by a Standard program[8]:

```
: foo ( "name" -- )
    CREATE -1 ,
DOES> ( -- )
    @ . ;
```

[7] It is legitimate both to read and write to this data area.

[8] Exercise: use this example as a starting point for your own implementation of `Value` and `TO` – if you get stuck, investigate the behaviour of `'` and `[']`.

```
foo first-word
foo second-word

123 ' first-word >BODY !
```

If `first-word` had been a `CREATE`d word, we could simply have executed it to get the address of its data field. However, since it was defined to have `DOES>` actions, its execution semantics are to perform those `DOES>` actions. To get the address of its data field it's necessary to use `'` to get its xt, then `>BODY` to translate the xt into the address of the data field. When you execute `first-word`, it will display `123`. When you execute `second-word` it will display `-1`.

In the examples above the stack comment after the `DOES>` specifies the stack effect of the defined words, not the stack effect of the following code (the following code expects the address of the body on the top of stack, which is not reflected in the stack comment). This is the convention that I use and recommend (it clashes a bit with using locals declarations for stack effect specification, though).

5.9.8.1 Applications of `CREATE..DOES>`

You may wonder how to use this feature. Here are some usage patterns:

When you see a sequence of code occurring several times, and you can identify a meaning, you will factor it out as a colon definition. When you see similar colon definitions, you can factor them using `CREATE..DOES>`. E.g., an assembler usually defines several words that look very similar:

```
: ori, ( reg-target reg-source n -- )
    0 asm-reg-reg-imm ;
: andi, ( reg-target reg-source n -- )
    1 asm-reg-reg-imm ;
```

This could be factored with:

```
: reg-reg-imm ( op-code -- )
    CREATE ,
DOES> ( reg-target reg-source n -- )
    @ asm-reg-reg-imm ;

0 reg-reg-imm ori,
1 reg-reg-imm andi,
```

Another view of `CREATE..DOES>` is to consider it as a crude way to supply a part of the parameters for a word (known as *currying* in the functional language community). E.g., `+` needs two parameters. Creating versions of `+` with one parameter fixed can be done like this:

```
: curry+ ( n1 "name" -- )
    CREATE ,
DOES> ( n2 -- n1+n2 )
    @ + ;

 3 curry+ 3+
-2 curry+ 2-
```

5.9.8.2 The gory details of CREATE..DOES>

`DOES>` *compilation colon-sys1 – colon-sys2 ; run-time nest-sys –* core "does"

This means that you need not use `CREATE` and `DOES>` in the same definition; you can put the `DOES>`-part in a separate definition. This allows us to, e.g., select among different `DOES>`-parts:

```
: does1
DOES> ( ... -- ... )
    ... ;

: does2
DOES> ( ... -- ... )
    ... ;

: def-word ( ... -- ... )
    create ...
    IF
       does1
    ELSE
       does2
    ENDIF ;
```

In this example, the selection of whether to use `does1` or `does2` is made at definition-time; at the time that the child word is `CREATE`d.

In a standard program you can apply a `DOES>`-part only if the last word was defined with `CREATE`. In Gforth, the `DOES>`-part will override the behaviour of the last word defined in any case. In a standard program, you can use `DOES>` only in a colon definition. In Gforth, you can also use it in interpretation state, in a kind of one-shot mode; for example:

```
CREATE name ( ... -- ... )
  initialization
DOES>
  code ;
```

is equivalent to the standard:

```
:noname
DOES>
    code ;
CREATE name EXECUTE ( ... -- ... )
    initialization
```

`>body` *xt – a_addr* core ">body"

Get the address of the body of the word represented by *xt* (the address of the word's data field).

5.9.8.3 Advanced does> usage example

The MIPS disassembler ('`arch/mips/disasm.fs`') contains many words for disassembling instructions, that follow a very repetetive scheme:

```
:noname disasm-operands s" inst-name" type ;
entry-num cells table + !
```

Of course, this inspires the idea to factor out the commonalities to allow a definition like

disasm-operands entry-num table `define-inst` *inst-name*

The parameters *disasm-operands* and *table* are usually correlated. Moreover, before I wrote the disassembler, there already existed code that defines instructions like this:

entry-num inst-format inst-name

This code comes from the assembler and resides in '`arch/mips/insts.fs`'.

So I had to define the *inst-format* words that performed the scheme above when executed. At first I chose to use run-time code-generation:

```
: inst-format ( entry-num "name" -- ; compiled code: addr w -- )
  :noname Postpone disasm-operands
  name Postpone sliteral Postpone type Postpone ;
  swap cells table + ! ;
```

Note that this supplies the other two parameters of the scheme above.

An alternative would have been to write this using `create/does>`:

```
: inst-format ( entry-num "name" -- )
  here name string, ( entry-num c-addr ) \ parse and save "name"
  noname create , ( entry-num )
  lastxt swap cells table + !
does> ( addr w -- )
  \ disassemble instruction w at addr
  @ >r
  disasm-operands
  r> count type ;
```

Somehow the first solution is simpler, mainly because it's simpler to shift a string from definition-time to use-time with `sliteral` than with `string,` and friends.

I wrote a lot of words following this scheme and soon thought about factoring out the commonalities among them. Note that this uses a two-level defining word, i.e., a word that defines ordinary defining words.

This time a solution involving `postpone` and friends seemed more difficult (try it as an exercise), so I decided to use a `create/does>` word; since I was already at it, I also used `create/does>` for the lower level (try using `postpone` etc. as an exercise), resulting in the following definition:

```
: define-format ( disasm-xt table-xt -- )
    \ define an instruction format that uses disasm-xt for
    \ disassembling and enters the defined instructions into table
    \ table-xt
    create 2,
does> ( u "inst" -- )
    \ defines an anonymous word for disassembling instruction inst,
    \ and enters it as u-th entry into table-xt
    2@ swap here name string, ( u table-xt disasm-xt c-addr ) \ remember string
    noname create 2,      \ define anonymous word
    execute lastxt swap ! \ enter xt of defined word into table-xt
does> ( addr w -- )
    \ disassemble instruction w at addr
```

```
2@ >r ( addr w disasm-xt R: c-addr )
execute ( R: c-addr ) \ disassemble operands
r> count type ; \ print name
```

Note that the tables here (in contrast to above) do the `cells +` by themselves (that's why you have to pass an xt). This word is used in the following way:

`'` *disasm-operands* `'` *table* `define-format` *inst-format*

As shown above, the defined instruction format is then used like this:

entry-num inst-format inst-name

In terms of currying, this kind of two-level defining word provides the parameters in three stages: first *disasm-operands* and *table*, then *entry-num* and *inst-name*, finally `addr w`, i.e., the instruction to be disassembled.

Of course this did not quite fit all the instruction format names used in '`insts.fs`', so I had to define a few wrappers that conditioned the parameters into the right form.

If you have trouble following this section, don't worry. First, this is involved and takes time (and probably some playing around) to understand; second, this is the first two-level `create/does>` word I have written in seventeen years of Forth; and if I did not have '`insts.fs`' to start with, I may well have elected to use just a one-level defining word (with some repeating of parameters when using the defining word). So it is not necessary to understand this, but it may improve your understanding of Forth.

5.9.8.4 `Const-does>`

A frequent use of `create`...`does>` is for transferring some values from definition-time to run-time. Gforth supports this use with

doc-const-does>

A typical use of this word is:

```
: curry+ ( n1 "name" -- )
1 0 CONST-DOES> ( n2 -- n1+n2 )
    + ;

3 curry+ 3+
```

Here the `1 0` means that 1 cell and 0 floats are transferred from definition to run-time.

The advantages of using `const-does>` are:

You don't have to deal with storing and retrieving the values, i.e., your program becomes more writable and readable.

When using `does>`, you have to introduce a `@` that cannot be optimized away (because you could change the data using `>body`...`!`); `const-does>` avoids this problem.

An ANS Forth implementation of `const-does>` is available in '`compat/const-does.fs`'.

5.9.9 Deferred words

The defining word `Defer` allows you to define a word by name without defining its behaviour; the definition of its behaviour is deferred. Here are two situation where this can be useful:

- Where you want to allow the behaviour of a word to be altered later, and for all precompiled references to the word to change when its behaviour is changed.
- For mutual recursion; See Section 5.8.5 [Calls and returns], page 70.

In the following example, `foo` always invokes the version of `greet` that prints "`Good morning`" whilst `bar` always invokes the version that prints "`Hello`". There is no way of getting `foo` to use the later version without re-ordering the source code and recompiling it.

```
: greet ." Good morning" ;
: foo ... greet ... ;
: greet ." Hello" ;
: bar ... greet ... ;
```

This problem can be solved by defining `greet` as a `Defer`red word. The behaviour of a `Defer`red word can be defined and redefined at any time by using `IS` to associate the xt of a previously-defined word with it. The previous example becomes:

```
Defer greet ( -- )
: foo ... greet ... ;
: bar ... greet ... ;
: greet1 ( -- ) ." Good morning" ;
: greet2 ( -- ) ." Hello" ;
' greet2 <IS> greet  \ make greet behave like greet2
```

Programming style note: You should write a stack comment for every deferred word, and put only XTs into deferred words that conform to this stack effect. Otherwise it's too difficult to use the deferred word.

A deferred word can be used to improve the statistics-gathering example from Section 5.9.8 [User-defined Defining Words], page 76; rather than edit the application's source code to change every `:` to a `my:`, do this:

```
: real: : ;     \ retain access to the original
defer :         \ redefine as a deferred word
' my: <IS> :      \ use special version of :
\
\ load application here
\
' real: <IS> :    \ go back to the original
```

One thing to note is that `<IS>` consumes its name when it is executed. If you want to specify the name at compile time, use `[IS]`:

```
: set-greet ( xt -- )
  [IS] greet ;

' greet1 set-greet
```

A deferred word can only inherit execution semantics from the xt (because that is all that an xt can represent – for more discussion of this see Section 5.11 [Tokens for Words], page 87); by default it will have default interpretation and compilation semantics deriving from this execution semantics. However, you can change the interpretation and compilation semantics of the deferred word in the usual ways:

```
: bar .... ; compile-only
Defer fred immediate
```

```
Defer jim

' bar <IS> jim  \ jim has default semantics
' bar <IS> fred \ fred is immediate
```

`Defer` *"name"* – gforth "Defer"

`<IS>` *"name" xt* – gforth "<IS>"

Changes the `defer`red word *name* to execute *xt*.

`[IS]` *compilation "name" – ; run-time xt* – gforth "bracket-is"

At run-time, changes the `defer`red word *name* to execute *xt*.

`IS` *xt "name"* – gforth "IS"

A combined word made up from `<IS>` and `[IS]`.

`What's` *interpretation "name" – xt; compilation "name" – ; run-time – xt* gforth "What's"

Xt is the XT that is currently assigned to *name*.

`defers` *compilation "name" – ; run-time ... – ...* gforth "defers"

Compiles the present contents of the deferred word *name* into the current definition. I.e., this produces static binding as if *name* was not deferred.

Definitions in ANS Forth for `defer`, `<is>` and `[is]` are provided in '`compat/defer.fs`'.

5.9.10 Aliases

The defining word `Alias` allows you to define a word by name that has the same behaviour as some other word. Here are two situation where this can be useful:

- When you want access to a word's definition from a different word list (for an example of this, see the definition of the `Root` word list in the Gforth source).
- When you want to create a synonym; a definition that can be known by either of two names (for example, `THEN` and `ENDIF` are aliases).

Like deferred words, an alias has default compilation and interpretation semantics at the beginning (not the modifications of the other word), but you can change them in the usual ways (`immediate`, `compile-only`). For example:

```
: foo ... ; immediate

' foo Alias bar \ bar is not an immediate word
' foo Alias fooby immediate \ fooby is an immediate word
```

Words that are aliases have the same xt, different headers in the dictionary, and consequently different name tokens (see Section 5.11 [Tokens for Words], page 87) and possibly different immediate flags. An alias can only have default or immediate compilation semantics; you can define aliases for combined words with `interpret/compile:` – see Section 5.10.1 [Combined words], page 85.

`Alias` *xt "name"* – gforth "Alias"

5.10 Interpretation and Compilation Semantics

The *interpretation semantics* of a (named) word are what the text interpreter does when it encounters the word in interpret state. It also appears in some other contexts, e.g., the execution token returned by `'` *word* identifies the interpretation semantics of *word* (in other words, `'` *word* `execute` is equivalent to interpret-state text interpretation of *word*).

The *compilation semantics* of a (named) word are what the text interpreter does when it encounters the word in compile state. It also appears in other contexts, e.g, `POSTPONE` *word* compiles[9] the compilation semantics of *word*.

The standard also talks about *execution semantics*. They are used only for defining the interpretation and compilation semantics of many words. By default, the interpretation semantics of a word are to `execute` its execution semantics, and the compilation semantics of a word are to `compile,` its execution semantics.[10]

Unnamed words (see Section 5.9.6 [Anonymous Definitions], page 75) cannot be encountered by the text interpreter, ticked, or `postpone`d, so they have no interpretation or compilation semantics. Their behaviour is represented by their XT (see Section 5.11 [Tokens for Words], page 87), and we call it execution semantics, too.

You can change the semantics of the most-recently defined word:

`immediate` – core "immediate"

Make the compilation semantics of a word be to `execute` the execution semantics.

`compile-only` – gforth "compile-only"

Remove the interpretation semantics of a word.

`restrict` – gforth "restrict"

A synonym for `compile-only`

By convention, words with non-default compilation semantics (e.g., immediate words) often have names surrounded with brackets (e.g., `[']`, see Section 5.11.1 [Execution token], page 87).

Note that ticking (`'`) a compile-only word gives an error ("Interpreting a compile-only word").

5.10.1 Combined Words

Gforth allows you to define *combined words* – words that have an arbitrary combination of interpretation and compilation semantics.

`interpret/compile:` *interp-xt comp-xt* `"`*name*`"` – gforth "interpret/compile:"

This feature was introduced for implementing `TO` and `S"`. I recommend that you do not define such words, as cute as they may be: they make it hard to get at both parts of the word in some contexts. E.g., assume you want to get an execution token for the compilation part. Instead, define two words, one that embodies the interpretation part, and one that

[9] In standard terminology, "appends to the current definition".

[10] In standard terminology: The default interpretation semantics are its execution semantics; the default compilation semantics are to append its execution semantics to the execution semantics of the current definition.

embodies the compilation part. Once you have done that, you can define a combined word with `interpret/compile:` for the convenience of your users.

You might try to use this feature to provide an optimizing implementation of the default compilation semantics of a word. For example, by defining:

```
:noname
   foo bar ;
:noname
   POSTPONE foo POSTPONE bar ;
interpret/compile: opti-foobar
```

as an optimizing version of:

```
: foobar
    foo bar ;
```

Unfortunately, this does not work correctly with `[compile]`, because `[compile]` assumes that the compilation semantics of all `interpret/compile:` words are non-default. I.e., `[compile] opti-foobar` would compile compilation semantics, whereas `[compile] foobar` would compile interpretation semantics.

Some people try to use *state-smart* words to emulate the feature provided by `interpret/compile:` (words are state-smart if they check `STATE` during execution). E.g., they would try to code `foobar` like this:

```
: foobar
  STATE @
  IF ( compilation state )
    POSTPONE foo POSTPONE bar
  ELSE
    foo bar
  ENDIF ; immediate
```

Although this works if `foobar` is only processed by the text interpreter, it does not work in other contexts (like `'` or `POSTPONE`). E.g., `' foobar` will produce an execution token for a state-smart word, not for the interpretation semantics of the original `foobar`; when you execute this execution token (directly with `EXECUTE` or indirectly through `COMPILE,`) in compile state, the result will not be what you expected (i.e., it will not perform `foo bar`). State-smart words are a bad idea. Simply don't write them[11]!

It is also possible to write defining words that define words with arbitrary combinations of interpretation and compilation semantics. In general, they look like this:

```
: def-word
    create-interpret/compile
    code1
interpretation>
    code2
<interpretation
compilation>
    code3
<compilation ;
```

[11] For a more detailed discussion of this topic, see M. Anton Ertl, `State`-*smartness—Why it is Evil and How to Exorcise it* (`http://www.complang.tuwien.ac.at/papers/ertl98.ps.gz`), EuroForth '98.

For a *word* defined with `def-word`, the interpretation semantics are to push the address of the body of *word* and perform *code2*, and the compilation semantics are to push the address of the body of *word* and perform *code3*. E.g., `constant` can also be defined like this (except that the defined constants don't behave correctly when `[compile]`d):

```
: constant ( n "name" -- )
    create-interpret/compile
    ,
interpretation> ( -- n )
    @
<interpretation
compilation> ( compilation. -- ; run-time. -- n )
    @ postpone literal
<compilation ;
```

`create-interpret/compile` "*name*" – gforth "create-interpret/compile"

`interpretation>` *compilation. – orig colon-sys* gforth "interpretation>"

`<interpretation` *compilation. orig colon-sys –* gforth "<interpretation"

`compilation>` *compilation. – orig colon-sys* gforth "compilation>"

`<compilation` *compilation. orig colon-sys –* gforth "<compilation"

Words defined with `interpret/compile:` and `create-interpret/compile` have an extended header structure that differs from other words; however, unless you try to access them with plain address arithmetic, you should not notice this. Words for accessing the header structure usually know how to deal with this; e.g., `'` *word* `>body` also gives you the body of a word created with `create-interpret/compile`.

5.11 Tokens for Words

This section describes the creation and use of tokens that represent words.

5.11.1 Execution token

An *execution token* (*XT*) represents some behaviour of a word. You can use `execute` to invoke this behaviour.

You can use `'` to get an execution token that represents the interpretation semantics of a named word:

```
5 ' .   ( n xt )
execute ( )      \ execute the xt (i.e., ".")
```

`'` "*name*" – *xt* core "tick"

xt represents *name*'s interpretation semantics. Perform `-14 throw` if the word has no interpretation semantics.

`'` parses at run-time; there is also a word `[']` that parses when it is compiled, and compiles the resulting XT:

```
: foo ['] . execute ;
5 foo
: bar ' execute ; \ by contrast,
5 bar .           \ ' parses "." when bar executes
```

`[']` *compilation. "name" – ; run-time. – xt* core "bracket-tick"

xt represents *name*'s interpretation semantics. Perform `-14 throw` if the word has no interpretation semantics.

If you want the execution token of *word*, write `[']` *word* in compiled code and `'` *word* in interpreted code. Gforth's `'` and `[']` behave somewhat unusually by complaining about compile-only words (because these words have no interpretation semantics). You might get what you want by using `COMP'` *word* `DROP` or `[COMP']` *word* `DROP` (for details see Section 5.11.2 [Compilation token], page 88).

Another way to get an XT is `:noname` or `lastxt` (see Section 5.9.6 [Anonymous Definitions], page 75). For anonymous words this gives an xt for the only behaviour the word has (the execution semantics). For named words, `lastxt` produces an XT for the same behaviour it would produce if the word was defined anonymously.

```
:noname ." hello" ;
execute
```

An XT occupies one cell and can be manipulated like any other cell.

In ANS Forth the XT is just an abstract data type (i.e., defined by the operations that produce or consume it). For old hands: In Gforth, the XT is implemented as a code field address (CFA).

`execute` *xt –* core "execute"

Perform the semantics represented by the execution token, *xt*.

`perform` *a-addr –* gforth "perform"

`@ execute`.

5.11.2 Compilation token

Gforth represents the compilation semantics of a named word by a *compilation token* consisting of two cells: *w xt*. The top cell *xt* is an execution token. The compilation semantics represented by the compilation token can be performed with `execute`, which consumes the whole compilation token, with an additional stack effect determined by the represented compilation semantics.

At present, the *w* part of a compilation token is an execution token, and the *xt* part represents either `execute` or `compile,`[12]. However, don't rely on that knowledge, unless necessary; future versions of Gforth may introduce unusual compilation tokens (e.g., a compilation token that represents the compilation semantics of a literal).

You can perform the compilation semantics represented by the compilation token with `execute`. You can compile the compilation semantics with `postpone,`. I.e., `COMP'` *word* `postpone,` is equivalent to `postpone` *word*.

`[COMP']` *compilation "name" – ; run-time – w xt* gforth "bracket-comp-tick"

Compilation token *w xt* represents *name*'s compilation semantics.

`COMP'` *"name" – w xt* gforth "comp-tick"

Compilation token *w xt* represents *name*'s compilation semantics.

[12] Depending upon the compilation semantics of the word. If the word has default compilation semantics, the *xt* will represent `compile,`. Otherwise (e.g., for immediate words), the *xt* will represent `execute`.

`postpone,` *w xt* – gforth "postpone-comma"

Compile the compilation semantics represented by the compilation token *w xt*.

5.11.3 Name token

Gforth represents named words by the *name token*, (*nt*). In Gforth, the abstract data type *name token* is implemented as a name field address (NFA).

`find-name` *c-addr u* – *nt* | *0* gforth "find-name"

Find the name *c-addr u* in the current search order. Return its *nt*, if found, otherwise 0.

`name>int` *nt* – *xt* gforth "name>int"

xt represents the interpretation semantics of the word *nt*. If *nt* has no interpretation semantics (i.e. is `compile-only`), *xt* is the execution token for `ticking-compile-only-error`, which performs `-2048 throw`.

`name?int` *nt* – *xt* gforth "name?int"

Like `name>int`, but perform `-2048 throw` if *nt* has no interpretation semantics.

`name>comp` *nt* – *w xt* gforth "name>comp"

w xt is the compilation token for the word *nt*.

`name>string` *nt* – *addr count* gforth "head-to-string"

addr count is the name of the word represented by *nt*.

`id.` *nt* – gforth "id."

Print the name of the word represented by *nt*.

`.name` *nt* – unknown ".name"

Gforth <=0.5.0 name for `id.`.

`.id` *nt* – unknown ".id"

F83 name for `id.`.

5.12 Compiling words

In contrast to most other languages, Forth has no strict boundary between compilation and run-time. E.g., you can run arbitrary code between defining words (or for computing data used by defining words like `constant`). Moreover, `Immediate` (see Section 5.10 [Interpretation and Compilation Semantics], page 85 and `[...]` (see below) allow running arbitrary code while compiling a colon definition (exception: you must not allot dictionary space).

5.12.1 Literals

The simplest and most frequent example is to compute a literal during compilation. E.g., the following definition prints an array of strings, one string per line:

```
: .strings ( addr u -- ) \ gforth
    2* cells bounds U+DO
cr i 2@ type
    2 cells +LOOP ;
```

With a simple-minded compiler like Gforth's, this computes `2 cells` on every loop iteration. You can compute this value once and for all at compile time and compile it into the definition like this:

```
: .strings ( addr u -- ) \ gforth
    2* cells bounds U+DO
cr i 2@ type
    [ 2 cells ] literal +LOOP ;
```

`[` switches the text interpreter to interpret state (you will get an `ok` prompt if you type this example interactively and insert a newline between `[` and `]`), so it performs the interpretation semantics of `2 cells`; this computes a number. `]` switches the text interpreter back into compile state. It then performs `Literal`'s compilation semantics, which are to compile this number into the current word. You can decompile the word with `see .strings` to see the effect on the compiled code.

You can also optimize the `2* cells` into `[ 2 cells ] literal *` in this way.

`[` – core "left-bracket"

Enter interpretation state. Immediate word.

`]` – core "right-bracket"

Enter compilation state.

`Literal` *compilation n – ; run-time – n* core "Literal"

Compilation semantics: compile the run-time semantics.
Run-time Semantics: push *n*.
Interpretation semantics: undefined.

`]L` *compilation: n – ; run-time: – n* gforth "]L"

equivalent to `] literal`

There are also words for compiling other data types than single cells as literals:

`2Literal` *compilation w1 w2 – ; run-time – w1 w2* double "two-literal"

Compile appropriate code such that, at run-time, cell pair *w1, w2* are placed on the stack. Interpretation semantics are undefined.

`FLiteral` *compilation r – ; run-time – r* float "f-literal"

Compile appropriate code such that, at run-time, *r* is placed on the (floating-point) stack. Interpretation semantics are undefined.

`SLiteral` *Compilation c-addr1 u ; run-time – c-addr2 u* string "SLiteral"

Compilation: compile the string specified by *c-addr1*, *u* into the current definition. Run-time: return *c-addr2 u* describing the address and length of the string.

You might be tempted to pass data from outside a colon definition to the inside on the data stack. This does not work, because `:` puhes a colon-sys, making stuff below unaccessible. E.g., this does not work:

```
5 : foo literal ; \ error: "unstructured"
```

Instead, you have to pass the value in some other way, e.g., through a variable:

```
variable temp
5 temp !
: foo [ temp @ ] literal ;
```

5.12.2 Macros

`Literal` and friends compile data values into the current definition. You can also write words that compile other words into the current definition. E.g.,

```
: compile-+ ( -- ) \ compiled code: ( n1 n2 -- n )
  POSTPONE + ;

: foo ( n1 n2 -- n )
  [ compile-+ ] ;
1 2 foo .
```

This is equivalent to `: foo + ;` (`see foo` to check this). What happens in this example? `Postpone` compiles the compilation semantics of `+` into `compile-+`; later the text interpreter executes `compile-+` and thus the compilation semantics of `+`, which compile (the execution semantics of) `+` into `foo`.[13]

`postpone` "*name*" – core "postpone"

Compiles the compilation semantics of *name*.

`[compile]` *compilation* "*name*" *– ; run-time ? – ?* core-ext "bracket-compile"

Compiling words like `compile-+` are usually immediate (or similar) so you do not have to switch to interpret state to execute them; mopifying the last example accordingly produces:

```
: [compile-+] ( compilation: --; interpretation: -- )
  \ compiled code: ( n1 n2 -- n )
  POSTPONE + ; immediate

: foo ( n1 n2 -- n )
  [compile-+] ;
1 2 foo .
```

Immediate compiling words are similar to macros in other languages (in particular, Lisp). The important differences to macros in, e.g., C are:

- You use the same language for defining and processing macros, not a separate preprocessing language and processor.
- Consequently, the full power of Forth is available in macro definitions. E.g., you can perform arbitrarily complex computations, or generate different code conditionally or in a loop (e.g., see Section 3.34 [Advanced macros Tutorial], page 34). This power is very useful when writing a parser generators or other code-generating software.
- Macros defined using `postpone` etc. deal with the language at a higher level than strings; name binding happens at macro definition time, so you can avoid the pitfalls of name collisions that can happen in C macros. Of course, Forth is a liberal language and also allows to shoot yourself in the foot with text-interpreted macros like

  ```
  : [compile-+] s" +" evaluate ; immediate
  ```

  Apart from binding the name at macro use time, using `evaluate` also makes your definition `state`-smart (see [state-smartness], page 86).

[13] A recent RFI answer requires that compiling words should only be executed in compile state, so this example is not guaranteed to work on all standard systems, but on any decent system it will work.

You may want the macro to compile a number into a word. The word to do it is `literal`, but you have to `postpone` it, so its compilation semantics take effect when the macro is executed, not when it is compiled:

```
: [compile-5] ( -- ) \ compiled code: ( -- n )
  5 POSTPONE literal ; immediate

: foo [compile-5] ;
foo .
```

You may want to pass parameters to a macro, that the macro should compile into the current definition. If the parameter is a number, then you can use `postpone literal` (similar for other values).

If you want to pass a word that is to be compiled, the usual way is to pass an execution token and `compile,` it:

```
: twice1 ( xt -- ) \ compiled code: ... -- ...
  dup compile, compile, ;

: 2+ ( n1 -- n2 )
  [ ' 1+ twice1 ] ;
```

`compile,` *xt* – core-ext "compile-comma"

Compile the word represented by the execution token *xt* into the current definition.

An alternative available in Gforth, that allows you to pass compile-only words as parameters is to use the compilation token (see Section 5.11.2 [Compilation token], page 88). The same example in this technique:

```
: twice ( ... ct -- ... ) \ compiled code: ... -- ...
  2dup 2>r execute 2r> execute ;

: 2+ ( n1 -- n2 )
  [ comp' 1+ twice ] ;
```

In the example above `2>r` and `2r>` ensure that `twice` works even if the executed compilation semantics has an effect on the data stack.

You can also define complete definitions with these words; this provides an alternative to using `does>` (see Section 5.9.8 [User-defined Defining Words], page 76). E.g., instead of

```
: curry+ ( n1 "name" -- )
    CREATE ,
DOES> ( n2 -- n1+n2 )
    @ + ;
```

you could define

```
: curry+ ( n1 "name" -- )
  \ name execution: ( n2 -- n1+n2 )
  >r : r> POSTPONE literal POSTPONE + POSTPONE ; ;

-3 curry+ 3-
see 3-
```

The sequence `>r : r>` is necessary, because `:` puts a colon-sys on the data stack that makes everything below it unaccessible.

This way of writing defining words is sometimes more, sometimes less convenient than using `does>` (see Section 5.9.8.3 [Advanced does> usage example], page 80). One advantage of this method is that it can be optimized better, because the compiler knows that the value compiled with `literal` is fixed, whereas the data associated with a `create`d word can be changed.

5.13 The Text Interpreter

The text interpreter[14] is an endless loop that processes input from the current input device. It is also called the outer interpreter, in contrast to the inner interpreter (see Chapter 14 [Engine], page 192) which executes the compiled Forth code on interpretive implementations.

The text interpreter operates in one of two states: *interpret state* and *compile state*. The current state is defined by the aptly-named variable `state`.

This section starts by describing how the text interpreter behaves when it is in interpret state, processing input from the user input device – the keyboard. This is the mode that a Forth system is in after it starts up.

The text interpreter works from an area of memory called the *input buffer*[15], which stores your keyboard input when you press the ⟨RET⟩ key. Starting at the beginning of the input buffer, it skips leading spaces (called *delimiters*) then parses a string (a sequence of non-space characters) until it reaches either a space character or the end of the buffer. Having parsed a string, it makes two attempts to process it:

- It looks for the string in a *dictionary* of definitions. If the string is found, the string names a *definition* (also known as a *word*) and the dictionary search returns information that allows the text interpreter to perform the word's *interpretation semantics*. In most cases, this simply means that the word will be executed.
- If the string is not found in the dictionary, the text interpreter attempts to treat it as a number, using the rules described in Section 5.13.2 [Number Conversion], page 96. If the string represents a legal number in the current radix, the number is pushed onto a parameter stack (the data stack for integers, the floating-point stack for floating-point numbers).

If both attempts fail, or if the word is found in the dictionary but has no interpretation semantics[16] the text interpreter discards the remainder of the input buffer, issues an error message and waits for more input. If one of the attempts succeeds, the text interpreter repeats the parsing process until the whole of the input buffer has been processed, at which point it prints the status message " `ok`" and waits for more input.

The text interpreter keeps track of its position in the input buffer by updating a variable called `>IN` (pronounced "to-in"). The value of `>IN` starts out as 0, indicating an offset of 0 from the start of the input buffer. The region from offset `>IN @` to the end of the input

---

[14] This is an expanded version of the material in Section 4.1 [Introducing the Text Interpreter], page 37.

[15] When the text interpreter is processing input from the keyboard, this area of memory is called the *terminal input buffer* (TIB) and is addressed by the (obsolescent) words `TIB` and `#TIB`.

[16] This happens if the word was defined as `COMPILE-ONLY`.

buffer is called the *parse area*[17]. This example shows how `>IN` changes as the text interpreter parses the input buffer:

```
: remaining >IN @ SOURCE 2 PICK - -ROT + SWAP
  CR ." ->" TYPE ." <-" ; IMMEDIATE

1 2 3 remaining + remaining .

: foo 1 2 3 remaining SWAP remaining ;
```

The result is:

```
->+ remaining .<-
->.<-5  ok

->SWAP remaining ;-<
->;<-  ok
```

The value of `>IN` can also be modified by a word in the input buffer that is executed by the text interpreter. This means that a word can "trick" the text interpreter into either skipping a section of the input buffer[18] or into parsing a section twice. For example:

```
: lat ." <<foo>>" ;
: flat ." <<bar>>" >IN DUP @ 3 - SWAP ! ;
```

When `flat` is executed, this output is produced[19]:

```
<<bar>><<foo>>
```

This technique can be used to work around some of the interoperability problems of parsing words. Of course, it's better to avoid parsing words where possible.

Two important notes about the behaviour of the text interpreter:

- It processes each input string to completion before parsing additional characters from the input buffer.
- It treats the input buffer as a read-only region (and so must your code).

When the text interpreter is in compile state, its behaviour changes in these ways:

- If a parsed string is found in the dictionary, the text interpreter will perform the word's *compilation semantics*. In most cases, this simply means that the execution semantics of the word will be appended to the current definition.
- When a number is encountered, it is compiled into the current definition (as a literal) rather than being pushed onto a parameter stack.
- If an error occurs, `state` is modified to put the text interpreter back into interpret state.
- Each time a line is entered from the keyboard, Gforth prints " `compiled`" rather than " `ok`".

When the text interpreter is using an input device other than the keyboard, its behaviour changes in these ways:

[17] In other words, the text interpreter processes the contents of the input buffer by parsing strings from the parse area until the parse area is empty.

[18] This is how parsing words work.

[19] Exercise for the reader: what would happen if the `3` were replaced with `4`?

- When the parse area is empty, the text interpreter attempts to refill the input buffer from the input source. When the input source is exhausted, the input source is set back to the previous input source.
- It doesn't print out " `ok`" or " `compiled`" messages each time the parse area is emptied.
- If an error occurs, the input source is set back to the user input device.

You can read about this in more detail in Section 5.13.1 [Input Sources], page 95.

`>in` unknown ">in"

`input-var` variable – *a-addr* is the address of a cell containing the char offset from the start of the input buffer to the start of the parse area.

`source` – *addr u* core-ext,file "source"

Return address *addr* and length *u* of the current input buffer

`tib` unknown "tib"

`#tib` unknown "#tib"

`input-var` variable – *a-addr* is the address of a cell containing the number of characters in the terminal input buffer. OBSOLESCENT: `source` superceeds the function of this word.

5.13.1 Input Sources

By default, the text interpreter processes input from the user input device (the keyboard) when Forth starts up. The text interpreter can process input from any of these sources:

- The user input device – the keyboard.
- A file, using the words described in Section 5.17.1 [Forth source files], page 106.
- A block, using the words described in Section 5.18 [Blocks], page 109.
- A text string, using `evaluate`.

A program can identify the current input device from the values of `source-id` and `blk`.

`source-id` – *0* `|` *-1* `|` *fileid* core-ext,file "source-i-d"

Return 0 (the input source is the user input device), -1 (the input source is a string being processed by `evaluate`) or a *fileid* (the input source is the file specified by *fileid*).

`blk` unknown "blk"

`input-var` variable – This cell contains the current block number

`save-input` – *x1 .. xn n* core-ext "save-input"

The *n* entries *xn - x1* describe the current state of the input source specification, in some platform-dependent way that can be used by `restore-input`.

`restore-input` *x1 .. xn n* – *flag* core-ext "restore-input"

Attempt to restore the input source specification to the state described by the *n* entries *xn - x1*. *flag* is true if the restore fails. In Gforth with the new input code, it fails only with a flag that can be used to throw again; it is also possible to save and restore between different active input streams. Note that closing the input streams must happen in the reverse order as they have been opened, but in between everything is allowed.

`evaluate` *... addr u* – *...* core,block "evaluate"

Save the current input source specification. Store `-1` in `source-id` and `0` in `blk`. Set `>IN` to `0` and make the string *c-addr u* the input source and input buffer. Interpret. When the parse area is empty, restore the input source specification.

`query` – core-ext "query"

Make the user input device the input source. Receive input into the Terminal Input Buffer. Set `>IN` to zero. OBSOLESCENT: superceeded by `accept`.

5.13.2 Number Conversion

This section describes the rules that the text interpreter uses when it tries to convert a string into a number.

Let <digit> represent any character that is a legal digit in the current number base[20].

Let <decimal digit> represent any character in the range 0-9.

Let {*a b*} represent the *optional* presence of any of the characters in the braces (*a* or *b* or neither).

Let * represent any number of instances of the previous character (including none).

Let any other character represent itself.

Now, the conversion rules are:

- A string of the form <digit><digit>* is treated as a single-precision (cell-sized) positive integer. Examples are 0 123 6784532 32343212343456 42
- A string of the form -<digit><digit>* is treated as a single-precision (cell-sized) negative integer, and is represented using 2's-complement arithmetic. Examples are -45 -5681 -0
- A string of the form <digit><digit>*.<digit>* is treated as a double-precision (double-cell-sized) positive integer. Examples are 3465. 3.465 34.65 (all three of these represent the same number).
- A string of the form -<digit><digit>*.<digit>* is treated as a double-precision (double-cell-sized) negative integer, and is represented using 2's-complement arithmetic. Examples are -3465. -3.465 -34.65 (all three of these represent the same number).
- A string of the form {+ -}<decimal digit>{.}<decimal digit>*{e E}{+ -}<decimal digit><decimal digit>* is treated as a floating-point number. Examples are 1e 1e0 1.e 1.e0 +1e+0 (which all represent the same number) +12.E-4

By default, the number base used for integer number conversion is given by the contents of the variable `base`. Note that a lot of confusion can result from unexpected values of `base`. If you change `base` anywhere, make sure to save the old value and restore it afterwards. In general I recommend keeping `base` decimal, and using the prefixes described below for the popular non-decimal bases.

`dpl` – *a-addr* gforth "dpl"

`User` variable – *a-addr* is the address of a cell that stores the position of the decimal point in the most recent numeric conversion. Initialised to -1. After the conversion of a number containing no decimal point, `dpl` is -1. After the conversion of `2.` it holds 0. After the conversion of 234123.9 it contains 1, and so forth.

`base` – *a-addr* core "base"

`User` variable – *a-addr* is the address of a cell that stores the number base used by default for number conversion during input and output.

---

[20] For example, 0-9 when the number base is decimal or 0-9, A-F when the number base is hexadecimal.

`hex` – core-ext "hex"

Set `base` to &16 (hexadecimal).

`decimal` – core "decimal"

Set `base` to &10 (decimal).

Gforth allows you to override the value of `base` by using a prefix[21] before the first digit of an (integer) number. Four prefixes are supported:

- `&` – decimal
- `%` – binary
- `$` – hexadecimal
- `'` – base `max-char+1`

Here are some examples, with the equivalent decimal number shown after in braces:

-$41 (-65), %1001101 (205), %1001.0001 (145 - a double-precision number), 'AB (16706; ascii A is 65, ascii B is 66, number is 65*256 + 66), 'ab (24930; ascii a is 97, ascii B is 98, number is 97*256 + 98), &905 (905), $abc (2478), $ABC (2478).

Number conversion has a number of traps for the unwary:

- You cannot determine the current number base using the code sequence `base @ .` – the number base is always 10 in the current number base. Instead, use something like `base @ dec.`
- If the number base is set to a value greater than 14 (for example, hexadecimal), the number 123E4 is ambiguous; the conversion rules allow it to be intepreted as either a single-precision integer or a floating-point number (Gforth treats it as an integer). The ambiguity can be resolved by explicitly stating the sign of the mantissa and/or exponent: 123E+4 or +123E4 – if the number base is decimal, no ambiguity arises; either representation will be treated as a floating-point number.
- There is a word `bin` but it does *not* set the number base! It is used to specify file types.
- ANS Forth requires the `.` of a double-precision number to be the final character in the string. Gforth allows the `.` to be anywhere after the first digit.
- The number conversion process does not check for overflow.
- In an ANS Forth program `base` is required to be decimal when converting floating-point numbers. In Gforth, number conversion to floating-point numbers always uses base &10, irrespective of the value of `base`.

You can read numbers into your programs with the words described in Section 5.19.5 [Input], page 119.

5.13.3 Interpret/Compile states

A standard program is not permitted to change `state` explicitly. However, it can change `state` implicitly, using the words `[` and `]`. When `[` is executed it switches `state` to interpret state, and therefore the text interpreter starts interpreting. When `]` is executed it

[21] Some Forth implementations provide a similar scheme by implementing `$` etc. as parsing words that process the subsequent number in the input stream and push it onto the stack. For example, see *Number Conversion and Literals*, by Wil Baden; Forth Dimensions 20(3) pages 26–27. In such implementations, unlike in Gforth, a space is required between the prefix and the number.

switches `state` to compile state and therefore the text interpreter starts compiling. The most common usage for these words is for switching into interpret state and back from within a colon definition; this technique can be used to compile a literal (for an example, see Section 5.12.1 [Literals], page 89) or for conditional compilation (for an example, see Section 5.13.4 [Interpreter Directives], page 98).

5.13.4 Interpreter Directives

These words are usually used in interpret state; typically to control which parts of a source file are processed by the text interpreter. There are only a few ANS Forth Standard words, but Gforth supplements these with a rich set of immediate control structure words to compensate for the fact that the non-immediate versions can only be used in compile state (see Section 5.8 [Control Structures], page 64). Typical usages:

```
FALSE Constant HAVE-ASSEMBLER
.
.
HAVE-ASSEMBLER [IF]
: ASSEMBLER-FEATURE
  ...
;
[ENDIF]
.
.
: SEE
  ... \ general-purpose SEE code
  [ HAVE-ASSEMBLER [IF] ]
  ... \ assembler-specific SEE code
  [ [ENDIF] ]
;
```

`[IF]` *flag* – tools-ext "bracket-if"

If flag is `TRUE` do nothing (and therefore execute subsequent words as normal). If flag is `FALSE`, parse and discard words from the parse area (refilling it if necessary using `REFILL`) including nested instances of `[IF]`.. `[ELSE]`.. `[THEN]` and `[IF]`.. `[THEN]` until the balancing `[ELSE]` or `[THEN]` has been parsed and discarded. Immediate word.

`[ELSE]` – tools-ext "bracket-else"

Parse and discard words from the parse area (refilling it if necessary using `REFILL`) including nested instances of `[IF]`.. `[ELSE]`.. `[THEN]` and `[IF]`.. `[THEN]` until the balancing `[THEN]` has been parsed and discarded. `[ELSE]` only gets executed if the balancing `[IF]` was `TRUE`; if it was `FALSE`, `[IF]` would have parsed and discarded the `[ELSE]`, leaving the subsequent words to be executed as normal. Immediate word.

`[THEN]` – tools-ext "bracket-then"

Do nothing; used as a marker for other words to parse and discard up to. Immediate word.

`[ENDIF]` – gforth "bracket-end-if"

Do nothing; synonym for `[THEN]`

`[IFDEF]` "<*spaces*>*name*" – gforth "bracket-if-def"

If name is found in the current search-order, behave like `[IF]` with a `TRUE` flag, otherwise behave like `[IF]` with a `FALSE` flag. Immediate word.

`[IFUNDEF]` "<*spaces*>*name*" – gforth "bracket-if-un-def"

If name is not found in the current search-order, behave like `[IF]` with a `TRUE` flag, otherwise behave like `[IF]` with a `FALSE` flag. Immediate word.

`[?DO]` *n-limit n-index* – gforth "bracket-question-do"

`[DO]` *n-limit n-index* – gforth "bracket-do"

`[FOR]` *n* – gforth "bracket-for"

`[LOOP]` – gforth "bracket-loop"

`[+LOOP]` *n* – gforth "bracket-question-plus-loop"

`[NEXT]` *n* – gforth "bracket-next"

`[BEGIN]` – gforth "bracket-begin"

`[UNTIL]` *flag* – gforth "bracket-until"

`[AGAIN]` – gforth "bracket-again"

`[WHILE]` *flag* – gforth "bracket-while"

`[REPEAT]` – gforth "bracket-repeat"

5.14 The Input Stream

The text interpreter reads from the input stream, which can come from several sources (see Section 5.13.1 [Input Sources], page 95). Some words, in particular defining words, but also words like `'`, read parameters from the input stream instead of from the stack.

Such words are called parsing words, because they parse the input stream. Parsing words are hard to use in other words, because it is hard to pass program-generated parameters through the input stream. They also usually have an unintuitive combination of interpretation and compilation semantics when implemented naively, leading to various approaches that try to produce a more intuitive behaviour (see Section 5.10.1 [Combined words], page 85).

It should be obvious by now that parsing words are a bad idea. If you want to implement a parsing word for convenience, also provide a factor of the word that does not parse, but takes the parameters on the stack. To implement the parsing word on top if it, you can use the following words:

`parse` *char* "*ccc*<*char*>" – *c-addr u* core-ext "parse"

Parse *ccc*, delimited by *char*, in the parse area. *c-addr u* specifies the parsed string within the parse area. If the parse area was empty, *u* is 0.

`parse-word` "*name*" – *c-addr u* gforth "parse-word"

Get the next word from the input buffer

`name` – *c-addr u* gforth-obsolete "name"

old name for `parse-word`

`word` *char* "<*chars*>*ccc*<*char*>" – *c-addr* core "word"

Skip leading delimiters. Parse *ccc*, delimited by *char*, in the parse area. *c-addr* is the address of a transient region containing the parsed string in counted-string format. If the parse area was empty or contained no characters other than delimiters, the resulting string has zero length. A program may replace characters within the counted string. OBSOLESCENT: the counted string has a trailing space that is not included in its length.

`\"-parse` "*string*"<"> – *c-addr u* unknown "\"-parse"

parses string, translating \-escapes to characters (as in C). The resulting string resides at `here char+`. The supported `\-escapes` are: `\a` BEL (alert), `\b` BS, `\e` ESC (not in C99), `\f` FF, `\n` newline, `\r` CR, `\t` HT, `\v` VT, `\"` ", \[0-7]+ octal numerical character value, `\x`[0-9a-f]+ hex numerical character value; a \ before any other character represents that character (only ', \, ? in C99).

`refill` – *flag* core-ext,block-ext,file-ext "refill"

Attempt to fill the input buffer from the input source. When the input source is the user input device, attempt to receive input into the terminal input device. If successful, make the result the input buffer, set `>IN` to 0 and return true; otherwise return false. When the input source is a block, add 1 to the value of `BLK` to make the next block the input source and current input buffer, and set `>IN` to 0; return true if the new value of `BLK` is a valid block number, false otherwise. When the input source is a text file, attempt to read the next line from the file. If successful, make the result the current input buffer, set `>IN` to 0 and return true; otherwise, return false. A successful result includes receipt of a line containing 0 characters.

Conversely, if you have the bad luck (or lack of foresight) to have to deal with parsing words without having such factors, how do you pass a string that is not in the input stream to it?

`execute-parsing` *... addr u xt* – *...* unknown "execute-parsing"

Make *addr u* the current input source, execute *xt* `( ... -- ... )`, then restore the previous input source.

If you want to run a parsing word on a file, the following word should help:

`execute-parsing-file` *i*x fileid xt* – *j*x* unknown "execute-parsing-file"

Make *fileid* the current input source, execute *xt* `( i*x -- j*x )`, then restore the previous input source.

5.15 Word Lists

A wordlist is a list of named words; you can add new words and look up words by name (and you can remove words in a restricted way with markers). Every named (and `reveal`ed) word is in one wordlist.

The text interpreter searches the wordlists present in the search order (a stack of wordlists), from the top to the bottom. Within each wordlist, the search starts conceptually at the newest word; i.e., if two words in a wordlist have the same name, the newer word is found.

New words are added to the *compilation wordlist* (aka current wordlist).

A word list is identified by a cell-sized word list identifier (*wid*) in much the same way as a file is identified by a file handle. The numerical value of the wid has no (portable) meaning, and might change from session to session.

The ANS Forth "Search order" word set is intended to provide a set of low-level tools that allow various different schemes to be implemented. Gforth also provides `vocabulary`, a traditional Forth word. '`compat/vocabulary.fs`' provides an implementation in ANS Forth.

`forth-wordlist` – *wid* search "forth-wordlist"

`Constant` – *wid* identifies the word list that includes all of the standard words provided by Gforth. When Gforth is invoked, this word list is the compilation word list and is at the top of the search order.

`definitions` – search "definitions"

Set the compilation word list to be the same as the word list that is currently at the top of the search order.

`get-current` – *wid* search "get-current"

wid is the identifier of the current compilation word list.

`set-current` *wid* – search "set-current"

Set the compilation word list to the word list identified by *wid*.

`get-order` – *widn .. wid1 n* search "get-order"

Copy the search order to the data stack. The current search order has *n* entries, of which *wid1* represents the wordlist that is searched first (the word list at the top of the search order) and *widn* represents the wordlist that is searched last.

`set-order` *widn .. wid1 n* – search "set-order"

If *n*=0, empty the search order. If *n*=-1, set the search order to the implementation-defined minimum search order (for Gforth, this is the word list `Root`). Otherwise, replace the existing search order with the *n* wid entries such that *wid1* represents the word list that will be searched first and *widn* represents the word list that will be searched last.

`wordlist` – *wid* search "wordlist"

Create a new, empty word list represented by *wid*.

`table` – *wid* gforth "table"

Create a case-sensitive wordlist.

`>order` *wid* – gforth "to-order"

Push *wid* on the search order.

`previous` – search-ext "previous"

Drop the wordlist at the top of the search order.

`also` – search-ext "also"

Like `DUP` for the search order. Usually used before a vocabulary (e.g., `also Forth`); the combined effect is to push the wordlist represented by the vocabulary on the search order.

`Forth` – search-ext "Forth"

Replace the *wid* at the top of the search order with the *wid* associated with the word list `forth-wordlist`.

`Only` – search-ext "Only"

Set the search order to the implementation-defined minimum search order (for Gforth, this is the word list `Root`).

`order` – search-ext "order"

Print the search order and the compilation word list. The word lists are printed in the order in which they are searched (which is reversed with respect to the conventional way of displaying stacks). The compilation word list is displayed last.

`find` *c-addr* – *xt* `+-1` | *c-addr 0* core,search "find"

Search all word lists in the current search order for the definition named by the counted string at *c-addr*. If the definition is not found, return 0. If the definition is found return 1 (if the definition has non-default compilation semantics) or -1 (if the definition has default compilation semantics). The *xt* returned in interpret state represents the interpretation semantics. The *xt* returned in compile state represented either the compilation semantics (for non-default compilation semantics) or the run-time semantics that the compilation semantics would `compile,` (for default compilation semantics). The ANS Forth standard does not specify clearly what the returned *xt* represents (and also talks about immediacy instead of non-default compilation semantics), so this word is questionable in portable programs. If non-portability is ok, `find-name` and friends are better (see Section 5.11.3 [Name token], page 89).

`search-wordlist` *c-addr count wid* – *0* | *xt* `+-1` search "search-wordlist"

Search the word list identified by *wid* for the definition named by the string at *c-addr count*. If the definition is not found, return 0. If the definition is found return 1 (if the definition is immediate) or -1 (if the definition is not immediate) together with the *xt*. In Gforth, the *xt* returned represents the interpretation semantics. ANS Forth does not specify clearly what *xt* represents.

`words` – tools "words"

Display a list of all of the definitions in the word list at the top of the search order.

`vlist` – gforth "vlist"

Old (pre-Forth-83) name for `WORDS`.

`Root` – gforth "Root"

Add the root wordlist to the search order stack. This vocabulary makes up the minimum search order and contains only a search-order words.

`Vocabulary` "*name*" – gforth "Vocabulary"

Create a definition "name" and associate a new word list with it. The run-time effect of "name" is to replace the *wid* at the top of the search order with the *wid* associated with the new word list.

`seal` – gforth "seal"

Remove all word lists from the search order stack other than the word list that is currently on the top of the search order stack.

`vocs` – gforth "vocs"

List vocabularies and wordlists defined in the system.

`current` – *addr* gforth "current"

`Variable` – holds the *wid* of the compilation word list.

`context` – *addr* gforth "context"

`context @` is the *wid* of the word list at the top of the search order.

5.15.1 Vocabularies

Here is an example of creating and using a new wordlist using ANS Forth words:

```
wordlist constant my-new-words-wordlist
: my-new-words get-order nip my-new-words-wordlist swap set-order ;

\ add it to the search order
also my-new-words

\ alternatively, add it to the search order and make it
\ the compilation word list
also my-new-words definitions
\ type "order" to see the problem
```

The problem with this example is that `order` has no way to associate the name `my-new-words` with the wid of the word list (in Gforth, `order` and `vocs` will display `???` for a wid that has no associated name). There is no Standard way of associating a name with a wid.

In Gforth, this example can be re-coded using `vocabulary`, which associates a name with a wid:

```
vocabulary my-new-words

\ add it to the search order
also my-new-words

\ alternatively, add it to the search order and make it
\ the compilation word list
my-new-words definitions
\ type "order" to see that the problem is solved
```

5.15.2 Why use word lists?

Here are some reasons why people use wordlists:

- To prevent a set of words from being used outside the context in which they are valid. Two classic examples of this are an integrated editor (all of the edit commands are defined in a separate word list; the search order is set to the editor word list when the editor is invoked; the old search order is restored when the editor is terminated) and an integrated assembler (the op-codes for the machine are defined in a separate word list which is used when a `CODE` word is defined).
- To organize the words of an application or library into a user-visible set (in `forth-wordlist` or some other common wordlist) and a set of helper words used just for the implementation (hidden in a separate wordlist). This keeps `words`' output smaller, separates implementation and interface, and reduces the chance of name conflicts within the common wordlist.
- To prevent a name-space clash between multiple definitions with the same name. For example, when building a cross-compiler you might have a word `IF` that generates

conditional code for your target system. By placing this definition in a different word list you can control whether the host system's `IF` or the target system's `IF` get used in any particular context by controlling the order of the word lists on the search order stack.

The downsides of using wordlists are:

Debugging becomes more cumbersome.

Name conflicts worked around with wordlists are still there, and you have to arrange the search order carefully to get the desired results; if you forget to do that, you get hard-to-find errors (as in any case where you read the code differently from the compiler; `see` can help seeing which of several possible words the name resolves to in such cases). `See` displays just the name of the words, not what wordlist they belong to, so it might be misleading. Using unique names is a better approach to avoid name conflicts.

You have to explicitly undo any changes to the search order. In many cases it would be more convenient if this happened implicitly. Gforth currently does not provide such a feature, but it may do so in the future.

5.15.3 Word list example

The following example is from the garbage collector (`http://www.complang.tuwien.ac.at/forth/garbage-collection.zip`) and uses wordlists to separate public words from helper words:

```
get-current ( wid )
vocabulary garbage-collector also garbage-collector definitions
... \ define helper words
( wid ) set-current \ restore original (i.e., public) compilation wordlist
... \ define the public (i.e., API) words
    \ they can refer to the helper words
previous \ restore original search order (helper words become invisible)
```

5.16 Environmental Queries

ANS Forth introduced the idea of "environmental queries" as a way for a program running on a system to determine certain characteristics of the system. The Standard specifies a number of strings that might be recognised by a system.

The Standard requires that the header space used for environmental queries be distinct from the header space used for definitions.

Typically, environmental queries are supported by creating a set of definitions in a word list that is *only* used during environmental queries; that is what Gforth does. There is no Standard way of adding definitions to the set of recognised environmental queries, but any implementation that supports the loading of optional word sets must have some mechanism for doing this (after loading the word set, the associated environmental query string must return `true`). In Gforth, the word list used to honour environmental queries can be manipulated just like any other word list.

`environment?` *c-addr u – false / ... true* core "environment-query"

c-addr, u specify a counted string. If the string is not recognised, return a `false` flag. Otherwise return a `true` flag and some (string-specific) information about the queried string.

`environment-wordlist` – *wid* gforth "environment-wordlist"

wid identifies the word list that is searched by environmental queries.

`gforth` – *c-addr u* gforth-environment "gforth"

Counted string representing a version string for this version of Gforth (for versions>0.3.0). The version strings of the various versions are guaranteed to be ordered lexicographically.

`os-class` – *c-addr u* gforth-environment "os-class"

Counted string representing a description of the host operating system.

Note that, whilst the documentation for (e.g.) `gforth` shows it returning two items on the stack, querying it using `environment?` will return an additional item; the `true` flag that shows that the string was recognised.

Here are some examples of using environmental queries:

```
s" address-unit-bits" environment? 0=
[IF]
     cr .( environmental attribute address-units-bits unknown... ) cr
[ELSE]
     drop \ ensure balanced stack effect
[THEN]

\ this might occur in the prelude of a standard program that uses THROW
s" exception" environment? [IF]
   0= [IF]
      : throw abort" exception thrown" ;
   [THEN]
[ELSE] \ we don't know, so make sure
   : throw abort" exception thrown" ;
[THEN]

s" gforth" environment? [IF] .( Gforth version ) TYPE
                        [ELSE] .( Not Gforth..) [THEN]

\ a program using v*
s" gforth" environment? [IF]
  s" 0.5.0" compare 0< [IF] \ v* is a primitive since 0.5.0
   : v* ( f_addr1 nstride1 f_addr2 nstride2 ucount -- r )
     >r swap 2swap swap 0e r> 0 ?DO
       dup f over + 2swap dup f f* f+ over + 2swap
     LOOP
     2drop 2drop ;
  [THEN]
[ELSE] \
  : v* ( f_addr1 nstride1 f_addr2 nstride2 ucount -- r )
  ...
[THEN]
```

Here is an example of adding a definition to the environment word list:

```
get-current environment-wordlist set-current
true constant block
true constant block-ext
```

```
set-current
```

You can see what definitions are in the environment word list like this:

```
environment-wordlist >order words previous
```

5.17 Files

Gforth provides facilities for accessing files that are stored in the host operating system's file-system. Files that are processed by Gforth can be divided into two categories:

- Files that are processed by the Text Interpreter (*Forth source files*).
- Files that are processed by some other program (*general files*).

5.17.1 Forth source files

The simplest way to interpret the contents of a file is to use one of these two formats:

```
include mysource.fs
s" mysource.fs" included
```

You usually want to include a file only if it is not included already (by, say, another source file). In that case, you can use one of these three formats:

```
require mysource.fs
needs mysource.fs
s" mysource.fs" required
```

It is good practice to write your source files such that interpreting them does not change the stack. Source files designed in this way can be used with `required` and friends without complications. For example:

```
1024 require foo.fs drop
```

Here you want to pass the argument 1024 (e.g., a buffer size) to '`foo.fs`'. Interpreting '`foo.fs`' has the stack effect (n – n), which allows its use with `require`. Of course with such parameters to required files, you have to ensure that the first `require` fits for all uses (i.e., `require` it early in the master load file).

`include-file` *i*x wfileid – j*x* unknown "include-file"

Interpret (process using the text interpreter) the contents of the file *wfileid*.

`included` *i*x c-addr u – j*x* file "included"

`include-file` the file whose name is given by the string *c-addr u*.

`included?` *c-addr u – f* gforth "included?"

True only if the file *c-addr u* is in the list of earlier included files. If the file has been loaded, it may have been specified as, say, '`foo.fs`' and found somewhere on the Forth search path. To return `true` from `included?`, you must specify the exact path to the file, even if that is '`./foo.fs`'

`include` *... "file" – ...* gforth "include"

`include-file` the file *file*.

`required` *i*x addr u – i*x* gforth "required"

`include-file` the file with the name given by *addr u*, if it is not `included` (or `required`) already. Currently this works by comparing the name of the file (with path) against the names of earlier included files.

`require` ... "*file*" – ... gforth "require"

`include-file` *file* only if it is not included already.

`needs` ... "*name*" – ... gforth "needs"

An alias for `require`; exists on other systems (e.g., Win32Forth).

`sourcefilename` – *c-addr u* gforth "sourcefilename"

The name of the source file which is currently the input source. The result is valid only while the file is being loaded. If the current input source is no (stream) file, the result is undefined. In Gforth, the result is valid during the whole seesion (but not across `savesystem` etc.).

`sourceline#` – *u* gforth "sourceline-number"

The line number of the line that is currently being interpreted from a (stream) file. The first line has the number 1. If the current input source is not a (stream) file, the result is undefined.

A definition in ANS Forth for `required` is provided in '`compat/required.fs`'.

5.17.2 General files

Files are opened/created by name and type. The following file access methods (FAMs) are recognised:

`r/o` – *fam* file "r-o"

`r/w` – *fam* file "r-w"

`w/o` – *fam* file "w-o"

`bin` *fam1* – *fam2* file "bin"

When a file is opened/created, it returns a file identifier, *wfileid* that is used for all other file commands. All file commands also return a status value, *wior*, that is 0 for a successful operation and an implementation-defined non-zero value in the case of an error.

`open-file` *c-addr u wfam* – *wfileid wior* file "open-file"

`create-file` *c-addr u wfam* – *wfileid wior* file "create-file"

`close-file` *wfileid* – *wior* file "close-file"

`delete-file` *c-addr u* – *wior* file "delete-file"

`rename-file` *c-addr1 u1 c-addr2 u2* – *wior* file-ext "rename-file"

Rename file *c_addr1 u1* to new name *c_addr2 u2*

`read-file` *c-addr u1 wfileid* – *u2 wior* file "read-file"

`read-line` *c_addr u1 wfileid* – *u2 flag wior* unknown "read-line"

`write-file` *c-addr u1 wfileid* – *wior* file "write-file"

`write-line` *c-addr u fileid* – *ior* file "write-line"

`emit-file` *c wfileid* – *wior* gforth "emit-file"

`flush-file` *wfileid* – *wior* file-ext "flush-file"

`file-status` *c-addr u* – *wfam wior* file-ext "file-status"

`file-position` *wfileid* – *ud wior* file "file-position"

`reposition-file` *ud wfileid* – *wior* file "reposition-file"

`file-size` *wfileid* – *ud wior* file "file-size" `resize-file` *ud wfileid – wior* file "resize-file"

`slurp-file` *c-addr1 u1 – c-addr2 u2* unknown "slurp-file"

c-addr1 u1 is the filename, *c-addr2 u2* is the file's contents

`slurp-fid` unknown "slurp-fid"

`stdin` *– wfileid* gforth "stdin"

`stdout` *– wfileid* gforth "stdout"

`stderr` *– wfileid* gforth "stderr"

5.17.3 Search Paths

If you specify an absolute filename (i.e., a filename starting with '`/`' or '`~`', or with '`:`' in the second position (as in '`C:...`')) for `included` and friends, that file is included just as you would expect.

If the filename starts with '`./`', this refers to the directory that the present file was `included` from. This allows files to include other files relative to their own position (irrespective of the current working directory or the absolute position). This feature is essential for libraries consisting of several files, where a file may include other files from the library. It corresponds to `#include "..."` in C. If the current input source is not a file, '`.`' refers to the directory of the innermost file being included, or, if there is no file being included, to the current working directory.

For relative filenames (not starting with '`./`'), Gforth uses a search path similar to Forth's search order (see Section 5.15 [Word Lists], page 100). It tries to find the given filename in the directories present in the path, and includes the first one it finds. There are separate search paths for Forth source files and general files. If the search path contains the directory '`.`', this refers to the directory of the current file, or the working directory, as if the file had been specified with '`./`'.

Use '`~+`' to refer to the current working directory (as in the `bash`).

5.17.3.1 Source Search Paths

The search path is initialized when you start Gforth (see Section 2.1 [Invoking Gforth], page 3). You can display it and change it using `fpath` in combination with the general path handling words.

`fpath` *– path-addr* gforth "fpath"

Here is an example of using `fpath` and `require`:

```
fpath path= /usr/lib/forth/|./
require timer.fs
```

5.17.3.2 General Search Paths

Your application may need to search files in several directories, like `included` does. To facilitate this, Gforth allows you to define and use your own search paths, by providing generic equivalents of the Forth search path words:

`open-path-file` *addr1 u1 path-addr – wfileid addr2 u2 0 | ior* gforth "open-path-file"

Look in path *path-addr* for the file specified by *addr1 u1*. If found, the resulting path and and (read-only) open file descriptor are returned. If the file is not found, *ior* is non-zero.

`path-allot` *umax –* unknown "path-allot"

`Allot` a path with *umax* characters capacity, initially empty.

`clear-path` *path-addr –* gforth "clear-path"

Set the path *path-addr* to empty.

`also-path` *c-addr len path-addr –* gforth "also-path"

add the directory *c-addr len* to *path-addr*.

`.path` *path-addr –* gforth ".path"

Display the contents of the search path *path-addr*.

`path+` *path-addr* "*dir*" – gforth "path+"

Add the directory *dir* to the search path *path-addr*.

`path=` *path-addr* "*dir1|dir2|dir3*" gforth "path="

Make a complete new search path; the path separator is `|`.

Here's an example of creating an empty search path:

```
create mypath 500 path-allot \ maximum length 500 chars (is checked)
```

5.18 Blocks

When you run Gforth on a modern desk-top computer, it runs under the control of an operating system which provides certain services. One of these services is *file services*, which allows Forth source code and data to be stored in files and read into Gforth (see Section 5.17 [Files], page 106).

Traditionally, Forth has been an important programming language on systems where it has interfaced directly to the underlying hardware with no intervening operating system. Forth provides a mechanism, called *blocks*, for accessing mass storage on such systems.

A block is a 1024-byte data area, which can be used to hold data or Forth source code. No structure is imposed on the contents of the block. A block is identified by its number; blocks are numbered contiguously from 1 to an implementation-defined maximum.

A typical system that used blocks but no operating system might use a single floppy-disk drive for mass storage, with the disks formatted to provide 256-byte sectors. Blocks would be implemented by assigning the first four sectors of the disk to block 1, the second four sectors to block 2 and so on, up to the limit of the capacity of the disk. The disk would not contain any file system information, just the set of blocks.

On systems that do provide file services, blocks are typically implemented by storing a sequence of blocks within a single *blocks file*. The size of the blocks file will be an exact multiple of 1024 bytes, corresponding to the number of blocks it contains. This is the mechanism that Gforth uses.

Only one blocks file can be open at a time. If you use block words without having specified a blocks file, Gforth defaults to the blocks file '`blocks.fb`'. Gforth uses the Forth search path when attempting to locate a blocks file (see Section 5.17.3.1 [Source Search Paths], page 108).

When you read and write blocks under program control, Gforth uses a number of *block buffers* as intermediate storage. These buffers are not used when you use `load` to interpret the contents of a block.

The behaviour of the block buffers is analagous to that of a cache. Each block buffer has three states:

- Unassigned
- Assigned-clean
- Assigned-dirty

Initially, all block buffers are *unassigned*. In order to access a block, the block (specified by its block number) must be assigned to a block buffer.

The assignment of a block to a block buffer is performed by `block` or `buffer`. Use `block` when you wish to modify the existing contents of a block. Use `buffer` when you don't care about the existing contents of the block[22].

Once a block has been assigned to a block buffer using `block` or `buffer`, that block buffer becomes the *current block buffer*. Data may only be manipulated (read or written) within the current block buffer.

When the contents of the current block buffer has been modified it is necessary, *before calling* `block` *or* `buffer` *again*, to either abandon the changes (by doing nothing) or mark the block as changed (assigned-dirty), using `update`. Using `update` does not change the blocks file; it simply changes a block buffer's state to *assigned-dirty*. The block will be written implicitly when it's buffer is needed for another block, or explicitly by `flush` or `save-buffers`.

word `Flush` writes all *assigned-dirty* blocks back to the blocks file on disk. Leaving Gforth with `bye` also performs a `flush`.

In Gforth, `block` and `buffer` use a *direct-mapped* algorithm to assign a block buffer to a block. That means that any particular block can only be assigned to one specific block buffer, called (for the particular operation) the *victim buffer*. If the victim buffer is *unassigned* or *assigned-clean* it is allocated to the new block immediately. If it is *assigned-dirty* its current contents are written back to the blocks file on disk before it is allocated to the new block.

Although no structure is imposed on the contents of a block, it is traditional to display the contents as 16 lines each of 64 characters. A block provides a single, continuous stream of input (for example, it acts as a single parse area) – there are no end-of-line characters within a block, and no end-of-file character at the end of a block. There are two consequences of this:

- The last character of one line wraps straight into the first character of the following line

[22] The ANS Forth definition of `buffer` is intended not to cause disk I/O; if the data associated with the particular block is already stored in a block buffer due to an earlier `block` command, `buffer` will return that block buffer and the existing contents of the block will be available. Otherwise, `buffer` will simply assign a new, empty block buffer for the block.

- The word `\` – comment to end of line – requires special treatment; in the context of a block it causes all characters until the end of the current 64-character "line" to be ignored.

In Gforth, when you use `block` with a non-existent block number, the current blocks file will be extended to the appropriate size and the block buffer will be initialised with spaces.

Gforth includes a simple block editor (type `use blocked.fb 0 list` for details) but doesn't encourage the use of blocks; the mechanism is only provided for backward compatibility – ANS Forth requires blocks to be available when files are.

Common techniques that are used when working with blocks include:

- A screen editor that allows you to edit blocks without leaving the Forth environment.
- Shadow screens; where every code block has an associated block containing comments (for example: code in odd block numbers, comments in even block numbers). Typically, the block editor provides a convenient mechanism to toggle between code and comments.
- Load blocks; a single block (typically block 1) contains a number of `thru` commands which `load` the whole of the application.

See Frank Sergeant's Pygmy Forth to see just how well blocks can be integrated into a Forth programming environment.

`open-blocks` *c-addr u* – gforth "open-blocks"

Use the file, whose name is given by *c-addr u*, as the blocks file.

`use` `"`*file*`"` – gforth "use"

Use *file* as the blocks file.

`block-offset` – *addr* gforth "block-offset"

User variable containing the number of the first block (default since 0.5.0: 0). Block files created with Gforth versions before 0.5.0 have the offset 1. If you use these files you can: `1 offset !`; or add 1 to every block number used; or prepend 1024 characters to the file.

`get-block-fid` – *wfileid* gforth "get-block-fid"

Return the file-id of the current blocks file. If no blocks file has been opened, use '`blocks.fb`' as the default blocks file.

`block-position` *u* – block "block-position"

Position the block file to the start of block *u*.

`list` *u* – block-ext "list"

Display block *u*. In Gforth, the block is displayed as 16 numbered lines, each of 64 characters.

`scr` – *a-addr* block-ext "s-c-r"

`User` variable – *a-addr* is the address of a cell containing the block number of the block most recently processed by `list`.

`block` *u* – *a-addr* block "block"

If a block buffer is assigned for block *u*, return its start address, *a-addr*. Otherwise, assign a block buffer for block *u* (if the assigned block buffer has been `update`d, transfer the contents to mass storage), read the block into the block buffer and return its start address, *a-addr*.

`buffer` *u – a-addr* block "buffer"

If a block buffer is assigned for block *u*, return its start address, *a-addr*. Otherwise, assign a block buffer for block *u* (if the assigned block buffer has been `update`d, transfer the contents to mass storage) and return its start address, *a-addr*. The subtle difference between `buffer` and `block` mean that you should only use `buffer` if you don't care about the previous contents of block *u*. In Gforth, this simply calls `block`.

`empty-buffers` – block-ext "empty-buffers"

Mark all block buffers as unassigned; if any had been marked as assigned-dirty (by `update`), the changes to those blocks will be lost.

`empty-buffer` *buffer –* gforth "empty-buffer"

`update` – block "update"

Mark the state of the current block buffer as assigned-dirty.

`updated?` *n – f* gforth "updated?"

Return true if `updated` has been used to mark block *n* as assigned-dirty.

`save-buffers` – block "save-buffers"

Transfer the contents of each `update`d block buffer to mass storage, then mark all block buffers as assigned-clean.

`save-buffer` *buffer –* gforth "save-buffer"

`flush` – block "flush"

Perform the functions of `save-buffers` then `empty-buffers`.

`load` *i*x n – j*x* block "load"

Save the current input source specification. Store *n* in `BLK`, set `>IN` to 0 and interpret. When the parse area is exhausted, restore the input source specification.

`thru` *i*x n1 n2 – j*x* block-ext "thru"

`load` the blocks *n1* through *n2* in sequence.

`+load` *i*x n – j*x* gforth "+load"

Used within a block to load the block specified as the current block `+` *n*.

`+thru` *i*x n1 n2 – j*x* gforth "+thru"

Used within a block to load the range of blocks specified as the current block `+` *n1* thru the current block `+` *n2*.

`-->` – gforth "chain"

If this symbol is encountered whilst loading block *n*, discard the remainder of the block and load block *n+1*. Used for chaining multiple blocks together as a single loadable unit. Not recommended, because it destroys the independence of loading. Use `thru` (which is standard) or `+thru` instead.

`block-included` *a-addr u –* gforth "block-included"

Use within a block that is to be processed by `load`. Save the current blocks file specification, open the blocks file specified by *a-addr u* and `load` block 1 from that file (which may in turn chain or load other blocks). Finally, close the blocks file and restore the original blocks file.

5.19 Other I/O

5.19.1 Simple numeric output

The simplest output functions are those that display numbers from the data or floating-point stacks. Floating-point output is always displayed using base 10. Numbers displayed from the data stack use the value stored in `base`.

`.` *n* – core "dot"

Display (the signed single number) *n* in free-format, followed by a space.

`dec.` *n* – gforth "dec."

Display *n* as a signed decimal number, followed by a space.

`hex.` *u* – gforth "hex."

Display *u* as an unsigned hex number, prefixed with a `"$"` and followed by a space.

`u.` *u* – core "u-dot"

Display (the unsigned single number) *u* in free-format, followed by a space.

`.r` *n1 n2* – core-ext "dot-r"

Display *n1* right-aligned in a field *n2* characters wide. If more than *n2* characters are needed to display the number, all digits are displayed. If appropriate, *n2* must include a character for a leading "-".

`u.r` *u n* – core-ext "u-dot-r"

Display *u* right-aligned in a field *n* characters wide. If more than *n* characters are needed to display the number, all digits are displayed.

`d.` *d* – double "d-dot"

Display (the signed double number) *d* in free-format. followed by a space.

`ud.` *ud* – gforth "u-d-dot"

Display (the signed double number) *ud* in free-format, followed by a space.

`d.r` *d n* – double "d-dot-r"

Display *d* right-aligned in a field *n* characters wide. If more than *n* characters are needed to display the number, all digits are displayed. If appropriate, *n* must include a character for a leading "-".

`ud.r` *ud n* – gforth "u-d-dot-r"

Display *ud* right-aligned in a field *n* characters wide. If more than *n* characters are needed to display the number, all digits are displayed.

`f.` *r* – float-ext "f-dot"

Display (the floating-point number) *r* without exponent, followed by a space.

`fe.` *r* – float-ext "f-e-dot"

Display *r* using engineering notation (with exponent dividable by 3), followed by a space.

`fs.` *r* – float-ext "f-s-dot"

Display *r* using scientific notation (with exponent), followed by a space.

`f.rdp` *rf +nr +nd +np* – gforth "f.rdp"

Print float *rf* formatted. The total width of the output is *nr*. For fixed-point notation, the number of digits after the decimal point is +*nd* and the minimum number of significant digits is *np*. `Set-precision` has no effect on `f.rdp`. Fixed-point notation is used if the number of siginicant digits would be at least *np* and if the number of digits before the decimal point would fit. If fixed-point notation is not used, exponential notation is used, and if that does not fit, asterisks are printed. We recommend using *nr*>=7 to avoid the risk of numbers not fitting at all. We recommend *nr*>=*np*+5 to avoid cases where `f.rdp` switches to exponential notation because fixed-point notation would have too few significant digits, yet exponential notation offers fewer significant digits. We recommend *nr*>=*nd*+2, if you want to have fixed-point notation for some numbers. We recommend *np*>*nr*, if you want to have exponential notation for all numbers.

Examples of printing the number 1234.5678E23 in the different floating-point output formats are shown below:

```
f. 123456779999999000000000000.
fe. 123.456779999999E24
fs. 1.23456779999999E26
```

5.19.2 Formatted numeric output

Forth traditionally uses a technique called *pictured numeric output* for formatted printing of integers. In this technique, digits are extracted from the number (using the current output radix defined by `base`), converted to ASCII codes and appended to a string that is built in a scratch-pad area of memory (see Section 8.1.1 [Implementation-defined options], page 167). Arbitrary characters can be appended to the string during the extraction process. The completed string is specified by an address and length and can be manipulated (`TYPE`ed, copied, modified) under program control.

All of the integer output words described in the previous section (see Section 5.19.1 [Simple numeric output], page 113) are implemented in Gforth using pictured numeric output.

Three important things to remember about pictured numeric output:

- It always operates on double-precision numbers; to display a single-precision number, convert it first (for ways of doing this see Section 5.5.2 [Double precision], page 51).
- It always treats the double-precision number as though it were unsigned. The examples below show ways of printing signed numbers.
- The string is built up from right to left; least significant digit first.

`<#` – core "less-number-sign"

Initialise/clear the pictured numeric output string.

`<<#` – gforth "less-less-number-sign"

Start a hold area that ends with `#>>`. Can be nested in each other and in `<#`. Note: if you do not match up the `<<#`s with `#>>`s, you will eventually run out of hold area; you can reset the hold area to empty with `<#`.

`#` *ud1* – *ud2* core "number-sign"

Used within `<#` and `#>`. Add the next least-significant digit to the pictured numeric output string. This is achieved by dividing *ud1* by the number in `base` to leave quotient *ud2* and remainder *n*; *n* is converted to the appropriate display code (eg ASCII code) and appended to the string. If the number has been fully converted, *ud1* will be 0 and `#` will append a "0" to the string.

`#s` *ud – 0 0* core "number-sign-s"

Used within `<#` and `#>`. Convert all remaining digits using the same algorithm as for `#`. `#s` will convert at least one digit. Therefore, if *ud* is 0, `#s` will append a "0" to the pictured numeric output string.

`hold` *char –* core "hold"

Used within `<#` and `#>`. Append the character *char* to the pictured numeric output string.

`sign` *n –* core "sign"

Used within `<#` and `#>`. If *n* (a *single* number) is negative, append the display code for a minus sign to the pictured numeric output string. Since the string is built up "backwards" this is usually used immediately prior to `#>`, as shown in the examples below.

`#>` *xd – addr u* core "number-sign-greater"

Complete the pictured numeric output string by discarding *xd* and returning *addr u*; the address and length of the formatted string. A Standard program may modify characters within the string.

`#>>` – gforth "number-sign-greater-greater"

Release the hold area started with `<<#`.

`represent` *r c-addr u – n f1 f2* float "represent"

`f>str-rdp` *rf +nr +nd +np – c-addr nr* gforth "f>str-rdp"

Convert *rf* into a string at *c-addr nr*. The conversion rules and the meanings of *nr +nd np* are the same as for `f.rdp`. The result in in the pictured numeric output buffer and will be destroyed by anything destroying that buffer.

doc-f>buf-rdp

Here are some examples of using pictured numeric output:

```
: my-u. ( u -- )
  \ Simplest use of pns.. behaves like Standard u.
  0              \ convert to unsigned double
  <<#            \ start conversion
  #s             \ convert all digits
  #>             \ complete conversion
  TYPE SPACE     \ display, with trailing space
  #>> ;          \ release hold area

: cents-only ( u -- )
  0              \ convert to unsigned double
  <<#            \ start conversion
  # #            \ convert two least-significant digits
  #>             \ complete conversion, discard other digits
  TYPE SPACE     \ display, with trailing space
```

```
  #>> ;          \ release hold area

: dollars-and-cents ( u -- )
  0              \ convert to unsigned double
  <<#            \ start conversion
  # #            \ convert two least-significant digits
  [char] . hold  \ insert decimal point
  #s             \ convert remaining digits
  [char] $ hold  \ append currency symbol
  #>             \ complete conversion
  TYPE SPACE     \ display, with trailing space
  #>> ;          \ release hold area

: my-. ( n -- )
  \ handling negatives.. behaves like Standard .
  s>d            \ convert to signed double
  swap over dabs \ leave sign byte followed by unsigned double
  <<#            \ start conversion
  #s             \ convert all digits
  rot sign       \ get at sign byte, append "-" if needed
  #>             \ complete conversion
  TYPE SPACE     \ display, with trailing space
  #>> ;          \ release hold area

: account. ( n -- )
  \ accountants don't like minus signs, they use parentheses
  \ for negative numbers
  s>d            \ convert to signed double
  swap over dabs \ leave sign byte followed by unsigned double
  <<#            \ start conversion
  2 pick         \ get copy of sign byte
  0< IF [char] ) hold THEN \ right-most character of output
  #s             \ convert all digits
  rot            \ get at sign byte
  0< IF [char] ( hold THEN
  #>             \ complete conversion
  TYPE SPACE     \ display, with trailing space
  #>> ;          \ release hold area
```

Here are some examples of using these words:

```
1 my-u. 1
hex -1 my-u. decimal FFFFFFFF
1 cents-only 01
1234 cents-only 34
2 dollars-and-cents $0.02
1234 dollars-and-cents $12.34
123 my-. 123
-123 my. -123
123 account. 123
```

```
-456 account. (456)
```

5.19.3 String Formats

Forth commonly uses two different methods for representing character strings:

- As a *counted string*, represented by a *c-addr*. The char addressed by *c-addr* contains a character-count, *n*, of the string and the string occupies the subsequent *n* char addresses in memory.
- As cell pair on the stack; *c-addr u*, where *u* is the length of the string in characters, and *c-addr* is the address of the first byte of the string.

ANS Forth encourages the use of the second format when representing strings.

`count` *c-addr1 – c-addr2 u* core "count"

c-addr2 is the first character and *u* the length of the counted string at *c-addr1*.

For words that move, copy and search for strings see Section 5.7.6 [Memory Blocks], page 63. For words that display characters and strings see Section 5.19.4 [Displaying characters and strings], page 117.

5.19.4 Displaying characters and strings

This section starts with a glossary of Forth words and ends with a set of examples.

`bl` *– c-char* core "b-l"

c-char is the character value for a space.

`space` *–* core "space"

Display one space.

`spaces` *u –* core "spaces"

Display *n* spaces.

`emit` *c –* core "emit"

Display the character associated with character value c.

`toupper` *c1 – c2* gforth "toupper"

If *c1* is a lower-case character (in the current locale), *c2* is the equivalent upper-case character. All other characters are unchanged.

`."` *compilation 'ccc"' – ; run-time –* core "dot-quote"

Compilation: Parse a string *ccc* delimited by a `"` (double quote). At run-time, display the string. Interpretation semantics for this word are undefined in ANS Forth. Gforth's interpretation semantics are to display the string. This is the simplest way to display a string from within a definition; see examples below.

`.(` *compilation&interpretation* `"`*ccc*<*paren*>`"` *–* core-ext "dot-paren"

Compilation and interpretation semantics: Parse a string *ccc* delimited by a `)` (right parenthesis). Display the string. This is often used to display progress information during compilation; see examples below.

`.\"` *compilation 'ccc"' – ; run-time –* gforth "dot-backslash-quote"

`type` *c-addr u –* core "type"

If *u*>0, display *u* characters from a string starting with the character stored at *c-addr*.

`typewhite` *addr n* – gforth "typewhite"

Like type, but white space is printed instead of the characters.

`cr` – core "c-r"

Output a newline (of the favourite kind of the host OS). Note that due to the way the Forth command line interpreter inserts newlines, the preferred way to use `cr` is at the start of a piece of text; e.g., `cr ." hello, world"`.

`at-xy` *u1 u2* – facility "at-x-y"

Position the cursor so that subsequent text output will take place at column *u1*, row *u2* of the display. (column 0, row 0 is the top left-hand corner of the display).

`page` – facility "page"

Clear the display and set the cursor to the top left-hand corner.

`S"` *compilation 'ccc"' – ; run-time – c-addr u* core,file "s-quote"

Compilation: Parse a string *ccc* delimited by a `"` (double quote). At run-time, return the length, *u*, and the start address, *c-addr* of the string. Interpretation: parse the string as before, and return *c-addr*, *u*. Gforth `allocate`s the string. The resulting memory leak is usually not a problem; the exception is if you create strings containing `S"` and `evaluate` them; then the leak is not bounded by the size of the interpreted files and you may want to `free` the strings. ANS Forth only guarantees one buffer of 80 characters, so in standard programs you should assume that the string lives only until the next `s"`.

`s\"` *compilation 'ccc"' – ; run-time – c-addr u* gforth "s-backslash-quote"

Like `S"`, but translates C-like \-escape-sequences into single characters. See `\"-parse` for details.

`C"` *compilation "ccc<quote>" – ; run-time – c-addr* core-ext "c-quote"

Compilation: parse a string *ccc* delimited by a `"` (double quote). At run-time, return *c-addr* which specifies the counted string *ccc*. Interpretation semantics are undefined.

`char` *'<spaces>ccc' – c* core "char"

Skip leading spaces. Parse the string *ccc* and return *c*, the display code representing the first character of *ccc*.

`[Char]` *compilation '<spaces>ccc' – ; run-time – c* core "bracket-char"

Compilation: skip leading spaces. Parse the string *ccc*. Run-time: return *c*, the display code representing the first character of *ccc*. Interpretation semantics for this word are undefined.

As an example, consider the following text, stored in a file '`test.fs`':

```
.( text-1)
: my-word
  ." text-2" cr
  .( text-3)
;

." text-4"

: my-char
  [char] ALPHABET emit
  char emit
```

```
;
```

When you load this code into Gforth, the following output is generated:

```
include test.fs ⟨RET⟩ text-1text-3text-4 ok
```

- Messages `text-1` and `text-3` are displayed because `.(` is an immediate word; it behaves in the same way whether it is used inside or outside a colon definition.
- Message `text-4` is displayed because of Gforth's added interpretation semantics for `."`.
- Message `text-2` is *not* displayed, because the text interpreter performs the compilation semantics for `."` within the definition of `my-word`.

Here are some examples of executing `my-word` and `my-char`:

```
my-word ⟨RET⟩ text-2
 ok
my-char fred ⟨RET⟩ Af ok
my-char jim ⟨RET⟩ Aj ok
```

- Message `text-2` is displayed because of the run-time behaviour of `."`.
- `[char]` compiles the "A" from "ALPHABET" and puts its display code on the stack at run-time. `emit` always displays the character when `my-char` is executed.
- `char` parses a string at run-time and the second `emit` displays the first character of the string.
- If you type `see my-char` you can see that `[char]` discarded the text "LPHABET" and only compiled the display code for "A" into the definition of `my-char`.

5.19.5 Input

For ways of storing character strings in memory see Section 5.19.3 [String Formats], page 117.

`key` – *char* core "key"

Receive (but do not display) one character, *char*.

`key?` – *flag* facility "key-question"

Determine whether a character is available. If a character is available, *flag* is true; the next call to `key` will yield the character. Once `key?` returns true, subsequent calls to `key?` before calling `key` or `ekey` will also return true.

`ekey` – *u* facility-ext "e-key"

`ekey?` – *flag* unknown "ekey?"

`ekey>char` *u* – *u false* | *c true* facility-ext "e-key-to-char"

`>number` *ud1 c-addr1 u1* – *ud2 c-addr2 u2* core "to-number"

Attempt to convert the character string *c-addr1 u1* to an unsigned number in the current number base. The double *ud1* accumulates the result of the conversion to form *ud2*. Conversion continues, left-to-right, until the whole string is converted or a character that is not convertable in the current number base is encountered (including `+` or `-`). For each convertable character, *ud1* is first multiplied by the value in `BASE` and then incremented by the value represented by the character. *c-addr2* is the location of the first unconverted character (past the end of the string if the whole string was converted). *u2* is the number of unconverted characters in the string. Overflow is not detected.

`>float` *c-addr u* – *flag* float "to-float"

Actual stack effect: (c_addr u – r t | f). Attempt to convert the character string *c-addr u* to internal floating-point representation. If the string represents a valid floating-point number *r* is placed on the floating-point stack and *flag* is true. Otherwise, *flag* is false. A string of blanks is a special case and represents the floating-point number 0.

`accept` *c-addr* +*n1* – +*n2* core "accept"

Get a string of up to *n1* characters from the user input device and store it at *c-addr*. *n2* is the length of the received string. The user indicates the end by pressing ⟨RET⟩. Gforth supports all the editing functions available on the Forth command line (including history and word completion) in `accept`.

`edit-line` *c-addr n1 n2* – *n3* gforth "edit-line"

edit the string with length *n2* in the buffer *c-addr n1*, like `accept`.

`pad` – *c-addr* core-ext "pad"

c-addr is the address of a transient region that can be used as temporary data storage. At least 84 characters of space is available.

`convert` *ud1 c-addr1* – *ud2 c-addr2* core-ext "convert"

OBSOLESCENT: superseded by `>number`.

`expect` *c-addr* +*n* – core-ext "expect"

Receive a string of at most +*n* characters, and store it in memory starting at *c-addr*. The string is displayed. Input terminates when the <return> key is pressed or +*n* characters have been received. The normal Gforth line editing capabilites are available. The length of the string is stored in `span`; it does not include the <return> character. OBSOLESCENT: superceeded by `accept`.

`span` – *c-addr* core-ext "span"

`Variable` – *c-addr* is the address of a cell that stores the length of the last string received by `expect`. OBSOLESCENT.

5.19.6 Pipes

In addition to using Gforth in pipes created by other processes (see Section 2.6 [Gforth in pipes], page 7), you can create your own pipe with `open-pipe`, and read from or write to it.

`open-pipe` *c-addr u wfam* – *wfileid wior* gforth "open-pipe"

`close-pipe` *wfileid* – *wretval wior* gforth "close-pipe"

If you write to a pipe, Gforth can throw a `broken-pipe-error`; if you don't catch this exception, Gforth will catch it and exit, usually silently (see Section 2.6 [Gforth in pipes], page 7). Since you probably do not want this, you should wrap a `catch` or `try` block around the code from `open-pipe` to `close-pipe`, so you can deal with the problem yourself, and then return to regular processing.

`broken-pipe-error` – *n* gforth "broken-pipe-error"

the error number for a broken pipe

5.20 Locals

Local variables can make Forth programming more enjoyable and Forth programs easier to read. Unfortunately, the locals of ANS Forth are laden with restrictions. Therefore, we provide not only the ANS Forth locals wordset, but also our own, more powerful locals wordset (we implemented the ANS Forth locals wordset through our locals wordset).

The ideas in this section have also been published in M. Anton Ertl, *Automatic Scoping of Local Variables* (`http://www.complang.tuwien.ac.at/papers/ertl94l.ps.gz`), EuroForth '94.

5.20.1 Gforth locals

Locals can be defined with

```
{ local1 local2 ... -- comment }
```

or

```
{ local1 local2 ... }
```

E.g.,

```
: max { n1 n2 -- n3 }
 n1 n2 > if
   n1
 else
   n2
 endif ;
```

The similarity of locals definitions with stack comments is intended. A locals definition often replaces the stack comment of a word. The order of the locals corresponds to the order in a stack comment and everything after the `--` is really a comment.

This similarity has one disadvantage: It is too easy to confuse locals declarations with stack comments, causing bugs and making them hard to find. However, this problem can be avoided by appropriate coding conventions: Do not use both notations in the same program. If you do, they should be distinguished using additional means, e.g. by position.

The name of the local may be preceded by a type specifier, e.g., `F:` for a floating point value:

```
: CX* { F: Ar F: Ai F: Br F: Bi -- Cr Ci }
\ complex multiplication
 Ar Br f* Ai Bi f* f-
 Ar Bi f* Ai Br f* f+ ;
```

Gforth currently supports cells (`W:`, `W^`), doubles (`D:`, `D^`), floats (`F:`, `F^`) and characters (`C:`, `C^`) in two flavours: a value-flavoured local (defined with `W:`, `D:` etc.) produces its value and can be changed with `TO`. A variable-flavoured local (defined with `W^` etc.) produces its address (which becomes invalid when the variable's scope is left). E.g., the standard word `emit` can be defined in terms of `type` like this:

```
: emit { C^ char* -- }
    char* 1 type ;
```

A local without type specifier is a `W:` local. Both flavours of locals are initialized with values from the data or FP stack.

Currently there is no way to define locals with user-defined data structures, but we are working on it.

Gforth allows defining locals everywhere in a colon definition. This poses the following questions:

5.20.1.1 Where are locals visible by name?

Basically, the answer is that locals are visible where you would expect it in block-structured languages, and sometimes a little longer. If you want to restrict the scope of a local, enclose its definition in `SCOPE`...`ENDSCOPE`.

`scope` *compilation – scope ; run-time –* gforth "scope"

`endscope` *compilation scope – ; run-time –* gforth "endscope"

These words behave like control structure words, so you can use them with `CS-PICK` and `CS-ROLL` to restrict the scope in arbitrary ways.

If you want a more exact answer to the visibility question, here's the basic principle: A local is visible in all places that can only be reached through the definition of the local[23]. In other words, it is not visible in places that can be reached without going through the definition of the local. E.g., locals defined in `IF`...`ENDIF` are visible until the `ENDIF`, locals defined in `BEGIN`...`UNTIL` are visible after the `UNTIL` (until, e.g., a subsequent `ENDSCOPE`).

The reasoning behind this solution is: We want to have the locals visible as long as it is meaningful. The user can always make the visibility shorter by using explicit scoping. In a place that can only be reached through the definition of a local, the meaning of a local name is clear. In other places it is not: How is the local initialized at the control flow path that does not contain the definition? Which local is meant, if the same name is defined twice in two independent control flow paths?

This should be enough detail for nearly all users, so you can skip the rest of this section. If you really must know all the gory details and options, read on.

In order to implement this rule, the compiler has to know which places are unreachable. It knows this automatically after `AHEAD`, `AGAIN`, `EXIT` and `LEAVE`; in other cases (e.g., after most `THROW`s), you can use the word `UNREACHABLE` to tell the compiler that the control flow never reaches that place. If `UNREACHABLE` is not used where it could, the only consequence is that the visibility of some locals is more limited than the rule above says. If `UNREACHABLE` is used where it should not (i.e., if you lie to the compiler), buggy code will be produced.

`UNREACHABLE` – gforth "UNREACHABLE"

Another problem with this rule is that at `BEGIN`, the compiler does not know which locals will be visible on the incoming back-edge. All problems discussed in the following are due to this ignorance of the compiler (we discuss the problems using `BEGIN` loops as examples; the discussion also applies to `?DO` and other loops). Perhaps the most insidious example is:

```
AHEAD
BEGIN
  x
[ 1 CS-ROLL ] THEN
  { x }
```

[23] In compiler construction terminology, all places dominated by the definition of the local.

```
  ...
UNTIL
```

This should be legal according to the visibility rule. The use of `x` can only be reached through the definition; but that appears textually below the use.

From this example it is clear that the visibility rules cannot be fully implemented without major headaches. Our implementation treats common cases as advertised and the exceptions are treated in a safe way: The compiler makes a reasonable guess about the locals visible after a `BEGIN`; if it is too pessimistic, the user will get a spurious error about the local not being defined; if the compiler is too optimistic, it will notice this later and issue a warning. In the case above the compiler would complain about `x` being undefined at its use. You can see from the obscure examples in this section that it takes quite unusual control structures to get the compiler into trouble, and even then it will often do fine.

If the `BEGIN` is reachable from above, the most optimistic guess is that all locals visible before the `BEGIN` will also be visible after the `BEGIN`. This guess is valid for all loops that are entered only through the `BEGIN`, in particular, for normal `BEGIN`...`WHILE`...`REPEAT` and `BEGIN`...`UNTIL` loops and it is implemented in our compiler. When the branch to the `BEGIN` is finally generated by `AGAIN` or `UNTIL`, the compiler checks the guess and warns the user if it was too optimistic:

```
IF
  { x }
BEGIN
  \ x ?
[ 1 cs-roll ] THEN
  ...
UNTIL
```

Here, `x` lives only until the `BEGIN`, but the compiler optimistically assumes that it lives until the `THEN`. It notices this difference when it compiles the `UNTIL` and issues a warning. The user can avoid the warning, and make sure that `x` is not used in the wrong area by using explicit scoping:

```
IF
  SCOPE
  { x }
  ENDSCOPE
BEGIN
[ 1 cs-roll ] THEN
  ...
UNTIL
```

Since the guess is optimistic, there will be no spurious error messages about undefined locals.

If the `BEGIN` is not reachable from above (e.g., after `AHEAD` or `EXIT`), the compiler cannot even make an optimistic guess, as the locals visible after the `BEGIN` may be defined later. Therefore, the compiler assumes that no locals are visible after the `BEGIN`. However, the user can use `ASSUME-LIVE` to make the compiler assume that the same locals are visible at the BEGIN as at the point where the top control-flow stack item was created.

`ASSUME-LIVE` *orig – orig* gforth “ASSUME-LIVE”

E.g.,

```
{ x }
AHEAD
ASSUME-LIVE
BEGIN
  x
[ 1 CS-ROLL ] THEN
  ...
UNTIL
```

Other cases where the locals are defined before the BEGIN can be handled by inserting an appropriate CS-ROLL before the ASSUME-LIVE (and changing the control-flow stack manipulation behind the ASSUME-LIVE).

Cases where locals are defined after the BEGIN (but should be visible immediately after the BEGIN) can only be handled by rearranging the loop. E.g., the "most insidious" example above can be arranged into:

```
BEGIN
  { x }
  ... 0=
WHILE
  x
REPEAT
```

5.20.1.2 How long do locals live?

The right answer for the lifetime question would be: A local lives at least as long as it can be accessed. For a value-flavoured local this means: until the end of its visibility. However, a variable-flavoured local could be accessed through its address far beyond its visibility scope. Ultimately, this would mean that such locals would have to be garbage collected. Since this entails un-Forth-like implementation complexities, I adopted the same cowardly solution as some other languages (e.g., C): The local lives only as long as it is visible; afterwards its address is invalid (and programs that access it afterwards are erroneous).

5.20.1.3 Locals programming style

The freedom to define locals anywhere has the potential to change programming styles dramatically. In particular, the need to use the return stack for intermediate storage vanishes. Moreover, all stack manipulations (except PICKs and ROLLs with run-time determined arguments) can be eliminated: If the stack items are in the wrong order, just write a locals definition for all of them; then write the items in the order you want.

This seems a little far-fetched and eliminating stack manipulations is unlikely to become a conscious programming objective. Still, the number of stack manipulations will be reduced dramatically if local variables are used liberally (e.g., compare max (see Section 5.20.1 [Gforth locals], page 121) with a traditional implementation of max).

This shows one potential benefit of locals: making Forth programs more readable. Of course, this benefit will only be realized if the programmers continue to honour the principle of factoring instead of using the added latitude to make the words longer.

Using TO can and should be avoided. Without TO, every value-flavoured local has only a single assignment and many advantages of functional languages apply to Forth. I.e., programs are easier to analyse, to optimize and to read: It is clear from the definition what the local stands for, it does not turn into something different later.

E.g., a definition using `TO` might look like this:

```
: strcmp { addr1 u1 addr2 u2 -- n }
 u1 u2 min 0
 ?do
   addr1 c@ addr2 c@ -
   ?dup-if
     unloop exit
   then
   addr1 char+ TO addr1
   addr2 char+ TO addr2
 loop
 u1 u2 - ;
```

Here, `TO` is used to update `addr1` and `addr2` at every loop iteration. `strcmp` is a typical example of the readability problems of using `TO`. When you start reading `strcmp`, you think that `addr1` refers to the start of the string. Only near the end of the loop you realize that it is something else.

This can be avoided by defining two locals at the start of the loop that are initialized with the right value for the current iteration.

```
: strcmp { addr1 u1 addr2 u2 -- n }
 addr1 addr2
 u1 u2 min 0
 ?do { s1 s2 }
   s1 c@ s2 c@ -
   ?dup-if
     unloop exit
   then
   s1 char+ s2 char+
 loop
 2drop
 u1 u2 - ;
```

Here it is clear from the start that `s1` has a different value in every loop iteration.

5.20.1.4 Locals implementation

Gforth uses an extra locals stack. The most compelling reason for this is that the return stack is not float-aligned; using an extra stack also eliminates the problems and restrictions of using the return stack as locals stack. Like the other stacks, the locals stack grows toward lower addresses. A few primitives allow an efficient implementation:

`@local#` *#noffset – w* gforth "fetch-local-number"

`f@local#` *#noffset – r* gforth "f-fetch-local-number"

`laddr#` *#noffset – c-addr* gforth "laddr-number"

`lp+!#` *#noffset –* gforth "lp-plus-store-number"

used with negative immediate values it allocates memory on the local stack, a positive immediate argument drops memory from the local stack `lp!` *c-addr* – gforth "lp-store"

`>l` *w* – gforth "to-l"

`f>l` *r* – gforth "f-to-l"

In addition to these primitives, some specializations of these primitives for commonly occurring inline arguments are provided for efficiency reasons, e.g., `@local0` as specialization of `@local#` for the inline argument 0. The following compiling words compile the right specialized version, or the general version, as appropriate:

`compile-lp+!` *n* – gforth "compile-l-p-plus-store"

Combinations of conditional branches and `lp+!#` like `?branch-lp+!#` (the locals pointer is only changed if the branch is taken) are provided for efficiency and correctness in loops.

A special area in the dictionary space is reserved for keeping the local variable names. `{` switches the dictionary pointer to this area and `}` switches it back and generates the locals initializing code. `W:` etc. are normal defining words. This special area is cleared at the start of every colon definition.

A special feature of Gforth's dictionary is used to implement the definition of locals without type specifiers: every word list (aka vocabulary) has its own methods for searching etc. (see Section 5.15 [Word Lists], page 100). For the present purpose we defined a word list with a special search method: When it is searched for a word, it actually creates that word using `W:`. `{` changes the search order to first search the word list containing `}`, `W:` etc., and then the word list for defining locals without type specifiers.

The lifetime rules support a stack discipline within a colon definition: The lifetime of a local is either nested with other locals lifetimes or it does not overlap them.

At `BEGIN`, `IF`, and `AHEAD` no code for locals stack pointer manipulation is generated. Between control structure words locals definitions can push locals onto the locals stack. `AGAIN` is the simplest of the other three control flow words. It has to restore the locals stack depth of the corresponding `BEGIN` before branching. The code looks like this:

`lp+!#` current-locals-size − dest-locals-size
`branch` <begin>

`UNTIL` is a little more complicated: If it branches back, it must adjust the stack just like `AGAIN`. But if it falls through, the locals stack must not be changed. The compiler generates the following code:

`?branch-lp+!#` <begin> current-locals-size − dest-locals-size

The locals stack pointer is only adjusted if the branch is taken.

`THEN` can produce somewhat inefficient code:

`lp+!#` current-locals-size − orig-locals-size
<orig target>:
`lp+!#` orig-locals-size − new-locals-size

The second `lp+!#` adjusts the locals stack pointer from the level at the *orig* point to the level after the `THEN`. The first `lp+!#` adjusts the locals stack pointer from the current level to the level at the orig point, so the complete effect is an adjustment from the current level to the right level after the `THEN`.

In a conventional Forth implementation a dest control-flow stack entry is just the target address and an orig entry is just the address to be patched. Our locals implementation adds a word list to every orig or dest item. It is the list of locals visible (or assumed visible) at the point described by the entry. Our implementation also adds a tag to identify the kind of entry, in particular to differentiate between live and dead (reachable and unreachable) orig entries.

A few unusual operations have to be performed on locals word lists:

`common-list` *list1 list2 – list3* gforth-internal "common-list"

`sub-list?` *list1 list2 – f* gforth-internal "sub-list?"

`list-size` *list – u* gforth-internal "list-size"

Several features of our locals word list implementation make these operations easy to implement: The locals word lists are organised as linked lists; the tails of these lists are shared, if the lists contain some of the same locals; and the address of a name is greater than the address of the names behind it in the list.

Another important implementation detail is the variable `dead-code`. It is used by `BEGIN` and `THEN` to determine if they can be reached directly or only through the branch that they resolve. `dead-code` is set by `UNREACHABLE`, `AHEAD`, `EXIT` etc., and cleared at the start of a colon definition, by `BEGIN` and usually by `THEN`.

Counted loops are similar to other loops in most respects, but `LEAVE` requires special attention: It performs basically the same service as `AHEAD`, but it does not create a control-flow stack entry. Therefore the information has to be stored elsewhere; traditionally, the information was stored in the target fields of the branches created by the `LEAVE`s, by organizing these fields into a linked list. Unfortunately, this clever trick does not provide enough space for storing our extended control flow information. Therefore, we introduce another stack, the leave stack. It contains the control-flow stack entries for all unresolved `LEAVE`s.

Local names are kept until the end of the colon definition, even if they are no longer visible in any control-flow path. In a few cases this may lead to increased space needs for the locals name area, but usually less than reclaiming this space would cost in code size.

5.20.2 ANS Forth locals

The ANS Forth locals wordset does not define a syntax for locals, but words that make it possible to define various syntaxes. One of the possible syntaxes is a subset of the syntax we used in the Gforth locals wordset, i.e.:

```
{ local1 local2 ... -- comment }
```

or

```
{ local1 local2 ... }
```

The order of the locals corresponds to the order in a stack comment. The restrictions are:

- Locals can only be cell-sized values (no type specifiers are allowed).
- Locals can be defined only outside control structures.
- Locals can interfere with explicit usage of the return stack. For the exact (and long) rules, see the standard. If you don't use return stack accessing words in a definition using locals, you will be all right. The purpose of this rule is to make locals implementation on the return stack easier.
- The whole definition must be in one line.

Locals defined in ANS Forth behave like `VALUE`s (see Section 5.9.4 [Values], page 75). I.e., they are initialized from the stack. Using their name produces their value. Their value can be changed using `TO`.

Since the syntax above is supported by Gforth directly, you need not do anything to use it. If you want to port a program using this syntax to another ANS Forth system, use '`compat/anslocal.fs`' to implement the syntax on the other system.

Note that a syntax shown in the standard, section A.13 looks similar, but is quite different in having the order of locals reversed. Beware!

The ANS Forth locals wordset itself consists of one word:

`(local)` *addr u* – local "paren-local-paren"

The ANS Forth locals extension wordset defines a syntax using `locals|`, but it is so awful that we strongly recommend not to use it. We have implemented this syntax to make porting to Gforth easy, but do not document it here. The problem with this syntax is that the locals are defined in an order reversed with respect to the standard stack comment notation, making programs harder to read, and easier to misread and miswrite. The only merit of this syntax is that it is easy to implement using the ANS Forth locals wordset.

5.21 Structures

This section presents the structure package that comes with Gforth. A version of the package implemented in ANS Forth is available in '`compat/struct.fs`'. This package was inspired by a posting on comp.lang.forth in 1989 (unfortunately I don't remember, by whom; possibly John Hayes). A version of this section has been published in M. Anton Ertl, Yet Another Forth Structures Package (`http://www.complang.tuwien.ac.at/forth/objects/structs.html`), Forth Dimensions 19(3), pages 13–16. Marcel Hendrix provided helpful comments.

5.21.1 Why explicit structure support?

If we want to use a structure containing several fields, we could simply reserve memory for it, and access the fields using address arithmetic (see Section 5.7.5 [Address arithmetic], page 61). As an example, consider a structure with the following fields

`a` is a float

`b` is a cell

`c` is a float

Given the (float-aligned) base address of the structure we get the address of the field

`a` without doing anything further.

`b` with `float+`

`c` with `float+ cell+ faligned`

It is easy to see that this can become quite tiring.

Moreover, it is not very readable, because seeing a `cell+` tells us neither which kind of structure is accessed nor what field is accessed; we have to somehow infer the kind of structure, and then look up in the documentation, which field of that structure corresponds to that offset.

Finally, this kind of address arithmetic also causes maintenance troubles: If you add or delete a field somewhere in the middle of the structure, you have to find and change all computations for the fields afterwards.

So, instead of using `cell+` and friends directly, how about storing the offsets in constants:

```
0 constant a-offset
0 float+ constant b-offset
0 float+ cell+ faligned c-offset
```

Now we can get the address of field `x` with `x-offset +`. This is much better in all respects. Of course, you still have to change all later offset definitions if you add a field. You can fix this by declaring the offsets in the following way:

```
0 constant a-offset
a-offset float+ constant b-offset
b-offset cell+ faligned constant c-offset
```

Since we always use the offsets with `+`, we could use a defining word `cfield` that includes the `+` in the action of the defined word:

```
: cfield ( n "name" -- )
    create ,
does> ( name execution: addr1 -- addr2 )
    @ + ;

0 cfield a
0 a float+ cfield b
0 b cell+ faligned cfield c
```

Instead of `x-offset +`, we now simply write `x`.

The structure field words now can be used quite nicely. However, their definition is still a bit cumbersome: We have to repeat the name, the information about size and alignment is distributed before and after the field definitions etc. The structure package presented here addresses these problems.

5.21.2 Structure Usage

You can define a structure for a (data-less) linked list with:

```
struct
    cell% field list-next
end-struct list%
```

With the address of the list node on the stack, you can compute the address of the field that contains the address of the next node with `list-next`. E.g., you can determine the length of a list with:

```
: list-length ( list -- n )
\ "list" is a pointer to the first element of a linked list
\ "n" is the length of the list
    0 BEGIN ( list1 n1 )
        over
    WHILE ( list1 n1 )
```

```
        1+ swap list-next @ swap
    REPEAT
    nip ;
```

You can reserve memory for a list node in the dictionary with `list% %allot`, which leaves the address of the list node on the stack. For the equivalent allocation on the heap you can use `list% %alloc` (or, for an `allocate`-like stack effect (i.e., with ior), use `list% %allocate`). You can get the the size of a list node with `list% %size` and its alignment with `list% %alignment`.

Note that in ANS Forth the body of a `create`d word is `aligned` but not necessarily `faligned`; therefore, if you do a:

```
create name foo% %allot drop
```

then the memory alloted for `foo%` is guaranteed to start at the body of *name* only if `foo%` contains only character, cell and double fields. Therefore, if your structure contains floats, better use

```
foo% %allot constant name
```

You can include a structure `foo%` as a field of another structure, like this:

```
struct
...
    foo% field ...
...
end-struct ...
```

Instead of starting with an empty structure, you can extend an existing structure. E.g., a plain linked list without data, as defined above, is hardly useful; You can extend it to a linked list of integers, like this:[24]

```
list%
    cell% field intlist-int
end-struct intlist%
```

`intlist%` is a structure with two fields: `list-next` and `intlist-int`.

You can specify an array type containing *n* elements of type `foo%` like this:

```
foo% n *
```

You can use this array type in any place where you can use a normal type, e.g., when defining a `field`, or with `%allot`.

The first field is at the base address of a structure and the word for this field (e.g., `list-next`) actually does not change the address on the stack. You may be tempted to leave it away in the interest of run-time and space efficiency. This is not necessary, because the structure package optimizes this case: If you compile a first-field words, no code is generated. So, in the interest of readability and maintainability you should include the word for the field when accessing the field.

[24] This feature is also known as *extended records*. It is the main innovation in the Oberon language; in other words, adding this feature to Modula-2 led Wirth to create a new language, write a new compiler etc. Adding this feature to Forth just required a few lines of code.

5.21.3 Structure Naming Convention

The field names that come to (my) mind are often quite generic, and, if used, would cause frequent name clashes. E.g., many structures probably contain a `counter` field. The structure names that come to (my) mind are often also the logical choice for the names of words that create such a structure.

Therefore, I have adopted the following naming conventions:

- The names of fields are of the form *struct*-*field*, where *struct* is the basic name of the structure, and *field* is the basic name of the field. You can think of field words as converting the (address of the) structure into the (address of the) field.
- The names of structures are of the form *struct*%, where *struct* is the basic name of the structure.

This naming convention does not work that well for fields of extended structures; e.g., the integer list structure has a field `intlist-int`, but has `list-next`, not `intlist-next`.

5.21.4 Structure Implementation

The central idea in the implementation is to pass the data about the structure being built on the stack, not in some global variable. Everything else falls into place naturally once this design decision is made.

The type description on the stack is of the form *align size*. Keeping the size on the top-of-stack makes dealing with arrays very simple.

`field` is a defining word that uses `Create` and `DOES>`. The body of the field contains the offset of the field, and the normal `DOES>` action is simply:

```
@ +
```

i.e., add the offset to the address, giving the stack effect *addr1* – *addr2* for a field.

This simple structure is slightly complicated by the optimization for fields with offset 0, which requires a different `DOES>`-part (because we cannot rely on there being something on the stack if such a field is invoked during compilation). Therefore, we put the different `DOES>`-parts in separate words, and decide which one to invoke based on the offset. For a zero offset, the field is basically a noop; it is immediate, and therefore no code is generated when it is compiled.

5.21.5 Structure Glossary

`%align` *align size* – gforth "%align"

Align the data space pointer to the alignment *align*.

`%alignment` *align size* – *align* gforth "%alignment"

The alignment of the structure.

`%alloc` *size align* – *addr* gforth "%alloc"

Allocate *size* address units with alignment *align*, giving a data block at *addr*; `throw` an ior code if not successful.

`%allocate` *align size* – *addr ior* gforth "%allocate"

Allocate *size* address units with alignment *align*, similar to `allocate`.

`%allot` *align size – addr* gforth "%allot"

Allot *size* address units of data space with alignment *align*; the resulting block of data is found at *addr*.

`cell%` *– align size* gforth "cell%"

`char%` *– align size* gforth "char%"

`dfloat%` *– align size* gforth "dfloat%"

`double%` *– align size* gforth "double%"

`end-struct` *align size* "*name*" – gforth "end-struct"

Define a structure/type descriptor *name* with alignment *align* and size *size1* (*size* rounded up to be a multiple of *align*).
`name` execution: – *align size1*

`field` *align1 offset1 align size* "*name*" – *align2 offset2* gforth "field"

Create a field *name* with offset *offset1*, and the type given by *align size*. *offset2* is the offset of the next field, and *align2* is the alignment of all fields.
`name` execution: *addr1* – *addr2*.
addr2=addr1+offset1

`float%` *– align size* gforth "float%"

`naligned` *addr1 n – addr2* gforth "naligned"

addr2 is the aligned version of *addr1* with respect to the alignment *n*.

`sfloat%` *– align size* gforth "sfloat%"

`%size` *align size – size* gforth "%size"

The size of the structure.

`struct` *– align size* gforth "struct"

An empty structure, used to start a structure definition.

5.22 Object-oriented Forth

Gforth comes with three packages for object-oriented programming: '`objects.fs`', '`oof.fs`', and '`mini-oof.fs`'; none of them is preloaded, so you have to `include` them before use. The most important differences between these packages (and others) are discussed in Section 5.22.6 [Comparison with other object models], page 150. All packages are written in ANS Forth and can be used with any other ANS Forth.

5.22.1 Why object-oriented programming?

Often we have to deal with several data structures (*objects*), that have to be treated similarly in some respects, but differently in others. Graphical objects are the textbook example: circles, triangles, dinosaurs, icons, and others, and we may want to add more during program development. We want to apply some operations to any graphical object, e.g., `draw` for displaying it on the screen. However, `draw` has to do something different for every kind of object.

We could implement `draw` as a big `CASE` control structure that executes the appropriate code depending on the kind of object to be drawn. This would be not be very elegant, and, moreover, we would have to change `draw` every time we add a new kind of graphical object (say, a spaceship).

What we would rather do is: When defining spaceships, we would tell the system: "Here's how you `draw` a spaceship; you figure out the rest".

This is the problem that all systems solve that (rightfully) call themselves object-oriented; the object-oriented packages presented here solve this problem (and not much else).

5.22.2 Object-Oriented Terminology

This section is mainly for reference, so you don't have to understand all of it right away. The terminology is mainly Smalltalk-inspired. In short:

class a data structure definition with some extras.

object an instance of the data structure described by the class definition.

instance variables
fields of the data structure.

selector (or *method selector*) a word (e.g., `draw`) that performs an operation on a variety of data structures (classes). A selector describes *what* operation to perform. In C++ terminology: a (pure) virtual function.

method the concrete definition that performs the operation described by the selector for a specific class. A method specifies *how* the operation is performed for a specific class.

selector invocation
a call of a selector. One argument of the call (the TOS (top-of-stack)) is used for determining which method is used. In Smalltalk terminology: a message (consisting of the selector and the other arguments) is sent to the object.

receiving object
the object used for determining the method executed by a selector invocation. In the '`objects.fs`' model, it is the object that is on the TOS when the selector is invoked. (*Receiving* comes from the Smalltalk *message* terminology.)

child class a class that has (*inherits*) all properties (instance variables, selectors, methods) from a *parent class*. In Smalltalk terminology: The subclass inherits from the superclass. In C++ terminology: The derived class inherits from the base class.

5.22.3 The '`objects.fs`' model

This section describes the '`objects.fs`' package. This material also has been published in M. Anton Ertl, *Yet Another Forth Objects Package* (`http://www.complang.tuwien.ac.at/forth/objects/objects.html`), Forth Dimensions 19(2), pages 37–43.

This section assumes that you have read Section 5.21 [Structures], page 128.

The techniques on which this model is based have been used to implement the parser generator, Gray, and have also been used in Gforth for implementing the various flavours of word lists (hashed or not, case-sensitive or not, special-purpose word lists for locals etc.).

Marcel Hendrix provided helpful comments on this section.

5.22.3.1 Properties of the '`objects.fs`' model

- It is straightforward to pass objects on the stack. Passing selectors on the stack is a little less convenient, but possible.
- Objects are just data structures in memory, and are referenced by their address. You can create words for objects with normal defining words like `constant`. Likewise, there is no difference between instance variables that contain objects and those that contain other data.
- Late binding is efficient and easy to use.
- It avoids parsing, and thus avoids problems with state-smartness and reduced extensibility; for convenience there are a few parsing words, but they have non-parsing counterparts. There are also a few defining words that parse. This is hard to avoid, because all standard defining words parse (except `:noname`); however, such words are not as bad as many other parsing words, because they are not state-smart.
- It does not try to incorporate everything. It does a few things and does them well (IMO). In particular, this model was not designed to support information hiding (although it has features that may help); you can use a separate package for achieving this.
- It is layered; you don't have to learn and use all features to use this model. Only a few features are necessary (see Section 5.22.3.2 [Basic Objects Usage], page 134, see Section 5.22.3.3 [The Objects base class], page 135, see Section 5.22.3.4 [Creating objects], page 135.), the others are optional and independent of each other.
- An implementation in ANS Forth is available.

5.22.3.2 Basic '`objects.fs`' Usage

You can define a class for graphical objects like this:

```
object class \ "object" is the parent class
  selector draw ( x y graphical -- )
end-class graphical
```

This code defines a class `graphical` with an operation `draw`. We can perform the operation `draw` on any `graphical` object, e.g.:

```
100 100 t-rex draw
```

where `t-rex` is a word (say, a constant) that produces a graphical object.

How do we create a graphical object? With the present definitions, we cannot create a useful graphical object. The class `graphical` describes graphical objects in general, but not any concrete graphical object type (C++ users would call it an *abstract class*); e.g., there is no method for the selector `draw` in the class `graphical`.

For concrete graphical objects, we define child classes of the class `graphical`, e.g.:

```
graphical class \ "graphical" is the parent class
  cell% field circle-radius

:noname ( x y circle -- )
  circle-radius @ draw-circle ;
overrides draw

:noname ( n-radius circle -- )
  circle-radius ! ;
overrides construct

end-class circle
```

Here we define a class `circle` as a child of `graphical`, with field `circle-radius` (which behaves just like a field (see Section 5.21 [Structures], page 128); it defines (using `overrides`) new methods for the selectors `draw` and `construct` (`construct` is defined in `object`, the parent class of `graphical`).

Now we can create a circle on the heap (i.e., `allocate`d memory) with:

```
50 circle heap-new constant my-circle
```

`heap-new` invokes `construct`, thus initializing the field `circle-radius` with 50. We can draw this new circle at (100,100) with:

```
100 100 my-circle draw
```

Note: You can only invoke a selector if the object on the TOS (the receiving object) belongs to the class where the selector was defined or one of its descendents; e.g., you can invoke `draw` only for objects belonging to `graphical` or its descendents (e.g., `circle`). Immediately before `end-class`, the search order has to be the same as immediately after `class`.

5.22.3.3 The '`object.fs`' base class

When you define a class, you have to specify a parent class. So how do you start defining classes? There is one class available from the start: `object`. It is ancestor for all classes and so is the only class that has no parent. It has two selectors: `construct` and `print`.

5.22.3.4 Creating objects

You can create and initialize an object of a class on the heap with `heap-new` (... class – object) and in the dictionary (allocation with `allot`) with `dict-new` (... class – object). Both words invoke `construct`, which consumes the stack items indicated by "..." above.

If you want to allocate memory for an object yourself, you can get its alignment and size with `class-inst-size 2@` (class – align size). Once you have memory for an object, you can initialize it with `init-object` (... class object –); `construct` does only a part of the necessary work.

5.22.3.5 Object-Oriented Programming Style

This section is not exhaustive.

In general, it is a good idea to ensure that all methods for the same selector have the same stack effect: when you invoke a selector, you often have no idea which method will be invoked, so, unless all methods have the same stack effect, you will not know the stack effect of the selector invocation.

One exception to this rule is methods for the selector `construct`. We know which method is invoked, because we specify the class to be constructed at the same place. Actually, I defined `construct` as a selector only to give the users a convenient way to specify initialization. The way it is used, a mechanism different from selector invocation would be more natural (but probably would take more code and more space to explain).

5.22.3.6 Class Binding

Normal selector invocations determine the method at run-time depending on the class of the receiving object. This run-time selection is called *late binding*.

Sometimes it's preferable to invoke a different method. For example, you might want to use the simple method for `print`ing `object`s instead of the possibly long-winded `print` method of the receiver class. You can achieve this by replacing the invocation of `print` with:

```
[bind] object print
```

in compiled code or:

```
bind object print
```

in interpreted code. Alternatively, you can define the method with a name (e.g., `print-object`), and then invoke it through the name. Class binding is just a (often more convenient) way to achieve the same effect; it avoids name clutter and allows you to invoke methods directly without naming them first.

A frequent use of class binding is this: When we define a method for a selector, we often want the method to do what the selector does in the parent class, and a little more. There is a special word for this purpose: `[parent]`; `[parent]` *selector* is equivalent to `[bind]` *parent selector*, where *parent* is the parent class of the current class. E.g., a method definition might look like:

```
:noname
  dup [parent] foo \ do parent's foo on the receiving object
  ... \ do some more
; overrides foo
```

In *Object-oriented programming in ANS Forth* (Forth Dimensions, March 1997), Andrew McKewan presents class binding as an optimization technique. I recommend not using it for this purpose unless you are in an emergency. Late binding is pretty fast with this model anyway, so the benefit of using class binding is small; the cost of using class binding where it is not appropriate is reduced maintainability.

While we are at programming style questions: You should bind selectors only to ancestor classes of the receiving object. E.g., say, you know that the receiving object is of class `foo` or its descendents; then you should bind only to `foo` and its ancestors.

5.22.3.7 Method conveniences

In a method you usually access the receiving object pretty often. If you define the method as a plain colon definition (e.g., with `:noname`), you may have to do a lot of stack gymnastics. To avoid this, you can define the method with `m: ... ;m`. E.g., you could define the method for `draw`ing a `circle` with

```
m: ( x y circle -- )
  ( x y ) this circle-radius @ draw-circle ;m
```

When this method is executed, the receiver object is removed from the stack; you can access it with `this` (admittedly, in this example the use of `m: ... ;m` offers no advantage). Note that I specify the stack effect for the whole method (i.e. including the receiver object), not just for the code between `m:` and `;m`. You cannot use `exit` in `m:...;m`; instead, use `exitm`.[25]

You will frequently use sequences of the form `this` *field* (in the example above: `this circle-radius`). If you use the field only in this way, you can define it with `inst-var` and eliminate the `this` before the field name. E.g., the `circle` class above could also be defined with:

```
graphical class
  cell% inst-var radius

m: ( x y circle -- )
  radius @ draw-circle ;m
overrides draw

m: ( n-radius circle -- )
  radius ! ;m
overrides construct

end-class circle
```

`radius` can only be used in `circle` and its descendent classes and inside `m:...;m`.

You can also define fields with `inst-value`, which is to `inst-var` what `value` is to `variable`. You can change the value of such a field with `[to-inst]`. E.g., we could also define the class `circle` like this:

```
graphical class
  inst-value radius

m: ( x y circle -- )
  radius draw-circle ;m
overrides draw

m: ( n-radius circle -- )
  [to-inst] radius ;m
overrides construct

end-class circle
```

[25] Moreover, for any word that calls `catch` and was defined before loading `objects.fs`, you have to redefine it like I redefined `catch`: `: catch this >r catch r> to-this ;`

5.22.3.8 Classes and Scoping

Inheritance is frequent, unlike structure extension. This exacerbates the problem with the field name convention (see Section 5.21.3 [Structure Naming Convention], page 131): One always has to remember in which class the field was originally defined; changing a part of the class structure would require changes for renaming in otherwise unaffected code.

To solve this problem, I added a scoping mechanism (which was not in my original charter): A field defined with `inst-var` (or `inst-value`) is visible only in the class where it is defined and in the descendent classes of this class. Using such fields only makes sense in `m:`-defined methods in these classes anyway.

This scoping mechanism allows us to use the unadorned field name, because name clashes with unrelated words become much less likely.

Once we have this mechanism, we can also use it for controlling the visibility of other words: All words defined after `protected` are visible only in the current class and its descendents. `public` restores the compilation (i.e. `current`) word list that was in effect before. If you have several `protected`s without an intervening `public` or `set-current`, `public` will restore the compilation word list in effect before the first of these `protected`s.

5.22.3.9 Dividing classes

You may want to do the definition of methods separate from the definition of the class, its selectors, fields, and instance variables, i.e., separate the implementation from the definition. You can do this in the following way:

```
graphical class
  inst-value radius
end-class circle

... \ do some other stuff

circle methods \ now we are ready

m: ( x y circle -- )
  radius draw-circle ;m
overrides draw

m: ( n-radius circle -- )
  [to-inst] radius ;m
overrides construct

end-methods
```

You can use several `methods`...`end-methods` sections. The only things you can do to the class in these sections are: defining methods, and overriding the class's selectors. You must not define new selectors or fields.

Note that you often have to override a selector before using it. In particular, you usually have to override `construct` with a new method before you can invoke `heap-new` and friends. E.g., you must not create a circle before the `overrides construct` sequence in the example above.

5.22.3.10 Object Interfaces

In this model you can only call selectors defined in the class of the receiving objects or in one of its ancestors. If you call a selector with a receiving object that is not in one of these classes, the result is undefined; if you are lucky, the program crashes immediately.

Now consider the case when you want to have a selector (or several) available in two classes: You would have to add the selector to a common ancestor class, in the worst case to `object`. You may not want to do this, e.g., because someone else is responsible for this ancestor class.

The solution for this problem is interfaces. An interface is a collection of selectors. If a class implements an interface, the selectors become available to the class and its descendents. A class can implement an unlimited number of interfaces. For the problem discussed above, we would define an interface for the selector(s), and both classes would implement the interface.

As an example, consider an interface `storage` for writing objects to disk and getting them back, and a class `foo` that implements it. The code would look like this:

```
interface
  selector write ( file object -- )
  selector read1 ( file object -- )
end-interface storage

bar class
  storage implementation

... overrides write
... overrides read1
...
end-class foo
```

(I would add a word `read` *(file – object)* that uses `read1` internally, but that's beyond the point illustrated here.)

Note that you cannot use `protected` in an interface; and of course you cannot define fields.

In the Neon model, all selectors are available for all classes; therefore it does not need interfaces. The price you pay in this model is slower late binding, and therefore, added complexity to avoid late binding.

5.22.3.11 '`objects.fs`' Implementation

An object is a piece of memory, like one of the data structures described with `struct...end-struct`. It has a field `object-map` that points to the method map for the object's class.

The *method map*[26] is an array that contains the execution tokens (*xt*s) of the methods for the object's class. Each selector contains an offset into a method map.

`selector` is a defining word that uses `CREATE` and `DOES>`. The body of the selector contains the offset; the `DOES>` action for a class selector is, basically:

---

[26] This is Self terminology; in C++ terminology: virtual function table.

```
( object addr ) @ over object-map @ + @ execute
```

Since `object-map` is the first field of the object, it does not generate any code. As you can see, calling a selector has a small, constant cost.

A class is basically a `struct` combined with a method map. During the class definition the alignment and size of the class are passed on the stack, just as with `struct`s, so `field` can also be used for defining class fields. However, passing more items on the stack would be inconvenient, so `class` builds a data structure in memory, which is accessed through the variable `current-interface`. After its definition is complete, the class is represented on the stack by a pointer (e.g., as parameter for a child class definition).

A new class starts off with the alignment and size of its parent, and a copy of the parent's method map. Defining new fields extends the size and alignment; likewise, defining new selectors extends the method map. `overrides` just stores a new *xt* in the method map at the offset given by the selector.

Class binding just gets the *xt* at the offset given by the selector from the class's method map and `compile,`s (in the case of `[bind]`) it.

I implemented `this` as a `value`. At the start of an `m:...;m` method the old `this` is stored to the return stack and restored at the end; and the object on the TOS is stored `TO this`. This technique has one disadvantage: If the user does not leave the method via `;m`, but via `throw` or `exit`, `this` is not restored (and `exit` may crash). To deal with the `throw` problem, I have redefined `catch` to save and restore `this`; the same should be done with any word that can catch an exception. As for `exit`, I simply forbid it (as a replacement, there is `exitm`).

`inst-var` is just the same as `field`, with a different `DOES>` action:

```
@ this +
```

Similar for `inst-value`.

Each class also has a word list that contains the words defined with `inst-var` and `inst-value`, and its protected words. It also has a pointer to its parent. `class` pushes the word lists of the class and all its ancestors onto the search order stack, and `end-class` drops them.

An interface is like a class without fields, parent and protected words; i.e., it just has a method map. If a class implements an interface, its method map contains a pointer to the method map of the interface. The positive offsets in the map are reserved for class methods, therefore interface map pointers have negative offsets. Interfaces have offsets that are unique throughout the system, unlike class selectors, whose offsets are only unique for the classes where the selector is available (invokable).

This structure means that interface selectors have to perform one indirection more than class selectors to find their method. Their body contains the interface map pointer offset in the class method map, and the method offset in the interface method map. The `does>` action for an interface selector is, basically:

```
( object selector-body )
2dup selector-interface @ ( object selector-body object interface-offset )
swap object-map @ + @ ( object selector-body map )
swap selector-offset @ + @ execute
```

where `object-map` and `selector-offset` are first fields and generate no code.

As a concrete example, consider the following code:

```
interface
  selector if1sel1
  selector if1sel2
end-interface if1

object class
  if1 implementation
  selector cl1sel1
  cell% inst-var cl1iv1

' m1 overrides construct
' m2 overrides if1sel1
' m3 overrides if1sel2
' m4 overrides cl1sel2
end-class cl1

create obj1 object dict-new drop
create obj2 cl1    dict-new drop
```

The data structure created by this code (including the data structure for `object`) is shown in the figure (`objects-implementation.eps`), assuming a cell size of 4.

5.22.3.12 '`objects.fs`' Glossary

`bind` *... "class" "selector" – ...* objects "bind"

Execute the method for *selector* in *class*.

`<bind>` *class selector-xt – xt* objects "<bind>"

xt is the method for the selector *selector-xt* in *class*.

`bind'` *"class" "selector" – xt* objects "bind"'

xt is the method for *selector* in *class*.

`[bind]` *compile-time: "class" "selector" – ; run-time: ... object – ...* objects "[bind]"

Compile the method for *selector* in *class*.

`class` *parent-class – align offset* objects "class"

Start a new class definition as a child of *parent-class*. *align offset* are for use by *field* etc.

`class->map` *class – map* objects "class->map"

map is the pointer to *class*'s method map; it points to the place in the map to which the selector offsets refer (i.e., where *object-map*s point to).

`class-inst-size` *class – addr* objects "class-inst-size"

Give the size specification for an instance (i.e. an object) of *class*; used as `class-inst-size 2 ( class -- align size )`.

`class-override!` *xt sel-xt class-map –* objects "class-override!"

xt is the new method for the selector *sel-xt* in *class-map*.

`class-previous` *class* – objects "class-previous"

Drop *class*'s wordlists from the search order. No checking is made whether *class*'s wordlists are actually on the search order.

`class>order` *class* – objects "class>order"

Add *class*'s wordlists to the head of the search-order.

`construct` *... object* – objects "construct"

Initialize the data fields of *object*. The method for the class *object* just does nothing: `( object -- )`.

`current'` "*selector*" – *xt* objects "current"'

xt is the method for *selector* in the current class.

`[current]` *compile-time:* "*selector*" – *; run-time: ... object* – *...* objects "[current]"

Compile the method for *selector* in the current class.

`current-interface` – *addr* objects "current-interface"

Variable: contains the class or interface currently being defined.

`dict-new` *... class* – *object* objects "dict-new"

`allot` and initialize an object of class *class* in the dictionary.

`end-class` *align offset* "*name*" – objects "end-class"

name execution: `-- class`
End a class definition. The resulting class is *class*.

`end-class-noname` *align offset* – *class* objects "end-class-noname"

End a class definition. The resulting class is *class*.

`end-interface` "*name*" – objects "end-interface"

`name` execution: `-- interface`
End an interface definition. The resulting interface is *interface*.

`end-interface-noname` – *interface* objects "end-interface-noname"

End an interface definition. The resulting interface is *interface*.

`end-methods` – objects "end-methods"

Switch back from defining methods of a class to normal mode (currently this just restores the old search order).

`exitm` – objects "exitm"

`exit` from a method; restore old `this`.

`heap-new` *... class* – *object* objects "heap-new"

`allocate` and initialize an object of class *class*.

`implementation` *interface* – objects "implementation"

The current class implements *interface*. I.e., you can use all selectors of the interface in the current class and its descendents.

`init-object` *... class object* – objects "init-object"

Initialize a chunk of memory (*object*) to an object of class *class*; then performs `construct`.

`inst-value` *align1 offset1* "*name*" – *align2 offset2* objects "inst-value"

name execution: `-- w`

w is the value of the field *name* in `this` object.

`inst-var` *align1 offset1 align size* "*name*" – *align2 offset2* objects "inst-var"

name execution: `-- addr`

addr is the address of the field *name* in `this` object.

`interface` – objects "interface"

Start an interface definition.

`m:` – *xt colon-sys; run-time: object* – objects "m:"

Start a method definition; *object* becomes new `this`.

`:m` "*name*" – *xt; run-time: object* – objects ":m"

Start a named method definition; *object* becomes new `this`. Has to be ended with `;m`.

`;m` *colon-sys –; run-time:* – objects ";m"

End a method definition; restore old `this`.

`method` *xt* "*name*" – objects "method"

`name` execution: `... object -- ...`

Create selector *name* and makes *xt* its method in the current class.

`methods` *class* – objects "methods"

Makes *class* the current class. This is intended to be used for defining methods to override selectors; you cannot define new fields or selectors.

`object` – *class* objects "object"

the ancestor of all classes.

`overrides` *xt* "*selector*" – objects "overrides"

replace default method for *selector* in the current class with *xt*. `overrides` must not be used during an interface definition.

`[parent]` *compile-time:* "*selector*" *– ; run-time: ... object – ...* objects "[parent]"

Compile the method for *selector* in the parent of the current class.

`print` *object* – objects "print"

Print the object. The method for the class *object* prints the address of the object and the address of its class.

`protected` – objects "protected"

Set the compilation wordlist to the current class's wordlist

`public` – objects "public"

Restore the compilation wordlist that was in effect before the last `protected` that actually changed the compilation wordlist.

`selector` "*name*" – objects "selector"

name execution: `... object -- ...`

Create selector *name* for the current class and its descendents; you can set a method for the selector in the current class with `overrides`.

`this` – *object* objects "this"

the receiving object of the current method (aka active object).

`<to-inst>` *w xt* – objects "<to-inst>"

store *w* into the field *xt* in `this` object.

`[to-inst]` *compile-time:* "*name*" *– ; run-time: w –* objects "[to-inst]"

store *w* into field *name* in `this` object.

`to-this` *object –* objects "to-this"

Set `this` (used internally, but useful when debugging).

`xt-new` *... class xt – object* objects "xt-new"

Make a new object, using `xt ( align size -- addr )` to get memory.

5.22.4 The 'oof.fs' model

This section describes the 'oof.fs' package.

The package described in this section has been used in bigFORTH since 1991, and used for two large applications: a chromatographic system used to create new medicaments, and a graphic user interface library (MINOS).

You can find a description (in German) of 'oof.fs' in *Object oriented bigFORTH* by Bernd Paysan, published in *Vierte Dimension* 10(2), 1994.

5.22.4.1 Properties of the 'oof.fs' model

- This model combines object oriented programming with information hiding. It helps you writing large application, where scoping is necessary, because it provides class-oriented scoping.
- Named objects, object pointers, and object arrays can be created, selector invocation uses the "object selector" syntax. Selector invocation to objects and/or selectors on the stack is a bit less convenient, but possible.
- Selector invocation and instance variable usage of the active object is straightforward, since both make use of the active object.
- Late binding is efficient and easy to use.
- State-smart objects parse selectors. However, extensibility is provided using a (parsing) selector `postpone` and a selector `'`.
- An implementation in ANS Forth is available.

5.22.4.2 Basic 'oof.fs' Usage

This section uses the same example as for `objects` (see Section 5.22.3.2 [Basic Objects Usage], page 134).

You can define a class for graphical objects like this:

```
object class graphical \ "object" is the parent class
  method draw ( x y graphical -- )
class;
```

This code defines a class `graphical` with an operation `draw`. We can perform the operation `draw` on any `graphical` object, e.g.:

```
100 100 t-rex draw
```

where `t-rex` is an object or object pointer, created with e.g. `graphical : t-rex`.

How do we create a graphical object? With the present definitions, we cannot create a useful graphical object. The class `graphical` describes graphical objects in general, but not any concrete graphical object type (C++ users would call it an *abstract class*); e.g., there is no method for the selector `draw` in the class `graphical`.

For concrete graphical objects, we define child classes of the class `graphical`, e.g.:

```
graphical class circle \ "graphical" is the parent class
  cell var circle-radius
how:
  : draw ( x y -- )
    circle-radius @ draw-circle ;

  : init ( n-radius -- (
    circle-radius ! ;
class;
```

Here we define a class `circle` as a child of `graphical`, with a field `circle-radius`; it defines new methods for the selectors `draw` and `init` (`init` is defined in `object`, the parent class of `graphical`).

Now we can create a circle in the dictionary with:

```
50 circle : my-circle
```

`:` invokes `init`, thus initializing the field `circle-radius` with 50. We can draw this new circle at (100,100) with:

```
100 100 my-circle draw
```

Note: You can only invoke a selector if the receiving object belongs to the class where the selector was defined or one of its descendents; e.g., you can invoke `draw` only for objects belonging to `graphical` or its descendents (e.g., `circle`). The scoping mechanism will check if you try to invoke a selector that is not defined in this class hierarchy, so you'll get an error at compilation time.

5.22.4.3 The '`oof.fs`' base class

When you define a class, you have to specify a parent class. So how do you start defining classes? There is one class available from the start: `object`. You have to use it as ancestor for all classes. It is the only class that has no parent. Classes are also objects, except that they don't have instance variables; class manipulation such as inheritance or changing definitions of a class is handled through selectors of the class `object`.

`object` provides a number of selectors:

- `class` for subclassing, `definitions` to add definitions later on, and `class?` to get type informations (is the class a subclass of the class passed on the stack?).

  `class` "*name*" – oof "class"

  `definitions` – oof "definitions"

  `class?` *o – flag* oof "class-query"

- `init` and `dispose` as constructor and destructor of the object. `init` is invocated after the object's memory is allocated, while `dispose` also handles deallocation. Thus if you redefine `dispose`, you have to call the parent's dispose with `super dispose`, too.

  `init` *...* – oof "init"

  `dispose` – oof "dispose"

- `new`, `new[]`, `:`, `ptr`, `asptr`, and `[]` to create named and unnamed objects and object arrays or object pointers.

  `new` – *o* oof "new"

  `new[]` *n* – *o* oof "new-array"

  `:` "*name*" – oof "define"

  `ptr` "*name*" – oof "ptr"

  `asptr` *o* "*name*" – oof "asptr"

  `[]` *n* "*name*" – oof "array"

- `::` and `super` for explicit scoping. You should use explicit scoping only for super classes or classes with the same set of instance variables. Explicitly-scoped selectors use early binding.

  `::` "*name*" – oof "scope"

  `super` "*name*" – oof "super"

- `self` to get the address of the object

  `self` – *o* oof "self"

- `bind`, `bound`, `link`, and `is` to assign object pointers and instance defers.

  `bind` *o* "*name*" – oof "bind"

  `bound` *class addr* "*name*" – oof "bound"

  `link` "*name*" – *class addr* oof "link"

  `is` *xt* "*name*" – oof "is"

- `'` to obtain selector tokens, `send` to invocate selectors form the stack, and `postpone` to generate selector invocation code.

  `'` "*name*" – *xt* oof "tick"

  `postpone` "*name*" – oof "postpone"

- `with` and `endwith` to select the active object from the stack, and enable its scope. Using `with` and `endwith` also allows you to create code using selector `postpone` without being trapped by the state-smart objects.

  `with` *o* – oof "with"

  `endwith` – oof "endwith"

5.22.4.4 Class Declaration

- Instance variables

  `var` *size* – oof "var"

  Create an instance variable

- Object pointers

`ptr` – oof "ptr"

Create an instance pointer

`asptr` *class* – oof "asptr"

Create an alias to an instance pointer, cast to another class.

- Instance defers

  `defer` – oof "defer"

  Create an instance defer

- Method selectors

  `early` – oof "early"

  Create a method selector for early binding.

  `method` – oof "method"

  Create a method selector.

- Class-wide variables

  `static` – oof "static"

  Create a class-wide cell-sized variable.

- End declaration

  `how:` – oof "how-to"

  End declaration, start implementation

  `class;` – oof "end-class"

  End class declaration or implementation

5.22.4.5 Class Implementation

5.22.5 The 'mini-oof.fs' model

Gforth's third object oriented Forth package is a 12-liner. It uses a mixture of the '`objects.fs`' and the '`oof.fs`' syntax, and reduces to the bare minimum of features. This is based on a posting of Bernd Paysan in comp.lang.forth.

5.22.5.1 Basic 'mini-oof.fs' Usage

There is a base class (`class`, which allocates one cell for the object pointer) plus seven other words: to define a method, a variable, a class; to end a class, to resolve binding, to allocate an object and to compile a class method.

`object` – *a-addr* mini-oof "object"

object is the base class of all objects.

`method` *m v* "*name*" – *m' v* mini-oof "method"

Define a selector.

`var` *m v size* "*name*" – *m v'* mini-oof "var"

Define a variable with *size* bytes.

`class` *class – class selectors vars* mini-oof "class"

Start the definition of a class.

`end-class` *class selectors vars "name"* – mini-oof "end-class"

End the definition of a class.

`defines` *xt class "name"* – mini-oof "defines"

Bind *xt* to the selector *name* in class *class*.

`new` *class* – *o* mini-oof "new"

Create a new incarnation of the class *class*.

`::` *class "name"* – mini-oof "colon-colon"

Compile the method for the selector *name* of the class *class* (not immediate!).

5.22.5.2 Mini-OOF Example

A short example shows how to use this package. This example, in slightly extended form, is supplied as '`moof-exm.fs`'

```
object class
  method init
  method draw
end-class graphical
```

This code defines a class `graphical` with an operation `draw`. We can perform the operation `draw` on any `graphical` object, e.g.:

```
100 100 t-rex draw
```

where `t-rex` is an object or object pointer, created with e.g. `graphical new Constant t-rex`.

For concrete graphical objects, we define child classes of the class `graphical`, e.g.:

```
graphical class
  cell var circle-radius
end-class circle \ "graphical" is the parent class

:noname ( x y -- )
  circle-radius @ draw-circle ; circle defines draw
:noname ( r -- )
  circle-radius ! ; circle defines init
```

There is no implicit init method, so we have to define one. The creation code of the object now has to call init explicitely.

```
circle new Constant my-circle
50 my-circle init
```

It is also possible to add a function to create named objects with automatic call of `init`, given that all objects have `init` on the same place:

```
: new: ( .. o "name" -- )
    new dup Constant init ;
80 circle new: large-circle
```

We can draw this new circle at (100,100) with:

```
100 100 my-circle draw
```

5.22.5.3 'mini-oof.fs' Implementation

Object-oriented systems with late binding typically use a "vtable"-approach: the first variable in each object is a pointer to a table, which contains the methods as function pointers. The vtable may also contain other information.

So first, let's declare selectors:

```
: method ( m v "name" -- m' v ) Create  over , swap cell+ swap
  DOES> ( ... o -- ... ) @ over @ + @ execute ;
```

During selector declaration, the number of selectors and instance variables is on the stack (in address units). `method` creates one selector and increments the selector number. To execute a selector, it takes the object, fetches the vtable pointer, adds the offset, and executes the method *xt* stored there. Each selector takes the object it is invoked with as top of stack parameter; it passes the parameters (including the object) unchanged to the appropriate method which should consume that object.

Now, we also have to declare instance variables

```
: var ( m v size "name" -- m v' ) Create  over , +
  DOES> ( o -- addr ) @ + ;
```

As before, a word is created with the current offset. Instance variables can have different sizes (cells, floats, doubles, chars), so all we do is take the size and add it to the offset. If your machine has alignment restrictions, put the proper `aligned` or `faligned` before the variable, to adjust the variable offset. That's why it is on the top of stack.

We need a starting point (the base object) and some syntactic sugar:

```
Create object  1 cells , 2 cells ,
: class ( class -- class selectors vars ) dup 2@ ;
```

For inheritance, the vtable of the parent object has to be copied when a new, derived class is declared. This gives all the methods of the parent class, which can be overridden, though.

```
: end-class  ( class selectors vars "name" -- )
  Create  here >r , dup , 2 cells ?DO ['] noop , 1 cells +LOOP
  cell+ dup cell+ r> rot @ 2 cells /string move ;
```

The first line creates the vtable, initialized with `noops`. The second line is the inheritance mechanism, it copies the xts from the parent vtable.

We still have no way to define new methods, let's do that now:

```
: defines ( xt class "name" -- ) ' >body @ + ! ;
```

To allocate a new object, we need a word, too:

```
: new ( class -- o )  here over @ allot swap over ! ;
```

Sometimes derived classes want to access the method of the parent object. There are two ways to achieve this with Mini-OOF: first, you could use named words, and second, you could look up the vtable of the parent object.

```
: :: ( class "name" -- ) ' >body @ + @ compile, ;
```

Nothing can be more confusing than a good example, so here is one. First let's declare a text object (called `button`), that stores text and position:

```
object class
  cell var text
  cell var len
  cell var x
  cell var y
  method init
  method draw
end-class button
```

Now, implement the two methods, `draw` and `init`:

```
:noname ( o -- )
 >r r@ x @ r@ y @ at-xy  r@ text @ r> len @ type ;
 button defines draw
:noname ( addr u o -- )
 >r 0 r@ x ! 0 r@ y ! r@ len ! r> text ! ;
 button defines init
```

To demonstrate inheritance, we define a class `bold-button`, with no new data and no new selectors:

```
button class
end-class bold-button

: bold   27 emit ." [1m" ;
: normal 27 emit ." [0m" ;
```

The class `bold-button` has a different draw method to `button`, but the new method is defined in terms of the draw method for `button`:

```
:noname bold [ button :: draw ] normal ; bold-button defines draw
```

Finally, create two objects and apply selectors:

```
button new Constant foo
s" thin foo" foo init
page
foo draw
bold-button new Constant bar
s" fat bar" bar init
1 bar y !
bar draw
```

5.22.6 Comparison with other object models

Many object-oriented Forth extensions have been proposed (*A survey of object-oriented Forths* (SIGPLAN Notices, April 1996) by Bradford J. Rodriguez and W. F. S. Poehlman lists 17). This section discusses the relation of the object models described here to two well-known and two closely-related (by the use of method maps) models. Andras Zsoter helped us with this section.

The most popular model currently seems to be the Neon model (see *Object-oriented programming in ANS Forth* (Forth Dimensions, March 1997) by Andrew McKewan) but this model has a number of limitations[27]:

[27] A longer version of this critique can be found in *On Standardizing Object-Oriented Forth Extensions* (Forth Dimensions, May 1997) by Anton Ertl.

- It uses a *selector object* syntax, which makes it unnatural to pass objects on the stack.
- It requires that the selector parses the input stream (at compile time); this leads to reduced extensibility and to bugs that are hard to find.
- It allows using every selector on every object; this eliminates the need for interfaces, but makes it harder to create efficient implementations.

Another well-known publication is *Object-Oriented Forth* (Academic Press, London, 1987) by Dick Pountain. However, it is not really about object-oriented programming, because it hardly deals with late binding. Instead, it focuses on features like information hiding and overloading that are characteristic of modular languages like Ada (83).

In Does late binding have to be slow? (`http://www.forth.org/oopf.html`) (Forth Dimensions 18(1) 1996, pages 31-35) Andras Zsoter describes a model that makes heavy use of an active object (like `this` in '`objects.fs`'): The active object is not only used for accessing all fields, but also specifies the receiving object of every selector invocation; you have to change the active object explicitly with `{ ... }`, whereas in '`objects.fs`' it changes more or less implicitly at `m: ... ;m`. Such a change at the method entry point is unnecessary with Zsoter's model, because the receiving object is the active object already. On the other hand, the explicit change is absolutely necessary in that model, because otherwise no one could ever change the active object. An ANS Forth implementation of this model is available through `http://www.forth.org/oopf.html`.

The '`oof.fs`' model combines information hiding and overloading resolution (by keeping names in various word lists) with object-oriented programming. It sets the active object implicitly on method entry, but also allows explicit changing (with `>o...o>` or with `with...endwith`). It uses parsing and state-smart objects and classes for resolving overloading and for early binding: the object or class parses the selector and determines the method from this. If the selector is not parsed by an object or class, it performs a call to the selector for the active object (late binding), like Zsoter's model. Fields are always accessed through the active object. The big disadvantage of this model is the parsing and the state-smartness, which reduces extensibility and increases the opportunities for subtle bugs; essentially, you are only safe if you never tick or `postpone` an object or class (Bernd disagrees, but I (Anton) am not convinced).

The '`mini-oof.fs`' model is quite similar to a very stripped-down version of the '`objects.fs`' model, but syntactically it is a mixture of the '`objects.fs`' and '`oof.fs`' models.

5.23 Programming Tools

5.23.1 Examining data and code

The following words inspect the stack non-destructively:

`.s` – tools "dot-s"

Display the number of items on the data stack, followed by a list of the items; TOS is the right-most item.

`f.s` – gforth "f-dot-s"

Display the number of items on the floating-point stack, followed by a list of the items; TOS is the right-most item.

There is a word `.r` but it does *not* display the return stack! It is used for formatted numeric output (see Section 5.19.1 [Simple numeric output], page 113).

`depth` – `+`*n* core "depth"

`+`*n* is the number of values that were on the data stack before `+`*n* itself was placed on the stack.

`fdepth` – `+`*n* float "f-depth"

`+`*n* is the current number of (floating-point) values on the floating-point stack.

`clearstack` *...* – gforth "clear-stack"

remove and discard all/any items from the data stack.

The following words inspect memory.

`?` *a-addr* – tools "question"

Display the contents of address *a-addr* in the current number base.

`dump` *addr u* – tools "dump"

Display *u* lines of memory starting at address *addr*. Each line displays the contents of 16 bytes. When Gforth is running under an operating system you may get '`Invalid memory address`' errors if you attempt to access arbitrary locations.

And finally, `see` allows to inspect code:

`see` `"<`*spaces*`>`*name*`"` – tools "see"

Locate *name* using the current search order. Display the definition of *name*. Since this is achieved by decompiling the definition, the formatting is mechanised and some source information (comments, interpreted sequences within definitions etc.) is lost.

`xt-see` *xt* – gforth "xt-see"

Decompile the definition represented by *xt*.

`simple-see` `"`*name*`"` – gforth "simple-see"

a simple decompiler that's closer to `dump` than `see`.

`simple-see-range` *addr1 addr2* – gforth "simple-see-range"

5.23.2 Forgetting words

Forth allows you to forget words (and everything that was alloted in the dictonary after them) in a LIFO manner.

`marker` `"<`*spaces*`>` *name*`"` – core-ext "marker"

Create a definition, *name* (called a *mark*) whose execution semantics are to remove itself and everything defined after it.

The most common use of this feature is during progam development: when you change a source file, forget all the words it defined and load it again (since you also forget everything defined after the source file was loaded, you have to reload that, too). Note that effects like storing to variables and destroyed system words are not undone when you forget words. With a system like Gforth, that is fast enough at starting up and compiling, I find it more convenient to exit and restart Gforth, as this gives me a clean slate.

Here's an example of using `marker` at the start of a source file that you are debugging; it ensures that you only ever have one copy of the file's definitions compiled at any time:

```
[IFDEF] my-code
    my-code
[ENDIF]

marker my-code
init-included-files

\ .. definitions start here
\ .
\ .
\ end
```

5.23.3 Debugging

Languages with a slow edit/compile/link/test development loop tend to require sophisticated tracing/stepping debuggers to facilate debugging.

A much better (faster) way in fast-compiling languages is to add printing code at well-selected places, let the program run, look at the output, see where things went wrong, add more printing code, etc., until the bug is found.

The simple debugging aids provided in '`debugs.fs`' are meant to support this style of debugging.

The word `~~` prints debugging information (by default the source location and the stack contents). It is easy to insert. If you use Emacs it is also easy to remove (*`C-x ~`* in the Emacs Forth mode to query-replace them with nothing). The deferred words `printdebugdata` and `.debugline` control the output of `~~`. The default source location output format works well with Emacs' compilation mode, so you can step through the program at the source level using *`C-x ``* (the advantage over a stepping debugger is that you can step in any direction and you know where the crash has happened or where the strange data has occurred).

`~~` *compilation* – ; *run-time* – gforth "tilde-tilde"

`printdebugdata` – gforth "print-debug-data"

`.debugline` *nfile nline* – gforth "print-debug-line"

`~~` (and assertions) will usually print the wrong file name if a marker is executed in the same file after their occurance. They will print '`*somewhere*`' as file name if a marker is executed in the same file before their occurance.

5.23.4 Assertions

It is a good idea to make your programs self-checking, especially if you make an assumption that may become invalid during maintenance (for example, that a certain field of a data structure is never zero). Gforth supports *assertions* for this purpose. They are used like this:

`assert(` *flag* `)`

The code between `assert(` and `)` should compute a flag, that should be true if everything is alright and false otherwise. It should not change anything else on the stack. The overall stack effect of the assertion is `( -- )`. E.g.

```
assert( 1 1 + 2 = ) \ what we learn in school
assert( dup 0<> ) \ assert that the top of stack is not zero
assert( false ) \ this code should not be reached
```

The need for assertions is different at different times. During debugging, we want more checking, in production we sometimes care more for speed. Therefore, assertions can be turned off, i.e., the assertion becomes a comment. Depending on the importance of an assertion and the time it takes to check it, you may want to turn off some assertions and keep others turned on. Gforth provides several levels of assertions for this purpose:

`assert0(` – gforth "assert-zero"

Important assertions that should always be turned on.

`assert1(` – gforth "assert-one"

Normal assertions; turned on by default.

`assert2(` – gforth "assert-two"

Debugging assertions.

`assert3(` – gforth "assert-three"

Slow assertions that you may not want to turn on in normal debugging; you would turn them on mainly for thorough checking.

`assert(` – gforth "assert("

Equivalent to `assert1(`

`)` – gforth "close-paren"

End an assertion.

The variable `assert-level` specifies the highest assertions that are turned on. I.e., at the default `assert-level` of one, `assert0(` and `assert1(` assertions perform checking, while `assert2(` and `assert3(` assertions are treated as comments.

The value of `assert-level` is evaluated at compile-time, not at run-time. Therefore you cannot turn assertions on or off at run-time; you have to set the `assert-level` appropriately before compiling a piece of code. You can compile different pieces of code at different `assert-levels` (e.g., a trusted library at level 1 and newly-written code at level 3).

`assert-level` – *a-addr* gforth "assert-level"

All assertions above this level are turned off.

If an assertion fails, a message compatible with Emacs' compilation mode is produced and the execution is aborted (currently with `ABORT"`. If there is interest, we will introduce a special throw code. But if you intend to `catch` a specific condition, using `throw` is probably more appropriate than an assertion).

Assertions (and `~~`) will usually print the wrong file name if a marker is executed in the same file after their occurance. They will print '`*somewhere*`' as file name if a marker is executed in the same file before their occurance.

Definitions in ANS Forth for these assertion words are provided in '`compat/assert.fs`'.

5.23.5 Singlestep Debugger

The singlestep debugger does not work in this release.

When you create a new word there's often the need to check whether it behaves correctly or not. You can do this by typing `dbg badword`. A debug session might look like this:

```
: badword 0 DO i . LOOP ;  ok
2 dbg badword
: badword
Scanning code...

Nesting debugger ready!

400D4738  8049BC4 0             -> [ 2 ] 00002 00000
400D4740  8049F68 DO            -> [ 0 ]
400D4744  804A0C8 i             -> [ 1 ] 00000
400D4748 400C5E60 .             -> 0 [ 0 ]
400D474C  8049D0C LOOP          -> [ 0 ]
400D4744  804A0C8 i             -> [ 1 ] 00001
400D4748 400C5E60 .             -> 1 [ 0 ]
400D474C  8049D0C LOOP          -> [ 0 ]
400D4758  804B384 ;             ->  ok
```

Each line displayed is one step. You always have to hit return to execute the next word that is displayed. If you don't want to execute the next word in a whole, you have to type *n* for `nest`. Here is an overview what keys are available:

⟨RET⟩ Next; Execute the next word.

n Nest; Single step through next word.

u Unnest; Stop debugging and execute rest of word. If we got to this word with nest, continue debugging with the calling word.

d Done; Stop debugging and execute rest.

s Stop; Abort immediately.

Debugging large application with this mechanism is very difficult, because you have to nest very deeply into the program before the interesting part begins. This takes a lot of time.

To do it more directly put a `BREAK:` command into your source code. When program execution reaches `BREAK:` the single step debugger is invoked and you have all the features described above.

If you have more than one part to debug it is useful to know where the program has stopped at the moment. You can do this by the `BREAK" string"` command. This behaves like `BREAK:` except that string is typed out when the "breakpoint" is reached.

`dbg` "*name*" – gforth "dbg"

`break:` – gforth "break:"

`break"` '*ccc*"' – gforth "break""

5.24 Assembler and Code Words

5.24.1 `Code` and `;code`

Gforth provides some words for defining primitives (words written in machine code), and for defining the machine-code equivalent of `DOES>`-based defining words. However, the machine-independent nature of Gforth poses a few problems: First of all, Gforth runs on several architectures, so it can provide no standard assembler. What's worse is that the register allocation not only depends on the processor, but also on the `gcc` version and options used.

The words that Gforth offers encapsulate some system dependences (e.g., the header structure), so a system-independent assembler may be used in Gforth. If you do not have an assembler, you can compile machine code directly with `,` and `c,`[28].

`assembler` – tools-ext "assembler"

`init-asm` – gforth "init-asm"

`code` "*name*" – *colon-sys* tools-ext "code"

`end-code` *colon-sys* – gforth "end-code"

`;code` *compilation. colon-sys1 – colon-sys2* tools-ext "semicolon-code"

`flush-icache` *c-addr u* – gforth "flush-icache"

Make sure that the instruction cache of the processor (if there is one) does not contain stale data at *c-addr* and *u* bytes afterwards. `END-CODE` performs a `flush-icache` automatically. Caveat: `flush-icache` might not work on your installation; this is usually the case if direct threading is not supported on your machine (take a look at your '`machine.h`') and your machine has a separate instruction cache. In such cases, `flush-icache` does nothing instead of flushing the instruction cache.

If `flush-icache` does not work correctly, `code` words etc. will not work (reliably), either.

The typical usage of these `code` words can be shown most easily by analogy to the equivalent high-level defining words:

```
: foo                             code foo
   <high-level Forth words>            <assembler>
;                                 end-code

: bar                             : bar
   <high-level Forth words>          <high-level Forth words>
   CREATE                            CREATE
      <high-level Forth words>          <high-level Forth words>
   DOES>                             ;code
      <high-level Forth words>          <assembler>
;                                 end-code
```

In the assembly code you will want to refer to the inner interpreter's registers (e.g., the data stack pointer) and you may want to use other registers for temporary storage. Unfortunately, the register allocation is installation-dependent.

[28] This isn't portable, because these words emit stuff in *data* space; it works because Gforth has unified code/data spaces. Assembler isn't likely to be portable anyway.

In particular, `ip` (Forth instruction pointer) and `rp` (return stack pointer) may be in different places in `gforth` and `gforth-fast`, or different installations. This means that you cannot write a `NEXT` routine that works reliably on both versions or different installations; so for doing `NEXT`, I recommend jumping to `' noop >code-address`, which contains nothing but a `NEXT`.

For general accesses to the inner interpreter's registers, the easiest solution is to use explicit register declarations (see section "Variables in Specified Registers" in *GNU C Manual*) for all of the inner interpreter's registers: You have to compile Gforth with `-DFORCE_REG` (configure option `--enable-force-reg`) and the appropriate declarations must be present in the `machine.h` file (see `mips.h` for an example; you can find a full list of all declarable register symbols with `grep register engine.c`). If you give explicit registers to all variables that are declared at the beginning of `engine()`, you should be able to use the other caller-saved registers for temporary storage. Alternatively, you can use the `gcc` option `-ffixed-REG` (see section "Options for Code Generation Conventions" in *GNU C Manual*) to reserve a register (however, this restriction on register allocation may slow Gforth significantly).

If this solution is not viable (e.g., because `gcc` does not allow you to explicitly declare all the registers you need), you have to find out by looking at the code where the inner interpreter's registers reside and which registers can be used for temporary storage. You can get an assembly listing of the engine's code with `make engine.s`.

In any case, it is good practice to abstract your assembly code from the actual register allocation. E.g., if the data stack pointer resides in register `$17`, create an alias for this register called `sp`, and use that in your assembly code.

Another option for implementing normal and defining words efficiently is to add the desired functionality to the source of Gforth. For normal words you just have to edit '`primitives`' (see Section 14.3.1 [Automatic Generation], page 196). Defining words (equivalent to `;CODE` words, for fast defined words) may require changes in '`engine.c`', '`kernel.fs`', '`prims2x.fs`', and possibly '`cross.fs`'.

5.24.2 Common Assembler

The assemblers in Gforth generally use a postfix syntax, i.e., the instruction name follows the operands.

The operands are passed in the usual order (the same that is used in the manual of the architecture). Since they all are Forth words, they have to be separated by spaces; you can also use Forth words to compute the operands.

The instruction names usually end with a `,`. This makes it easier to visually separate instructions if you put several of them on one line; it also avoids shadowing other Forth words (e.g., `and`).

Registers are usually specified by number; e.g., (decimal) `11` specifies registers R11 and F11 on the Alpha architecture (which one, depends on the instruction). The usual names are also available, e.g., `s2` for R11 on Alpha.

Control flow is specified similar to normal Forth code (see Section 5.8.4 [Arbitrary control structures], page 68), with `if,`, `ahead,`, `then,`, `begin,`, `until,`, `again,`, `cs-roll`, `cs-pick`, `else,`, `while,`, and `repeat,`. The conditions are specified in a way specific to each assembler.

Note that the register assignments of the Gforth engine can change between Gforth versions, or even between different compilations of the same Gforth version (e.g., if you use a different GCC version). So if you want to refer to Gforth's registers (e.g., the stack pointer or TOS), I recommend defining your own words for refering to these registers, and using them later on; then you can easily adapt to a changed register assignment. The stability of the register assignment is usually better if you build Gforth with `--enable-force-reg`.

The most common use of these registers is to dispatch to the next word (the `next` routine). A portable way to do this is to jump to `' noop >code-address` (of course, this is less efficient than integrating the `next` code and scheduling it well).

Another difference between Gforth version is that the top of stack is kept in memory in `gforth` and, on most platforms, in a register in `gforth-fast`.

5.24.3 Common Disassembler

You can disassemble a `code` word with `see` (see Section 5.23.3 [Debugging], page 153). You can disassemble a section of memory with

doc-disasm

The disassembler generally produces output that can be fed into the assembler (i.e., same syntax, etc.). It also includes additional information in comments. In particular, the address of the instruction is given in a comment before the instruction.

`See` may display more or less than the actual code of the word, because the recognition of the end of the code is unreliable. You can use `disasm` if it did not display enough. It may display more, if the code word is not immediately followed by a named word. If you have something else there, you can follow the word with `align last  ,` to ensure that the end is recognized.

5.24.4 386 Assembler

The 386 assembler included in Gforth was written by Bernd Paysan, it's available under GPL, and originally part of bigFORTH.

The 386 disassembler included in Gforth was written by Andrew McKewan and is in the public domain.

The disassembler displays code in an Intel-like prefix syntax.

The assembler uses a postfix syntax with reversed parameters.

The assembler includes all instruction of the Athlon, i.e. 486 core instructions, Pentium and PPro extensions, floating point, MMX, 3Dnow!, but not ISSE. It's an integrated 16- and 32-bit assembler. Default is 32 bit, you can switch to 16 bit with .86 and back to 32 bit with .386.

There are several prefixes to switch between different operation sizes, `.b` for byte accesses, `.w` for word accesses, `.d` for double-word accesses. Addressing modes can be switched with `.wa` for 16 bit addresses, and `.da` for 32 bit addresses. You don't need a prefix for byte register names (`AL` et al).

For floating point operations, the prefixes are `.fs` (IEEE single), `.fl` (IEEE double), `.fx` (extended), `.fw` (word), `.fd` (double-word), and `.fq` (quad-word).

The MMX opcodes don't have size prefixes, they are spelled out like in the Intel assembler. Instead of move from and to memory, there are PLDQ/PLDD and PSTQ/PSTD.

The registers lack the 'e' prefix; even in 32 bit mode, eax is called ax. Immediate values are indicated by postfixing them with `#`, e.g., `3 #`. Here are some examples of addressing modes in various syntaxes:

Gforth	Intel (NASM)	AT&T (gas)	Name
.w ax	ax	%ax	register (16 bit)
ax	eax	%eax	register (32 bit)
3 #	offset 3	$3	immediate
1000 #)	byte ptr 1000	1000	displacement
bx)	[ebx]	(%ebx)	base
100 di d)	100[edi]	100(%edi)	base+displacement
20 ax *4 i#)	20[eax*4]	20(,%eax,4)	(index*scale)+displacement
di ax *4 i)	[edi][eax*4]	(%edi,%eax,4)	base+(index*scale)
4 bx cx di)	4[ebx][ecx]	4(%ebx,%ecx)	base+index+displacement
12 sp ax *2 di)	12[esp][eax*2]	12(%esp,%eax,2)	base+(index*scale)+displacement

You can use `L)` and `LI)` instead of `D)` and `DI)` to enforce 32-bit displacement fields (useful for later patching).

Some example of instructions are:

```
ax bx mov             \ move ebx,eax
3 # ax mov            \ mov eax,3
100 di ) ax mov       \ mov eax,100[edi]
4 bx cx di) ax mov    \ mov eax,4[ebx][ecx]
.w ax bx mov          \ mov bx,ax
```

The following forms are supported for binary instructions:

```
<reg> <reg> <inst>
<n> # <reg> <inst>
<mem> <reg> <inst>
<reg> <mem> <inst>
```

Immediate to memory is not supported. The shift/rotate syntax is:

```
<reg/mem> 1 # shl \ shortens to shift without immediate
<reg/mem> 4 # shl
<reg/mem> cl shl
```

Precede string instructions (`movs` etc.) with `.b` to get the byte version.

The control structure words `IF UNTIL` etc. must be preceded by one of these conditions: `vs vc u< u>= 0= 0<> u<= u> 0< 0>= ps pc < >= <= >`. (Note that most of these words shadow some Forth words when `assembler` is in front of `forth` in the search path, e.g., in `code` words). Currently the control structure words use one stack item, so you have to use `roll` instead of `cs-roll` to shuffle them (you can also use `swap` etc.).

Here is an example of a `code` word (assumes that the stack pointer is in esi and the TOS is in ebx):

```
code my+ ( n1 n2 -- n )
    4 si D) bx add
    4 # si add
    Next
end-code
```

5.24.5 Alpha Assembler

The Alpha assembler and disassembler were originally written by Bernd Thallner.

The register names `a0`–`a5` are not available to avoid shadowing hex numbers.

Immediate forms of arithmetic instructions are distinguished by a `#` just before the `,`, e.g., `and#,` (note: `lda,` does not count as arithmetic instruction).

You have to specify all operands to an instruction, even those that other assemblers consider optional, e.g., the destination register for `br,`, or the destination register and hint for `jmp,`.

You can specify conditions for `if,` by removing the first `b` and the trailing `,` from a branch with a corresponding name; e.g.,

```
11 fgt if, \ if F11>0e
  ...
endif,
```

`fbgt,` gives `fgt`.

5.24.6 MIPS assembler

The MIPS assembler was originally written by Christian Pirker.

Currently the assembler and disassembler only cover the MIPS-I architecture (R3000), and don't support FP instructions.

The register names `$a0`–`$a3` are not available to avoid shadowing hex numbers.

Because there is no way to distinguish registers from immediate values, you have to explicitly use the immediate forms of instructions, i.e., `addiu,`, not just `addu,` (`as` does this implicitly).

If the architecture manual specifies several formats for the instruction (e.g., for `jalr,`), you usually have to use the one with more arguments (i.e., two for `jalr,`). When in doubt, see `arch/mips/testasm.fs` for an example of correct use.

Branches and jumps in the MIPS architecture have a delay slot. You have to fill it yourself (the simplest way is to use `nop,`), the assembler does not do it for you (unlike `as`). Even `if,`, `ahead,`, `until,`, `again,`, `while,`, `else,` and `repeat,` need a delay slot. Since `begin,` and `then,` just specify branch targets, they are not affected.

Note that you must not put branches, jumps, or `li,` into the delay slot: `li,` may expand to several instructions, and control flow instructions may not be put into the branch delay slot in any case.

For branches the argument specifying the target is a relative address; You have to add the address of the delay slot to get the absolute address.

The MIPS architecture also has load delay slots and restrictions on using `mfhi,` and `mflo,`; you have to order the instructions yourself to satisfy these restrictions, the assembler does not do it for you.

You can specify the conditions for `if,` etc. by taking a conditional branch and leaving away the `b` at the start and the `,` at the end. E.g.,

```
4 5 eq if,
  ... \ do something if $4 equals $5
then,
```

5.24.7 Other assemblers

If you want to contribute another assembler/disassembler, please contact us (`anton@mips.complang.tuwien.ac.at`) to check if we have such an assembler already. If you are writing them from scratch, please use a similar syntax style as the one we use (i.e., postfix, commas at the end of the instruction names, see Section 5.24.2 [Common Assembler], page 157); make the output of the disassembler be valid input for the assembler, and keep the style similar to the style we used.

Hints on implementation: The most important part is to have a good test suite that contains all instructions. Once you have that, the rest is easy. For actual coding you can take a look at '`arch/mips/disasm.fs`' to get some ideas on how to use data for both the assembler and disassembler, avoiding redundancy and some potential bugs. You can also look at that file (and see Section 5.9.8.3 [Advanced `does>` usage example], page 80) to get ideas how to factor a disassembler.

Start with the disassembler, because it's easier to reuse data from the disassembler for the assembler than the other way round.

For the assembler, take a look at '`arch/alpha/asm.fs`', which shows how simple it can be.

5.25 Threading Words

These words provide access to code addresses and other threading stuff in Gforth (and, possibly, other interpretive Forths). It more or less abstracts away the differences between direct and indirect threading (and, for direct threading, the machine dependences). However, at present this wordset is still incomplete. It is also pretty low-level; some day it will hopefully be made unnecessary by an internals wordset that abstracts implementation details away completely.

The terminology used here stems from indirect threaded Forth systems; in such a system, the XT of a word is represented by the CFA (code field address) of a word; the CFA points to a cell that contains the code address. The code address is the address of some machine code that performs the run-time action of invoking the word (e.g., the `dovar:` routine pushes the address of the body of the word (a variable) on the stack).

In an indirect threaded Forth, you can get the code address of *name* with `'` *name* `@`; in Gforth you can get it with `'` *name* `>code-address`, independent of the threading method.

`threading-method` – *n* gforth "threading-method"

0 if the engine is direct threaded. Note that this may change during the lifetime of an image.

`>code-address` *xt* – *c_addr* gforth ">code-address"

c-addr is the code address of the word *xt*.

`code-address!` *c_addr xt* – gforth "code-address!"

Create a code field with code address *c-addr* at *xt*.

For a word defined with `DOES>`, the code address usually points to a jump instruction (the *does-handler*) that jumps to the dodoes routine (in Gforth on some platforms, it can also point to the dodoes routine itself). What you are typically interested in, though, is whether a word is a `DOES>`-defined word, and what Forth code it executes; `>does-code` tells you that.

`>does-code` *xt – a_addr* gforth ">does-code"

If *xt* is the execution token of a child of a `DOES>` word, *a-addr* is the start of the Forth code after the `DOES>`; Otherwise *a-addr* is 0.

To create a `DOES>`-defined word with the following basic words, you have to set up a `DOES>`-handler with `does-handler!`; `/does-handler` aus behind you have to place your executable Forth code. Finally you have to create a word and modify its behaviour with `does-handler!`.

`does-code!` *a_addr xt –* gforth "does-code!"

Create a code field at *xt* for a child of a `DOES>`-word; *a-addr* is the start of the Forth code after `DOES>`.

`does-handler!` *a_addr –* gforth "does-handler!"

Create a `DOES>`-handler at address *a-addr*. Normally, *a-addr* points just behind a `DOES>`.

`/does-handler` *– n* gforth "/does-handler"

The size of a `DOES>`-handler (includes possible padding).

The code addresses produced by various defining words are produced by the following words:

`docol:` *– addr* gforth "docol:"

The code address of a colon definition.

`docon:` *– addr* gforth "docon:"

The code address of a `CONSTANT`.

`dovar:` *– addr* gforth "dovar:"

The code address of a `CREATE`d word.

`douser:` *– addr* gforth "douser:"

The code address of a `USER` variable.

`dodefer:` *– addr* gforth "dodefer:"

The code address of a `defer`ed word.

`dofield:` *– addr* gforth "dofield:"

The code address of a `field`.

The following two words generalize `>code-address`, `>does-code`, `code-address!`, and `does-code!`:

`>definer` *xt – definer* unknown ">definer"

Definer is a unique identifier for the way the *xt* was defined. Words defined with different `does>`-codes have different definers. The definer can be used for comparison and in `definer!`.

`definer!` *definer xt –* unknown "definer!"

The word represented by *xt* changes its behaviour to the behaviour associated with *definer*.

5.26 Passing Commands to the Operating System

Gforth allows you to pass an arbitrary string to the host operating system shell (if such a thing exists) for execution.

`sh` "..." – gforth "sh"

Parse a string and use `system` to pass it to the host operating system for execution in a sub-shell.

`system` *c-addr u* – gforth "system"

Pass the string specified by *c-addr u* to the host operating system for execution in a sub-shell.

`$?` – *n* gforth "dollar-question"

`Value` – the exit status returned by the most recently executed `system` command.

`getenv` *c-addr1 u1* – *c-addr2 u2* gforth "getenv"

The string *c-addr1 u1* specifies an environment variable. The string *c-addr2 u2* is the host operating system's expansion of that environment variable. If the environment variable does not exist, *c-addr2 u2* specifies a string 0 characters in length.

5.27 Keeping track of Time

`ms` *n* – facility-ext "ms"

Wait at least *n* milli-second.

`time&date` – *nsec nmin nhour nday nmonth nyear* facility-ext "time-and-date"

Report the current time of day. Seconds, minutes and hours are numbered from 0. Months are numbered from 1.

`utime` – *dtime* gforth "utime"

Report the current time in microseconds since some epoch.

`cputime` – *duser dsystem* gforth "cputime"

duser and dsystem are the respective user- and system-level CPU times used since the start of the Forth system (excluding child processes), in microseconds (the granularity may be much larger, however). On platforms without the getrusage call, it reports elapsed time (since some epoch) for duser and 0 for dsystem.

5.28 Miscellaneous Words

These section lists the ANS Forth words that are not documented elsewhere in this manual. Ultimately, they all need proper homes.

`quit` *?? – ??* core "quit"

Empty the return stack, make the user input device the input source, enter interpret state and start the text interpreter.

The following ANS Forth words are not currently supported by Gforth (see Chapter 8 [ANS conformance], page 166):

`EDITOR` `EMIT?` `FORGET`

6 Error messages

A typical Gforth error message looks like this:

```
in file included from \evaluated string/:-1
in file included from ./yyy.fs:1
./xxx.fs:4: Invalid memory address
bar
^^^
Backtrace:
$400E664C @
$400E6664 foo
```

The message identifying the error is `Invalid memory address`. The error happened when text-interpreting line 4 of the file '`./xxx.fs`'. This line is given (it contains `bar`), and the word on the line where the error happened, is pointed out (with `^^^`).

The file containing the error was included in line 1 of '`./yyy.fs`', and '`yyy.fs`' was included from a non-file (in this case, by giving '`yyy.fs`' as command-line parameter to Gforth).

At the end of the error message you find a return stack dump that can be interpreted as a backtrace (possibly empty). On top you find the top of the return stack when the `throw` happened, and at the bottom you find the return stack entry just above the return stack of the topmost text interpreter.

To the right of most return stack entries you see a guess for the word that pushed that return stack entry as its return address. This gives a backtrace. In our case we see that `bar` called `foo`, and `foo` called `@` (and `@` had an *Invalid memory address* exception).

Note that the backtrace is not perfect: We don't know which return stack entries are return addresses (so we may get false positives); and in some cases (e.g., for `abort"`) we cannot determine from the return address the word that pushed the return address, so for some return addresses you see no names in the return stack dump.

The return stack dump represents the return stack at the time when a specific `throw` was executed. In programs that make use of `catch`, it is not necessarily clear which `throw` should be used for the return stack dump (e.g., consider one `throw` that indicates an error, which is caught, and during recovery another error happens; which `throw` should be used for the stack dump?). Gforth presents the return stack dump for the first `throw` after the last executed (not returned-to) `catch`; this works well in the usual case.

`Gforth` is able to do a return stack dump for throws generated from primitives (e.g., invalid memory address, stack empty etc.); `gforth-fast` is only able to do a return stack dump from a directly called `throw` (including `abort` etc.). Given an exception caused by a primitive in `gforth-fast`, you will typically see no return stack dump at all; however, if the exception is caught by `catch` (e.g., for restoring some state), and then `throw`n again, the return stack dump will be for the first such `throw`.

7 Tools

See also Chapter 12 [Emacs and Gforth], page 184.

7.1 'ans-report.fs': Report the words used, sorted by wordset

If you want to label a Forth program as ANS Forth Program, you must document which wordsets the program uses; for extension wordsets, it is helpful to list the words the program requires from these wordsets (because Forth systems are allowed to provide only some words of them).

The '`ans-report.fs`' tool makes it easy for you to determine which words from which wordset and which non-ANS words your application uses. You simply have to include '`ans-report.fs`' before loading the program you want to check. After loading your program, you can get the report with `print-ans-report`. A typical use is to run this as batch job like this:

```
gforth ans-report.fs myprog.fs -e "print-ans-report bye"
```

The output looks like this (for '`compat/control.fs`'):

```
The program uses the following words
from CORE :
: POSTPONE THEN ; immediate ?dup IF 0=
from BLOCK-EXT :
\
from FILE :
(
```

7.1.1 Caveats

Note that '`ans-report.fs`' just checks which words are used, not whether they are used in an ANS Forth conforming way!

Some words are defined in several wordsets in the standard. '`ans-report.fs`' reports them for only one of the wordsets, and not necessarily the one you expect. It depends on usage which wordset is the right one to specify. E.g., if you only use the compilation semantics of `S"`, it is a Core word; if you also use its interpretation semantics, it is a File word.

8 ANS conformance

To the best of our knowledge, Gforth is an

ANS Forth System

- providing the Core Extensions word set
- providing the Block word set
- providing the Block Extensions word set
- providing the Double-Number word set
- providing the Double-Number Extensions word set
- providing the Exception word set
- providing the Exception Extensions word set
- providing the Facility word set
- providing `EKEY`, `EKEY>CHAR`, `EKEY?`, `MS` and `TIME&DATE` from the Facility Extensions word set
- providing the File Access word set
- providing the File Access Extensions word set
- providing the Floating-Point word set
- providing the Floating-Point Extensions word set
- providing the Locals word set
- providing the Locals Extensions word set
- providing the Memory-Allocation word set
- providing the Memory-Allocation Extensions word set (that one's easy)
- providing the Programming-Tools word set
- providing `;CODE`, `AHEAD`, `ASSEMBLER`, `BYE`, `CODE`, `CS-PICK`, `CS-ROLL`, `STATE`, `[ELSE]`, `[IF]`, `[THEN]` from the Programming-Tools Extensions word set
- providing the Search-Order word set
- providing the Search-Order Extensions word set
- providing the String word set
- providing the String Extensions word set (another easy one)

In addition, ANS Forth systems are required to document certain implementation choices. This chapter tries to meet these requirements. In many cases it gives a way to ask the system for the information instead of providing the information directly, in particular, if the information depends on the processor, the operating system or the installation options chosen, or if they are likely to change during the maintenance of Gforth.

8.1 The Core Words

8.1.1 Implementation Defined Options

(Cell) aligned addresses:
: processor-dependent. Gforth's alignment words perform natural alignment (e.g., an address aligned for a datum of size 8 is divisible by 8). Unaligned accesses usually result in a `-23 THROW`.

`EMIT` *and non-graphic characters:*
: The character is output using the C library function (actually, macro) `putc`.

character editing of `ACCEPT` *and* `EXPECT`*:*
: This is modeled on the GNU readline library (see section "Command Line Editing" in *The GNU Readline Library*) with Emacs-like key bindings. *Tab* deviates a little by producing a full word completion every time you type it (instead of producing the common prefix of all completions). See Section 2.3 [Command-line editing], page 5.

character set:
: The character set of your computer and display device. Gforth is 8-bit-clean (but some other component in your system may make trouble).

Character-aligned address requirements:
: installation-dependent. Currently a character is represented by a C `unsigned char`; in the future we might switch to `wchar_t` (Comments on that requested).

character-set extensions and matching of names:
: Any character except the ASCII NUL character can be used in a name. Matching is case-insensitive (except in `TABLE`s). The matching is performed using the C library function `strncasecmp`, whose function is probably influenced by the locale. E.g., the `C` locale does not know about accents and umlauts, so they are matched case-sensitively in that locale. For portability reasons it is best to write programs such that they work in the `C` locale. Then one can use libraries written by a Polish programmer (who might use words containing ISO Latin-2 encoded characters) and by a French programmer (ISO Latin-1) in the same program (of course, `WORDS` will produce funny results for some of the words (which ones, depends on the font you are using)). Also, the locale you prefer may not be available in other operating systems. Hopefully, Unicode will solve these problems one day.

conditions under which control characters match a space delimiter:
: If `WORD` is called with the space character as a delimiter, all white-space characters (as identified by the C macro `isspace()`) are delimiters. `PARSE`, on the other hand, treats space like other delimiters. `SWORD` treats space like `WORD`, but behaves like `PARSE` otherwise. `Name`, which is used by the outer interpreter (aka text interpreter) by default, treats all white-space characters as delimiters.

format of the control-flow stack:
: The data stack is used as control-flow stack. The size of a control-flow stack item in cells is given by the constant `cs-item-size`. At the time of this writing, an item consists of a (pointer to a) locals list (third), an address in the code (second), and a tag for identifying the item (TOS). The following tags are used: `defstart`, `live-orig`, `dead-orig`, `dest`, `do-dest`, `scopestart`.

conversion of digits > 35
The characters `[\]^_'` are the digits with the decimal value 36−41. There is no way to input many of the larger digits.

display after input terminates in `ACCEPT` *and* `EXPECT`*:*
The cursor is moved to the end of the entered string. If the input is terminated using the *Return* key, a space is typed.

exception abort sequence of `ABORT"`*:*
The error string is stored into the variable `"error` and a `-2 throw` is performed.

input line terminator:
For interactive input, *C-m* (CR) and *C-j* (LF) terminate lines. One of these characters is typically produced when you type the *Enter* or *Return* key.

maximum size of a counted string:
`s" /counted-string" environment? drop .`. Currently 255 characters on all platforms, but this may change.

maximum size of a parsed string:
Given by the constant `/line`. Currently 255 characters.

maximum size of a definition name, in characters:
MAXU/8

maximum string length for `ENVIRONMENT?`*, in characters:*
MAXU/8

method of selecting the user input device:
The user input device is the standard input. There is currently no way to change it from within Gforth. However, the input can typically be redirected in the command line that starts Gforth.

method of selecting the user output device:
`EMIT` and `TYPE` output to the file-id stored in the value `outfile-id` (`stdout` by default). Gforth uses unbuffered output when the user output device is a terminal, otherwise the output is buffered.

methods of dictionary compilation:
What are we expected to document here?

number of bits in one address unit:
`s" address-units-bits" environment? drop .`. 8 in all current platforms.

number representation and arithmetic:
Processor-dependent. Binary two's complement on all current platforms.

ranges for integer types:
Installation-dependent. Make environmental queries for `MAX-N`, `MAX-U`, `MAX-D` and `MAX-UD`. The lower bounds for unsigned (and positive) types is 0. The lower bound for signed types on two's complement and one's complement machines machines can be computed by adding 1 to the upper bound.

read-only data space regions:
The whole Forth data space is writable.

size of buffer at `WORD`*:*
: `PAD HERE - .`. 104 characters on 32-bit machines. The buffer is shared with the pictured numeric output string. If overwriting `PAD` is acceptable, it is as large as the remaining dictionary space, although only as much can be sensibly used as fits in a counted string.

size of one cell in address units:
: `1 cells .`.

size of one character in address units:
: `1 chars .`. 1 on all current platforms.

size of the keyboard terminal buffer:
: Varies. You can determine the size at a specific time using `lp@ tib - .`. It is shared with the locals stack and TIBs of files that include the current file. You can change the amount of space for TIBs and locals stack at Gforth startup with the command line option `-l`.

size of the pictured numeric output buffer:
: `PAD HERE - .`. 104 characters on 32-bit machines. The buffer is shared with `WORD`.

size of the scratch area returned by `PAD`*:*
: The remainder of dictionary space. `unused pad here - - .`.

system case-sensitivity characteristics:
: Dictionary searches are case-insensitive (except in `TABLE`s). However, as explained above under *character-set extensions*, the matching for non-ASCII characters is determined by the locale you are using. In the default `C` locale all non-ASCII characters are matched case-sensitively.

system prompt:
: `ok` in interpret state, `compiled` in compile state.

division rounding:
: installation dependent. `s" floored" environment? drop .`. We leave the choice to `gcc` (what to use for `/`) and to you (whether to use `fm/mod`, `sm/rem` or simply `/`).

values of `STATE` *when true:*
: -1.

values returned after arithmetic overflow:
: On two's complement machines, arithmetic is performed modulo 2**bits-per-cell for single arithmetic and 4**bits-per-cell for double arithmetic (with appropriate mapping for signed types). Division by zero typically results in a `-55 throw` (Floating-point unidentified fault) or `-10 throw` (divide by zero).

whether the current definition can be found after `DOES>`*:*
: No.

8.1.2 Ambiguous conditions

a name is neither a word nor a number:
: `-13 throw` (Undefined word).

a definition name exceeds the maximum length allowed:
`-19 throw` (Word name too long)

addressing a region not inside the various data spaces of the forth system:
The stacks, code space and header space are accessible. Machine code space is typically readable. Accessing other addresses gives results dependent on the operating system. On decent systems: `-9 throw` (Invalid memory address).

argument type incompatible with parameter:
This is usually not caught. Some words perform checks, e.g., the control flow words, and issue a `ABORT"` or `-12 THROW` (Argument type mismatch).

attempting to obtain the execution token of a word with undefined execution semantics:
`-14 throw` (Interpreting a compile-only word). In some cases, you get an execution token for `compile-only-error` (which performs a `-14 throw` when executed).

dividing by zero:
On some platforms, this produces a `-10 throw` (Division by zero); on other systems, this typically results in a `-55 throw` (Floating-point unidentified fault).

insufficient data stack or return stack space:
Depending on the operating system, the installation, and the invocation of Gforth, this is either checked by the memory management hardware, or it is not checked. If it is checked, you typically get a `-3 throw` (Stack overflow), `-5 throw` (Return stack overflow), or `-9 throw` (Invalid memory address) (depending on the platform and how you achieved the overflow) as soon as the overflow happens. If it is not checked, overflows typically result in mysterious illegal memory accesses, producing `-9 throw` (Invalid memory address) or `-23 throw` (Address alignment exception); they might also destroy the internal data structure of `ALLOCATE` and friends, resulting in various errors in these words.

insufficient space for loop control parameters:
Like other return stack overflows.

insufficient space in the dictionary:
If you try to allot (either directly with `allot`, or indirectly with `,`, `create` etc.) more memory than available in the dictionary, you get a `-8 throw` (Dictionary overflow). If you try to access memory beyond the end of the dictionary, the results are similar to stack overflows.

interpreting a word with undefined interpretation semantics:
For some words, we have defined interpretation semantics. For the others: `-14 throw` (Interpreting a compile-only word).

modifying the contents of the input buffer or a string literal:
These are located in writable memory and can be modified.

overflow of the pictured numeric output string:
`-17 throw` (Pictured numeric ouput string overflow).

parsed string overflow:
`PARSE` cannot overflow. `WORD` does not check for overflow.

producing a result out of range:
On two's complement machines, arithmetic is performed modulo 2**bits-per-cell for single arithmetic and 4**bits-per-cell for double arithmetic (with appropriate mapping for signed types). Division by zero typically results in a `-10 throw` (divide by zero) or `-55 throw` (floating point unidentified fault). `convert` and `>number` currently overflow silently.

reading from an empty data or return stack:
The data stack is checked by the outer (aka text) interpreter after every word executed. If it has underflowed, a `-4 throw` (Stack underflow) is performed. Apart from that, stacks may be checked or not, depending on operating system, installation, and invocation. If they are caught by a check, they typically result in `-4 throw` (Stack underflow), `-6 throw` (Return stack underflow) or `-9 throw` (Invalid memory address), depending on the platform and which stack underflows and by how much. Note that even if the system uses checking (through the MMU), your program may have to underflow by a significant number of stack items to trigger the reaction (the reason for this is that the MMU, and therefore the checking, works with a page-size granularity). If there is no checking, the symptoms resulting from an underflow are similar to those from an overflow. Unbalanced return stack errors can result in a variety of symptoms, including `-9 throw` (Invalid memory address) and Illegal Instruction (typically `-260 throw`).

unexpected end of the input buffer, resulting in an attempt to use a zero-length string as a name:
`Create` and its descendants perform a `-16 throw` (Attempt to use zero-length string as a name). Words like `'` probably will not find what they search. Note that it is possible to create zero-length names with `nextname` (should it not?).

`>IN` *greater than input buffer:*
The next invocation of a parsing word returns a string with length 0.

`RECURSE` *appears after* `DOES>`*:*
Compiles a recursive call to the defining word, not to the defined word.

argument input source different than current input source for `RESTORE-INPUT`*:*
`-12 THROW`. Note that, once an input file is closed (e.g., because the end of the file was reached), its source-id may be reused. Therefore, restoring an input source specification referencing a closed file may lead to unpredictable results instead of a `-12 THROW`.

In the future, Gforth may be able to restore input source specifications from other than the current input source.

data space containing definitions gets de-allocated:
Deallocation with `allot` is not checked. This typically results in memory access faults or execution of illegal instructions.

data space read/write with incorrect alignment:
Processor-dependent. Typically results in a `-23 throw` (Address alignment exception). Under Linux-Intel on a 486 or later processor with alignment turned on, incorrect alignment results in a `-9 throw` (Invalid memory address). There

are reportedly some processors with alignment restrictions that do not report violations.

data space pointer not properly aligned, `,`*,* `C,`*:*
Like other alignment errors.

less than u+2 stack items (`PICK` *and* `ROLL`*):*
Like other stack underflows.

loop control parameters not available:
Not checked. The counted loop words simply assume that the top of return stack items are loop control parameters and behave accordingly.

most recent definition does not have a name (`IMMEDIATE`*):*
`abort" last word was headerless"`.

name not defined by `VALUE` *used by* `TO`*:*
`-32 throw` (Invalid name argument) (unless name is a local or was defined by `CONSTANT`; in the latter case it just changes the constant).

name not found (`'`*,* `POSTPONE`*,* `[']`*,* `[COMPILE]`*):*
`-13 throw` (Undefined word)

parameters are not of the same type (`DO`*,* `?DO`*,* `WITHIN`*):*
Gforth behaves as if they were of the same type. I.e., you can predict the behaviour by interpreting all parameters as, e.g., signed.

`POSTPONE` *or* `[COMPILE]` *applied to* `TO`*:*
Assume `: X POSTPONE TO ; IMMEDIATE`. `X` performs the compilation semantics of `TO`.

String longer than a counted string returned by `WORD`*:*
Not checked. The string will be ok, but the count will, of course, contain only the least significant bits of the length.

u greater than or equal to the number of bits in a cell (`LSHIFT`*,* `RSHIFT`*):*
Processor-dependent. Typical behaviours are returning 0 and using only the low bits of the shift count.

word not defined via `CREATE`*:*
`>BODY` produces the PFA of the word no matter how it was defined.

`DOES>` changes the execution semantics of the last defined word no matter how it was defined. E.g., `CONSTANT DOES>` is equivalent to `CREATE , DOES>`.

words improperly used outside `<#` *and* `#>`*:*
Not checked. As usual, you can expect memory faults.

8.1.3 Other system documentation

nonstandard words using `PAD`*:*
None.

operator's terminal facilities available:
After processing the OS's command line, Gforth goes into interactive mode, and you can give commands to Gforth interactively. The actual facilities available depend on how you invoke Gforth.

program data space available:
`UNUSED .` gives the remaining dictionary space. The total dictionary space can be specified with the `-m` switch (see Section 2.1 [Invoking Gforth], page 3) when Gforth starts up.

return stack space available:
You can compute the total return stack space in cells with `s" RETURN-STACK-CELLS" environment? drop .`. You can specify it at startup time with the `-r` switch (see Section 2.1 [Invoking Gforth], page 3).

stack space available:
You can compute the total data stack space in cells with `s" STACK-CELLS" environment? drop .`. You can specify it at startup time with the `-d` switch (see Section 2.1 [Invoking Gforth], page 3).

system dictionary space required, in address units:
Type `here forthstart - .` after startup. At the time of this writing, this gives 80080 (bytes) on a 32-bit system.

8.2 The optional Block word set

8.2.1 Implementation Defined Options

the format for display by `LIST`*:*
First the screen number is displayed, then 16 lines of 64 characters, each line preceded by the line number.

the length of a line affected by `\`*:*
64 characters.

8.2.2 Ambiguous conditions

correct block read was not possible:
Typically results in a `throw` of some OS-derived value (between -512 and -2048). If the blocks file was just not long enough, blanks are supplied for the missing portion.

I/O exception in block transfer:
Typically results in a `throw` of some OS-derived value (between -512 and -2048).

invalid block number:
`-35 throw` (Invalid block number)

a program directly alters the contents of `BLK`*:*
The input stream is switched to that other block, at the same position. If the storing to `BLK` happens when interpreting non-block input, the system will get quite confused when the block ends.

no current block buffer for `UPDATE`*:*
`UPDATE` has no effect.

8.2.3 Other system documentation

any restrictions a multiprogramming system places on the use of buffer addresses:
No restrictions (yet).

the number of blocks available for source and data:
depends on your disk space.

8.3 The optional Double Number word set

8.3.1 Ambiguous conditions

d outside of range of n in `D>S`*:*
The least significant cell of *d* is produced.

8.4 The optional Exception word set

8.4.1 Implementation Defined Options

`THROW`*-codes used in the system:*
The codes -256−-511 are used for reporting signals. The mapping from OS signal numbers to throw codes is -256−*signal*. The codes -512−-2047 are used for OS errors (for file and memory allocation operations). The mapping from OS error numbers to throw codes is -512−`errno`. One side effect of this mapping is that undefined OS errors produce a message with a strange number; e.g., `-1000 THROW` results in `Unknown error 488` on my system.

8.5 The optional Facility word set

8.5.1 Implementation Defined Options

encoding of keyboard events (`EKEY`*):*
Keys corresponding to ASCII characters are encoded as ASCII characters. Other keys are encoded with the constants `k-left`, `k-right`, `k-up`, `k-down`, `k-home`, `k-end`, `k1`, `k2`, `k3`, `k4`, `k5`, `k6`, `k7`, `k8`, `k9`, `k10`, `k11`, `k12`.

duration of a system clock tick:
System dependent. With respect to `MS`, the time is specified in microseconds. How well the OS and the hardware implement this, is another question.

repeatability to be expected from the execution of `MS`*:*
System dependent. On Unix, a lot depends on load. If the system is lightly loaded, and the delay is short enough that Gforth does not get swapped out, the performance should be acceptable. Under MS-DOS and other single-tasking systems, it should be good.

8.5.2 Ambiguous conditions

`AT-XY` *can't be performed on user output device:*
: Largely terminal dependent. No range checks are done on the arguments. No errors are reported. You may see some garbage appearing, you may see simply nothing happen.

8.6 The optional File-Access word set

8.6.1 Implementation Defined Options

file access methods used:
: `R/O`, `R/W` and `BIN` work as you would expect. `W/O` translates into the C file opening mode `w` (or `wb`): The file is cleared, if it exists, and created, if it does not (with both `open-file` and `create-file`). Under Unix `create-file` creates a file with 666 permissions modified by your umask.

file exceptions:
: The file words do not raise exceptions (except, perhaps, memory access faults when you pass illegal addresses or file-ids).

file line terminator:
: System-dependent. Gforth uses C's newline character as line terminator. What the actual character code(s) of this are is system-dependent.

file name format:
: System dependent. Gforth just uses the file name format of your OS.

information returned by `FILE-STATUS`*:*
: `FILE-STATUS` returns the most powerful file access mode allowed for the file: Either `R/O`, `W/O` or `R/W`. If the file cannot be accessed, `R/O BIN` is returned. `BIN` is applicable along with the returned mode.

input file state after an exception when including source:
: All files that are left via the exception are closed.

ior values and meaning:
: The *ior*s returned by the file and memory allocation words are intended as throw codes. They typically are in the range -512−-2047 of OS errors. The mapping from OS error numbers to *ior*s is -512−*errno*.

maximum depth of file input nesting:
: limited by the amount of return stack, locals/TIB stack, and the number of open files available. This should not give you troubles.

maximum size of input line:
: `/line`. Currently 255.

methods of mapping block ranges to files:
: By default, blocks are accessed in the file '`blocks.fb`' in the current working directory. The file can be switched with `USE`.

number of string buffers provided by `S"`*:*
: 1

size of string buffer used by `S"`*:*
: `/line`. currently 255.

8.6.2 Ambiguous conditions

attempting to position a file outside its boundaries:
: `REPOSITION-FILE` is performed as usual: Afterwards, `FILE-POSITION` returns the value given to `REPOSITION-FILE`.

attempting to read from file positions not yet written:
: End-of-file, i.e., zero characters are read and no error is reported.

file-id is invalid (`INCLUDE-FILE`*):*
: An appropriate exception may be thrown, but a memory fault or other problem is more probable.

I/O exception reading or closing file-id (`INCLUDE-FILE`, `INCLUDED`*):*
: The *ior* produced by the operation, that discovered the problem, is thrown.

named file cannot be opened (`INCLUDED`*):*
: The *ior* produced by `open-file` is thrown.

requesting an unmapped block number:
: There are no unmapped legal block numbers. On some operating systems, writing a block with a large number may overflow the file system and have an error message as consequence.

using `source-id` *when* `blk` *is non-zero:*
: `source-id` performs its function. Typically it will give the id of the source which loaded the block. (Better ideas?)

8.7 The optional Floating-Point word set

8.7.1 Implementation Defined Options

format and range of floating point numbers:
: System-dependent; the `double` type of C.

results of `REPRESENT` *when float is out of range:*
: System dependent; `REPRESENT` is implemented using the C library function `ecvt()` and inherits its behaviour in this respect.

rounding or truncation of floating-point numbers:
: System dependent; the rounding behaviour is inherited from the hosting C compiler. IEEE-FP-based (i.e., most) systems by default round to nearest, and break ties by rounding to even (i.e., such that the last bit of the mantissa is 0).

size of floating-point stack:
: `s" FLOATING-STACK" environment? drop .` gives the total size of the floating-point stack (in floats). You can specify this on startup with the command-line option `-f` (see Section 2.1 [Invoking Gforth], page 3).

width of floating-point stack:
`1 floats`.

8.7.2 Ambiguous conditions

`df@` *or* `df!` *used with an address that is not double-float aligned:*
System-dependent. Typically results in a `-23 THROW` like other alignment violations.

`f@` *or* `f!` *used with an address that is not float aligned:*
System-dependent. Typically results in a `-23 THROW` like other alignment violations.

floating-point result out of range:
System-dependent. Can result in a `-43 throw` (floating point overflow), `-54 throw` (floating point underflow), `-41 throw` (floating point inexact result), `-55 THROW` (Floating-point unidentified fault), or can produce a special value representing, e.g., Infinity.

`sf@` *or* `sf!` *used with an address that is not single-float aligned:*
System-dependent. Typically results in an alignment fault like other alignment violations.

`base` *is not decimal (*`REPRESENT`*,* `F.`*,* `FE.`*,* `FS.`*):*
The floating-point number is converted into decimal nonetheless.

Both arguments are equal to zero (`FATAN2`*):*
System-dependent. `FATAN2` is implemented using the C library function `atan2()`.

Using `FTAN` *on an argument r1 where cos(r1) is zero:*
System-dependent. Anyway, typically the cos of *r1* will not be zero because of small errors and the tan will be a very large (or very small) but finite number.

d cannot be presented precisely as a float in `D>F`*:*
The result is rounded to the nearest float.

dividing by zero:
Platform-dependent; can produce an Infinity, NaN, `-42 throw` (floating point divide by zero) or `-55 throw` (Floating-point unidentified fault).

exponent too big for conversion (`DF!`*,* `DF@`*,* `SF!`*,* `SF@`*):*
System dependent. On IEEE-FP based systems the number is converted into an infinity.

float<1 (`FACOSH`*):*
Platform-dependent; on IEEE-FP systems typically produces a NaN.

float=<-1 (`FLNP1`*):*
Platform-dependent; on IEEE-FP systems typically produces a NaN (or a negative infinity for *float*=-1).

float=<0 (`FLN`*,* `FLOG`*):*
Platform-dependent; on IEEE-FP systems typically produces a NaN (or a negative infinity for *float*=0).

float<0 (`FASINH`*,* `FSQRT`*):*
: Platform-dependent; for `fsqrt` this typically gives a NaN, for `fasinh` some platforms produce a NaN, others a number (bug in the C library?).

|float|>1 (`FACOS`*,* `FASIN`*,* `FATANH`*):*
: Platform-dependent; IEEE-FP systems typically produce a NaN.

integer part of float cannot be represented by d in `F>D`*:*
: Platform-dependent; typically, some double number is produced and no error is reported.

string larger than pictured numeric output area (`f.`*,* `fe.`*,* `fs.`*):*
: `Precision` characters of the numeric output area are used. If `precision` is too high, these words will smash the data or code close to `here`.

8.8 The optional Locals word set

8.8.1 Implementation Defined Options

maximum number of locals in a definition:
: `s" #locals" environment? drop .`. Currently 15. This is a lower bound, e.g., on a 32-bit machine there can be 41 locals of up to 8 characters. The number of locals in a definition is bounded by the size of locals-buffer, which contains the names of the locals.

8.8.2 Ambiguous conditions

executing a named local in interpretation state:
: Locals have no interpretation semantics. If you try to perform the interpretation semantics, you will get a `-14 throw` somewhere (Interpreting a compile-only word). If you perform the compilation semantics, the locals access will be compiled (irrespective of state).

name not defined by `VALUE` *or* `(LOCAL)` *(*`TO`*):*
: `-32 throw` (Invalid name argument)

8.9 The optional Memory-Allocation word set

8.9.1 Implementation Defined Options

values and meaning of ior:
: The *ior*s returned by the file and memory allocation words are intended as throw codes. They typically are in the range -512−-2047 of OS errors. The mapping from OS error numbers to *ior*s is -512−*errno*.

8.10 The optional Programming-Tools word set

8.10.1 Implementation Defined Options

ending sequence for input following `;CODE` *and* `CODE`*:*
: `END-CODE`

manner of processing input following `;CODE` *and* `CODE`*:*
: The `ASSEMBLER` vocabulary is pushed on the search order stack, and the input is processed by the text interpreter, (starting) in interpret state.

search order capability for `EDITOR` *and* `ASSEMBLER`*:*
: The ANS Forth search order word set.

source and format of display by `SEE`*:*
: The source for `see` is the executable code used by the inner interpreter. The current `see` tries to output Forth source code (and on some platforms, assembly code for primitives) as well as possible.

8.10.2 Ambiguous conditions

deleting the compilation word list (`FORGET`*):*
: Not implemented (yet).

fewer than u+1 items on the control-flow stack (`CS-PICK`*,* `CS-ROLL`*):*
: This typically results in an `abort"` with a descriptive error message (may change into a `-22 throw` (Control structure mismatch) in the future). You may also get a memory access error. If you are unlucky, this ambiguous condition is not caught.

name can't be found (`FORGET`*):*
: Not implemented (yet).

name not defined via `CREATE`*:*
: `;CODE` behaves like `DOES>` in this respect, i.e., it changes the execution semantics of the last defined word no matter how it was defined.

`POSTPONE` *applied to* `[IF]`*:*
: After defining `: X POSTPONE [IF] ; IMMEDIATE`. `X` is equivalent to `[IF]`.

reaching the end of the input source before matching `[ELSE]` *or* `[THEN]`*:*
: Continue in the same state of conditional compilation in the next outer input source. Currently there is no warning to the user about this.

removing a needed definition (`FORGET`*):*
: Not implemented (yet).

8.11 The optional Search-Order word set

8.11.1 Implementation Defined Options

maximum number of word lists in search order:
`s" wordlists" environment? drop .`. Currently 16.

minimum search order:
`root root`.

8.11.2 Ambiguous conditions

changing the compilation word list (during compilation):
The word is entered into the word list that was the compilation word list at the start of the definition. Any changes to the name field (e.g., `immediate`) or the code field (e.g., when executing `DOES>`) are applied to the latest defined word (as reported by `last` or `lastxt`), if possible, irrespective of the compilation word list.

search order empty (`previous`*):*
`abort" Vocstack empty"`.

too many word lists in search order (`also`*):*
`abort" Vocstack full"`.

9 Should I use Gforth extensions?

As you read through the rest of this manual, you will see documentation for *Standard* words, and documentation for some appealing Gforth *extensions*. You might ask yourself the question: *"Should I restrict myself to the standard, or should I use the extensions?"*

The answer depends on the goals you have for the program you are working on:

- Is it just for yourself or do you want to share it with others?
- If you want to share it, do the others all use Gforth?
- If it is just for yourself, do you want to restrict yourself to Gforth?

If restricting the program to Gforth is ok, then there is no reason not to use extensions. It is still a good idea to keep to the standard where it is easy, in case you want to reuse these parts in another program that you want to be portable.

If you want to be able to port the program to other Forth systems, there are the following points to consider:

- Most Forth systems that are being maintained support the ANS Forth standard. So if your program complies with the standard, it will be portable among many systems.
- A number of the Gforth extensions can be implemented in ANS Forth using public-domain files provided in the '`compat/`' directory. These are mentioned in the text in passing. There is no reason not to use these extensions, your program will still be ANS Forth compliant; just include the appropriate compat files with your program.
- The tool '`ans-report.fs`' (see Section 7.1 [ANS Report], page 165) makes it easy to analyse your program and determine what non-Standard words it relies upon. However, it does not check whether you use standard words in a non-standard way.
- Some techniques are not standardized by ANS Forth, and are hard or impossible to implement in a standard way, but can be implemented in most Forth systems easily, and usually in similar ways (e.g., accessing word headers). Forth has a rich historical precedent for programmers taking advantage of implementation-dependent features of their tools (for example, relying on a knowledge of the dictionary structure). Sometimes these techniques are necessary to extract every last bit of performance from the hardware, sometimes they are just a programming shorthand.
- Does using a Gforth extension save more work than the porting this part to other Forth systems (if any) will cost?
- Is the additional functionality worth the reduction in portability and the additional porting problems?

In order to perform these consideratios, you need to know what's standard and what's not. This manual generally states if something is non-standard, but the authoritative source is the standard document (`http://www.taygeta.com/forth/dpans.html`). Appendix A of the Standard (*Rationale*) provides a valuable insight into the thought processes of the technical committee.

Note also that portability between Forth systems is not the only portability issue; there is also the issue of portability between different platforms (processor/OS combinations).

10 Model

This chapter has yet to be written. It will contain information, on which internal structures you can rely.

11 Integrating Gforth into C programs

This is not yet implemented.

Several people like to use Forth as scripting language for applications that are otherwise written in C, C++, or some other language.

The Forth system ATLAST provides facilities for embedding it into applications; unfortunately it has several disadvantages: most importantly, it is not based on ANS Forth, and it is apparently dead (i.e., not developed further and not supported). The facilities provided by Gforth in this area are inspired by ATLAST's facilities, so making the switch should not be hard.

We also tried to design the interface such that it can easily be implemented by other Forth systems, so that we may one day arrive at a standardized interface. Such a standard interface would allow you to replace the Forth system without having to rewrite C code.

You embed the Gforth interpreter by linking with the library `libgforth.a` (give the compiler the option `-lgforth`). All global symbols in this library that belong to the interface, have the prefix `forth_`. (Global symbols that are used internally have the prefix `gforth_`).

You can include the declarations of Forth types and the functions and variables of the interface with `#include <forth.h>`.

Types.

Variables.

Data and FP Stack pointer. Area sizes.

functions.

forth_init(imagefile) forth_evaluate(string) exceptions? forth_goto(address) (or forth_execute(xt)?) forth_continue() (a corountining mechanism)

Adding primitives.

No checking.

Signals?

Accessing the Stacks

12 Emacs and Gforth

Gforth comes with ‘`gforth.el`’, an improved version of ‘`forth.el`’ by Goran Rydqvist (included in the TILE package). The improvements are:

- A better handling of indentation.
- A custom hilighting engine for Forth-code.
- Comment paragraph filling (*M-q*)
- Commenting (*C-x* \) and uncommenting (*C-u C-x* \) of regions
- Removal of debugging tracers (*C-x ~*, see Section 5.23.3 [Debugging], page 153).
- Support of the `info-lookup` feature for looking up the documentation of a word.
- Support for reading and writing blocks files.

To get a basic description of these features, enter Forth mode and type *C-h m*.

In addition, Gforth supports Emacs quite well: The source code locations given in error messages, debugging output (from `~~`) and failed assertion messages are in the right format for Emacs' compilation mode (see section “Running Compilations under Emacs” in *Emacs Manual*) so the source location corresponding to an error or other message is only a few keystrokes away (*C-x `* for the next error, *C-c C-c* for the error under the cursor).

Moreover, for words documented in this manual, you can look up the glossary entry quickly by using *C-h TAB* (`info-lookup-symbol`, see section “Documentation Commands” in *Emacs Manual*). This feature requires Emacs 20.3 or later and does not work for words containing `:`.

12.1 Installing gforth.el

To make the features from ‘`gforth.el`’ available in Emacs, add the following lines to your ‘`.emacs`’ file:

```
(autoload 'forth-mode "gforth.el")
(setq auto-mode-alist (cons '("\\.fs\\'" . forth-mode)
    auto-mode-alist))
(autoload 'forth-block-mode "gforth.el")
(setq auto-mode-alist (cons '("\\.fb\\'" . forth-block-mode)
    auto-mode-alist))
(add-hook 'forth-mode-hook (function (lambda ()
   ;; customize variables here:
   (setq forth-indent-level 4)
   (setq forth-minor-indent-level 2)
   (setq forth-hilight-level 3)
   ;;; ...
)))
```

12.2 Emacs Tags

If you `require` ‘`etags.fs`’, a new ‘`TAGS`’ file will be produced (see section “Tags Tables” in *Emacs Manual*) that contains the definitions of all words defined afterwards. You can then find the source for a word using *M-.*. Note that Emacs can use several tags files at the same time (e.g., one for the Gforth sources and one for your program, see section "Selecting a Tags Table" in *Emacs Manual*). The TAGS file for the preloaded words is '`$(datadir)/gforth/$(VERSION)/TAGS`' (e.g., '`/usr/local/share/gforth/0.2.0/TAGS`'). To get the best behaviour with '`etags.fs`', you should avoid putting definitions both before and after `require` etc., otherwise you will see the same file visited several times by commands like `tags-search`.

12.3 Hilighting

'`gforth.el`' comes with a custom source hilighting engine. When you open a file in `forth-mode`, it will be completely parsed, assigning faces to keywords, comments, strings etc. While you edit the file, modified regions get parsed and updated on-the-fly.

Use the variable 'forth-hilight-level' to change the level of decoration from 0 (no hilighting at all) to 3 (the default). Even if you set the hilighting level to 0, the parser will still work in the background, collecting information about whether regions of text are "compiled" or "interpreted". Those information are required for auto-indentation to work properly. Set 'forth-disable-parser' to non-nil if your computer is too slow to handle parsing. This will have an impact on the smartness of the auto-indentation engine, though.

Sometimes Forth sources define new features that should be hilighted, new control structures, defining-words etc. You can use the variable 'forth-custom-words' to make `forth-mode` hilight additional words and constructs. See the docstring of 'forth-words' for details (in Emacs, type *`C-h v forth-words`*).

'forth-custom-words' is meant to be customized in your '`.emacs`' file. To customize hilighing in a file-specific manner, set 'forth-local-words' in a local-variables section at the end of your source file (see section "Variables" in *Emacs Manual*).

Example:

```
0 [IF]
   Local Variables:
   forth-local-words:
      ((("t:") definition-starter (font-lock-keyword-face . 1)
        "[ \t\n]" t name (font-lock-function-name-face . 3))
       ((";t") definition-ender (font-lock-keyword-face . 1)))
   End:
[THEN]
```

12.4 Auto-Indentation

`forth-mode` automatically tries to indent lines in a smart way, whenever you type ⟨TAB⟩ or break a line with *`C-m`*.

Simple customization can be achieved by setting 'forth-indent-level' and 'forth-minor-indent-level' in your '`.emacs`' file. For historical reasons '`gforth.el`' indents per default by multiples of 4 columns. To use the more traditional 3-column indentation, add the following lines to your '`.emacs`':

```
(add-hook 'forth-mode-hook (function (lambda ()
   ;; customize variables here:
   (setq forth-indent-level 3)
```

```
      (setq forth-minor-indent-level 1)
   )))
```

If you want indentation to recognize non-default words, customize it by setting 'forth-custom-indent-words' in your '`.emacs`'. See the docstring of 'forth-indent-words' for details (in Emacs, type *`C-h v forth-indent-words`*).

To customize indentation in a file-specific manner, set 'forth-local-indent-words' in a local-variables section at the end of your source file (see section "Local Variables in Files" in *Emacs Manual*).

Example:

```
0 [IF]
   Local Variables:
   forth-local-indent-words:
      ((("t:") (0 . 2) (0 . 2))
       ((";t") (-2 . 0) (0 . -2)))
   End:
[THEN]
```

12.5 Blocks Files

`forth-mode` Autodetects blocks files by checking whether the length of the first line exceeds 1023 characters. It then tries to convert the file into normal text format. When you save the file, it will be written to disk as normal stream-source file.

If you want to write blocks files, use `forth-blocks-mode`. It inherits all the features from `forth-mode`, plus some additions:

- Files are written to disk in blocks file format.
- Screen numbers are displayed in the mode line (enumerated beginning with the value of 'forth-block-base')
- Warnings are displayed when lines exceed 64 characters.
- The beginning of the currently edited block is marked with an overlay-arrow.

There are some restrictions you should be aware of. When you open a blocks file that contains tabulator or newline characters, these characters will be translated into spaces when the file is written back to disk. If tabs or newlines are encountered during blocks file reading, an error is output to the echo area. So have a look at the '*Messages*' buffer, when Emacs' bell rings during reading.

Please consult the docstring of `forth-blocks-mode` for more information by typing *`C-h v forth-blocks-mode`*).

13 Image Files

An image file is a file containing an image of the Forth dictionary, i.e., compiled Forth code and data residing in the dictionary. By convention, we use the extension `.fi` for image files.

13.1 Image Licensing Issues

An image created with `gforthmi` (see Section 13.5.1 [gforthmi], page 189) or `savesystem` (see Section 13.3 [Non-Relocatable Image Files], page 188) includes the original image; i.e., according to copyright law it is a derived work of the original image.

Since Gforth is distributed under the GNU GPL, the newly created image falls under the GNU GPL, too. In particular, this means that if you distribute the image, you have to make all of the sources for the image available, including those you wrote. For details see Section D.2 [GNU General Public License (Section 3)], page 213.

If you create an image with `cross` (see Section 13.5.2 [cross.fs], page 190), the image contains only code compiled from the sources you gave it; if none of these sources is under the GPL, the terms discussed above do not apply to the image. However, if your image needs an engine (a gforth binary) that is under the GPL, you should make sure that you distribute both in a way that is at most a *mere aggregation*, if you don't want the terms of the GPL to apply to the image.

13.2 Image File Background

Gforth consists not only of primitives (in the engine), but also of definitions written in Forth. Since the Forth compiler itself belongs to those definitions, it is not possible to start the system with the engine and the Forth source alone. Therefore we provide the Forth code as an image file in nearly executable form. When Gforth starts up, a C routine loads the image file into memory, optionally relocates the addresses, then sets up the memory (stacks etc.) according to information in the image file, and (finally) starts executing Forth code.

The image file variants represent different compromises between the goals of making it easy to generate image files and making them portable.

Win32Forth 3.4 and Mitch Bradley's `cforth` use relocation at run-time. This avoids many of the complications discussed below (image files are data relocatable without further ado), but costs performance (one addition per memory access).

By contrast, the Gforth loader performs relocation at image load time. The loader also has to replace tokens that represent primitive calls with the appropriate code-field addresses (or code addresses in the case of direct threading).

There are three kinds of image files, with different degrees of relocatability: non-relocatable, data-relocatable, and fully relocatable image files.

These image file variants have several restrictions in common; they are caused by the design of the image file loader:

- There is only one segment; in particular, this means, that an image file cannot represent `ALLOCATE`d memory chunks (and pointers to them). The contents of the stacks are not represented, either.

- The only kinds of relocation supported are: adding the same offset to all cells that represent data addresses; and replacing special tokens with code addresses or with pieces of machine code.

  If any complex computations involving addresses are performed, the results cannot be represented in the image file. Several applications that use such computations come to mind:

  - Hashing addresses (or data structures which contain addresses) for table lookup. If you use Gforth's `table`s or `wordlist`s for this purpose, you will have no problem, because the hash tables are recomputed automatically when the system is started. If you use your own hash tables, you will have to do something similar.
  - There's a cute implementation of doubly-linked lists that uses `XOR`ed addresses. You could represent such lists as singly-linked in the image file, and restore the doubly-linked representation on startup.[1]
  - The code addresses of run-time routines like `docol:` cannot be represented in the image file (because their tokens would be replaced by machine code in direct threaded implementations). As a workaround, compute these addresses at run-time with `>code-address` from the executions tokens of appropriate words (see the definitions of `docol:` and friends in '`kernel/getdoers.fs`').
  - On many architectures addresses are represented in machine code in some shifted or mangled form. You cannot put `CODE` words that contain absolute addresses in this form in a relocatable image file. Workarounds are representing the address in some relative form (e.g., relative to the CFA, which is present in some register), or loading the address from a place where it is stored in a non-mangled form.

13.3 Non-Relocatable Image Files

These files are simple memory dumps of the dictionary. They are specific to the executable (i.e., '`gforth`' file) they were created with. What's worse, they are specific to the place on which the dictionary resided when the image was created. Now, there is no guarantee that the dictionary will reside at the same place the next time you start Gforth, so there's no guarantee that a non-relocatable image will work the next time (Gforth will complain instead of crashing, though).

You can create a non-relocatable image file with

`savesystem` "*name*" – gforth "savesystem"

13.4 Data-Relocatable Image Files

These files contain relocatable data addresses, but fixed code addresses (instead of tokens). They are specific to the executable (i.e., '`gforth`' file) they were created with. For direct threading on some architectures (e.g., the i386), data-relocatable images do not work. You get a data-relocatable image, if you use '`gforthmi`' with a Gforth binary that is not doubly indirect threaded (see Section 13.5 [Fully Relocatable Image Files], page 189).

[1] In my opinion, though, you should think thrice before using a doubly-linked list (whatever implementation).

13.5 Fully Relocatable Image Files

These image files have relocatable data addresses, and tokens for code addresses. They can be used with different binaries (e.g., with and without debugging) on the same machine, and even across machines with the same data formats (byte order, cell size, floating point format). However, they are usually specific to the version of Gforth they were created with. The files '`gforth.fi`' and '`kernl*.fi`' are fully relocatable.

There are two ways to create a fully relocatable image file:

13.5.1 '`gforthmi`'

You will usually use '`gforthmi`'. If you want to create an image *file* that contains everything you would load by invoking Gforth with `gforth` *options*, you simply say:

`gforthmi` *file options*

E.g., if you want to create an image '`asm.fi`' that has the file '`asm.fs`' loaded in addition to the usual stuff, you could do it like this:

```
gforthmi asm.fi asm.fs
```

'`gforthmi`' is implemented as a sh script and works like this: It produces two non-relocatable images for different addresses and then compares them. Its output reflects this: first you see the output (if any) of the two Gforth invocations that produce the non-relocatable image files, then you see the output of the comparing program: It displays the offset used for data addresses and the offset used for code addresses; moreover, for each cell that cannot be represented correctly in the image files, it displays a line like this:

```
     78DC         BFFFFA50       BFFFFA40
```

This means that at offset $78dc from `forthstart`, one input image contains $bffffa50, and the other contains $bffffa40. Since these cells cannot be represented correctly in the output image, you should examine these places in the dictionary and verify that these cells are dead (i.e., not read before they are written).

If you insert the option `--application` in front of the image file name, you will get an image that uses the `--appl-image` option instead of the `--image-file` option (see Section 2.1 [Invoking Gforth], page 3). When you execute such an image on Unix (by typing the image name as command), the Gforth engine will pass all options to the image instead of trying to interpret them as engine options.

If you type '`gforthmi`' with no arguments, it prints some usage instructions.

There are a few wrinkles: After processing the passed *options*, the words `savesystem` and `bye` must be visible. A special doubly indirect threaded version of the '`gforth`' executable is used for creating the non-relocatable images; you can pass the exact filename of this executable through the environment variable `GFORTHD` (default: '`gforth-ditc`'); if you pass a version that is not doubly indirect threaded, you will not get a fully relocatable image, but a data-relocatable image (because there is no code address offset). The normal '`gforth`' executable is used for creating the relocatable image; you can pass the exact filename of this executable through the environment variable `GFORTH`.

13.5.2 'cross.fs'

You can also use `cross`, a batch compiler that accepts a Forth-like programming language (see Chapter 15 [Cross Compiler], page 201).

`cross` allows you to create image files for machines with different data sizes and data formats than the one used for generating the image file. You can also use it to create an application image that does not contain a Forth compiler. These features are bought with restrictions and inconveniences in programming. E.g., addresses have to be stored in memory with special words (`A!`, `A,`, etc.) in order to make the code relocatable.

13.6 Stack and Dictionary Sizes

If you invoke Gforth with a command line flag for the size (see Section 2.1 [Invoking Gforth], page 3), the size you specify is stored in the dictionary. If you save the dictionary with `savesystem` or create an image with '`gforthmi`', this size will become the default for the resulting image file. E.g., the following will create a fully relocatable version of '`gforth.fi`' with a 1MB dictionary:

```
gforthmi gforth.fi -m 1M
```

In other words, if you want to set the default size for the dictionary and the stacks of an image, just invoke '`gforthmi`' with the appropriate options when creating the image.

Note: For cache-friendly behaviour (i.e., good performance), you should make the sizes of the stacks modulo, say, 2K, somewhat different. E.g., the default stack sizes are: data: 16k (mod 2k=0); fp: 15.5k (mod 2k=1.5k); return: 15k(mod 2k=1k); locals: 14.5k (mod 2k=0.5k).

13.7 Running Image Files

You can invoke Gforth with an image file *image* instead of the default '`gforth.fi`' with the `-i` flag (see Section 2.1 [Invoking Gforth], page 3):

```
gforth -i image
```

If your operating system supports starting scripts with a line of the form `#! ...`, you just have to type the image file name to start Gforth with this image file (note that the file extension `.fi` is just a convention). I.e., to run Gforth with the image file *image*, you can just type *image* instead of `gforth -i` *image*. This works because every `.fi` file starts with a line of this format:

```
#! /usr/local/bin/gforth-0.4.0 -i
```

The file and pathname for the Gforth engine specified on this line is the specific Gforth executable that it was built against; i.e. the value of the environment variable `GFORTH` at the time that '`gforthmi`' was executed.

You can make use of the same shell capability to make a Forth source file into an executable. For example, if you place this text in a file:

```
#! /usr/local/bin/gforth

." Hello, world" CR
bye
```

and then make the file executable (chmod +x in Unix), you can run it directly from the command line. The sequence `#!` is used in two ways; firstly, it is recognised as a “magic sequence” by the operating system[2] secondly it is treated as a comment character by Gforth. Because of the second usage, a space is required between `#!` and the path to the executable (moreover, some Unixes require the sequence `#! /`).

The disadvantage of this latter technique, compared with using ‘`gforthmi`’, is that it is slightly slower; the Forth source code is compiled on-the-fly, each time the program is invoked.

`#!` – gforth “hash-bang”

An alias for `\`

13.8 Modifying the Startup Sequence

You can add your own initialization to the startup sequence through the deferred word `'cold`. `'cold` is invoked just before the image-specific command line processing (i.e., loading files and evaluating (`-e`) strings) starts.

A sequence for adding your initialization usually looks like this:

```
:noname
    Defers 'cold \ do other initialization stuff (e.g., rehashing wordlists)
    ... \ your stuff
; IS 'cold
```

You can make a turnkey image by letting `'cold` execute a word (your turnkey application) that never returns; instead, it exits Gforth via `bye` or `throw`.

You can access the (image-specific) command-line arguments through the variables `argc` and `argv`. `arg` provides convenient access to `argv`.

If `'cold` exits normally, Gforth processes the command-line arguments as files to be loaded and strings to be evaluated. Therefore, `'cold` should remove the arguments it has used in this case.

`'cold` – gforth “tick-cold”

`argc` – *addr* gforth “argc”

`Variable` – the number of command-line arguments (including the command name).

`argv` – *addr* gforth “argv”

`Variable` – a pointer to a vector of pointers to the command-line arguments (including the command-name). Each argument is represented as a C-style string.

`arg` *n* – *addr count* gforth “arg”

Return the string for the nth command-line argument.

[2] The Unix kernel actually recognises two types of files: executable files and files of data, where the data is processed by an interpreter that is specified on the “interpreter line” – the first line of the file, starting with the sequence #!. There may be a small limit (e.g., 32) on the number of characters that may be specified on the interpreter line.

14 Engine

Reading this chapter is not necessary for programming with Gforth. It may be helpful for finding your way in the Gforth sources.

The ideas in this section have also been published in the following papers: Bernd Paysan, *ANS fig/GNU/??? Forth* (in German), Forth-Tagung '93; M. Anton Ertl, *A Portable Forth Engine* (`http://www.complang.tuwien.ac.at/papers/ertl93.ps.Z`), EuroForth '93; M. Anton Ertl, *Threaded code variations and optimizations (extended version)* (`http://www.complang.tuwien.ac.at/papers/ertl02.ps.gz`), Forth-Tagung '02.

14.1 Portability

An important goal of the Gforth Project is availability across a wide range of personal machines. fig-Forth, and, to a lesser extent, F83, achieved this goal by manually coding the engine in assembly language for several then-popular processors. This approach is very labor-intensive and the results are short-lived due to progress in computer architecture.

Others have avoided this problem by coding in C, e.g., Mitch Bradley (cforth), Mikael Patel (TILE) and Dirk Zoller (pfe). This approach is particularly popular for UNIX-based Forths due to the large variety of architectures of UNIX machines. Unfortunately an implementation in C does not mix well with the goals of efficiency and with using traditional techniques: Indirect or direct threading cannot be expressed in C, and switch threading, the fastest technique available in C, is significantly slower. Another problem with C is that it is very cumbersome to express double integer arithmetic.

Fortunately, there is a portable language that does not have these limitations: GNU C, the version of C processed by the GNU C compiler (see section "Extensions to the C Language Family" in *GNU C Manual*). Its labels as values feature (see section "Labels as Values" in *GNU C Manual*) makes direct and indirect threading possible, its `long long` type (see section "Double-Word Integers" in *GNU C Manual*) corresponds to Forth's double numbers on many systems. GNU C is freely available on all important (and many unimportant) UNIX machines, VMS, 80386s running MS-DOS, the Amiga, and the Atari ST, so a Forth written in GNU C can run on all these machines.

Writing in a portable language has the reputation of producing code that is slower than assembly. For our Forth engine we repeatedly looked at the code produced by the compiler and eliminated most compiler-induced inefficiencies by appropriate changes in the source code.

However, register allocation cannot be portably influenced by the programmer, leading to some inefficiencies on register-starved machines. We use explicit register declarations (see section "Variables in Specified Registers" in *GNU C Manual*) to improve the speed on some machines. They are turned on by using the configuration flag `--enable-force-reg` (`gcc` switch `-DFORCE_REG`). Unfortunately, this feature not only depends on the machine, but also on the compiler version: On some machines some compiler versions produce incorrect code when certain explicit register declarations are used. So by default `-DFORCE_REG` is not used.

14.2 Threading

GNU C's labels as values extension (available since `gcc-2.0`, see section "Labels as Values" in *GNU C Manual*) makes it possible to take the address of *label* by writing `&&`*label*. This address can then be used in a statement like `goto *`*address*. I.e., `goto *&&x` is the same as `goto x`.

With this feature an indirect threaded `NEXT` looks like:

```
cfa = *ip++;
ca = *cfa;
goto *ca;
```

For those unfamiliar with the names: `ip` is the Forth instruction pointer; the `cfa` (code-field address) corresponds to ANS Forths execution token and points to the code field of the next word to be executed; The `ca` (code address) fetched from there points to some executable code, e.g., a primitive or the colon definition handler `docol`.

Direct threading is even simpler:

```
ca = *ip++;
goto *ca;
```

Of course we have packaged the whole thing neatly in macros called `NEXT` and `NEXT1` (the part of `NEXT` after fetching the cfa).

14.2.1 Scheduling

There is a little complication: Pipelined and superscalar processors, i.e., RISC and some modern CISC machines can process independent instructions while waiting for the results of an instruction. The compiler usually reorders (schedules) the instructions in a way that achieves good usage of these delay slots. However, on our first tries the compiler did not do well on scheduling primitives. E.g., for `+` implemented as

```
n=sp[0]+sp[1];
sp++;
sp[0]=n;
NEXT;
```

the `NEXT` comes strictly after the other code, i.e., there is nearly no scheduling. After a little thought the problem becomes clear: The compiler cannot know that `sp` and `ip` point to different addresses (and the version of `gcc` we used would not know it even if it was possible), so it could not move the load of the cfa above the store to the TOS. Indeed the pointers could be the same, if code on or very near the top of stack were executed. In the interest of speed we chose to forbid this probably unused "feature" and helped the compiler in scheduling: `NEXT` is divided into several parts: `NEXT_P0`, `NEXT_P1` and `NEXT_P2`). `+` now looks like:

```
NEXT_P0;
n=sp[0]+sp[1];
sp++;
NEXT_P1;
sp[0]=n;
NEXT_P2;
```

There are various schemes that distribute the different operations of NEXT between these parts in several ways; in general, different schemes perform best on different processors. We use a scheme for most architectures that performs well for most processors of this architecture; in the future we may switch to benchmarking and chosing the scheme on installation time.

14.2.2 Direct or Indirect Threaded?

Threaded forth code consists of references to primitives (simple machine code routines like `+`) and to non-primitives (e.g., colon definitions, variables, constants); for a specific class of non-primitives (e.g., variables) there is one code routine (e.g., `dovar`), but each variable needs a separate reference to its data.

Traditionally Forth has been implemented as indirect threaded code, because this allows to use only one cell to reference a non-primitive (basically you point to the data, and find the code address there).

However, threaded code in Gforth (since 0.6.0) uses two cells for non-primitives, one for the code address, and one for the data address; the data pointer is an immediate argument for the virtual machine instruction represented by the code address. We call this *primitive-centric* threaded code, because all code addresses point to simple primitives. E.g., for a variable, the code address is for `lit` (also used for integer literals like `99`).

Primitive-centric threaded code allows us to use (faster) direct threading as dispatch method, completely portably (direct threaded code in Gforth before 0.6.0 required architecture-specific code). It also eliminates the performance problems related to I-cache consistency that 386 implementations have with direct threaded code, and allows additional optimizations.

There is a catch, however: the *xt* parameter of `execute` can occupy only one cell, so how do we pass non-primitives with their code *and* data addresses to them? Our answer is to use indirect threaded dispatch for `execute` and other words that use a single-cell xt. So, normal threaded code in colon definitions uses direct threading, and `execute` and similar words, which dispatch to xts on the data stack, use indirect threaded code. We call this *hybrid direct/indirect* threaded code.

The engines `gforth` and `gforth-fast` use hybrid direct/indirect threaded code. This means that with these engines you cannot use `,` to compile an xt. Instead, you have to use `compile,`.

If you want to compile xts with `,`, use `gforth-itc`. This engine uses plain old indirect threaded code. It still compiles in a primitive-centric style, so you cannot use `compile,` instead of `,` (e.g., for producing tables of xts with `] word1 word2 ... [`). If you want to do that, you have to use `gforth-itc` and execute `' , is compile,`. Your program can check if it is running on a hybrid direct/indirect threaded engine or a pure indirect threaded engine with `threading-method` (see Section 5.25 [Threading Words], page 161).

14.2.3 Dynamic Superinstructions

The engines `gforth` and `gforth-fast` use another optimization: Dynamic superinstructions with replication. As an example, consider the following colon definition:

```
: squared ( n1 -- n2 )
  dup * ;
```

Gforth compiles this into the threaded code sequence

```
dup
*
;s
```

In normal direct threaded code there is a code address occupying one cell for each of these primitives. Each code address points to a machine code routine, and the interpreter jumps to this machine code in order to execute the primitive. The routines for these three primitives are (in `gforth-fast` on the 386):

```
Code dup
( $804B950 )  add     esi , # -4  \ $83 $C6 $FC
( $804B953 )  add     ebx , # 4  \ $83 $C3 $4
( $804B956 )  mov     dword ptr 4 [esi] , ecx  \ $89 $4E $4
( $804B959 )  jmp     dword ptr FC [ebx]  \ $FF $63 $FC
end-code
Code *
( $804ACC4 )  mov     eax , dword ptr 4 [esi]  \ $8B $46 $4
( $804ACC7 )  add     esi , # 4  \ $83 $C6 $4
( $804ACCA )  add     ebx , # 4  \ $83 $C3 $4
( $804ACCD )  imul    ecx , eax  \ $F $AF $C8
( $804ACD0 )  jmp     dword ptr FC [ebx]  \ $FF $63 $FC
end-code
Code ;s
( $804A693 )  mov     eax , dword ptr [edi]  \ $8B $7
( $804A695 )  add     edi , # 4  \ $83 $C7 $4
( $804A698 )  lea     ebx , dword ptr 4 [eax]  \ $8D $58 $4
( $804A69B )  jmp     dword ptr FC [ebx]  \ $FF $63 $FC
end-code
```

With dynamic superinstructions and replication the compiler does not just lay down the threaded code, but also copies the machine code fragments, usually without the jump at the end.

```
( $4057D27D )  add     esi , # -4  \ $83 $C6 $FC
( $4057D280 )  add     ebx , # 4  \ $83 $C3 $4
( $4057D283 )  mov     dword ptr 4 [esi] , ecx  \ $89 $4E $4
( $4057D286 )  mov     eax , dword ptr 4 [esi]  \ $8B $46 $4
( $4057D289 )  add     esi , # 4  \ $83 $C6 $4
( $4057D28C )  add     ebx , # 4  \ $83 $C3 $4
( $4057D28F )  imul    ecx , eax  \ $F $AF $C8
( $4057D292 )  mov     eax , dword ptr [edi]  \ $8B $7
( $4057D294 )  add     edi , # 4  \ $83 $C7 $4
( $4057D297 )  lea     ebx , dword ptr 4 [eax]  \ $8D $58 $4
( $4057D29A )  jmp     dword ptr FC [ebx]  \ $FF $63 $FC
```

Only when a threaded-code control-flow change happens (e.g., in `;s`), the jump is appended. This optimization eliminates many of these jumps and makes the rest much more predictable. The speedup depends on the processor and the application; on the Athlon and Pentium III this optimization typically produces a speedup by a factor of 2.

The code addresses in the direct-threaded code are set to point to the appropriate points in the copied machine code, in this example like this:

```
primitive  code address
   dup       $4057D27D
   *         $4057D286
   ;s        $4057D292
```

Thus there can be threaded-code jumps to any place in this piece of code. This also simplifies decompilation quite a bit.

You can disable this optimization with '`--no-dynamic`'. You can use the copying without eliminating the jumps (i.e., dynamic replication, but without superinstructions) with '`--no-super`'; this gives the branch prediction benefit alone; the effect on performance depends on the CPU; on the Athlon and Pentium III the speedup is a little less than for dynamic superinstructions with replication.

One use of these options is if you want to patch the threaded code. With superinstructions, many of the dispatch jumps are eliminated, so patching often has no effect. These options preserve all the dispatch jumps.

On some machines dynamic superinstructions are disabled by default, because it is unsafe on these machines. However, if you feel adventurous, you can enable it with '`--dynamic`'.

14.2.4 DOES>

One of the most complex parts of a Forth engine is `dodoes`, i.e., the chunk of code executed by every word defined by a `CREATE`...`DOES>` pair; actually with primitive-centric code, this is only needed if the xt of the word is `execute`d. The main problem here is: How to find the Forth code to be executed, i.e. the code after the `DOES>` (the `DOES>`-code)? There are two solutions:

In fig-Forth the code field points directly to the `dodoes` and the `DOES>`-code address is stored in the cell after the code address (i.e. at *CFA* `cell+`). It may seem that this solution is illegal in the Forth-79 and all later standards, because in fig-Forth this address lies in the body (which is illegal in these standards). However, by making the code field larger for all words this solution becomes legal again. We use this approach. Leaving a cell unused in most words is a bit wasteful, but on the machines we are targeting this is hardly a problem.

14.3 Primitives

14.3.1 Automatic Generation

Since the primitives are implemented in a portable language, there is no longer any need to minimize the number of primitives. On the contrary, having many primitives has an advantage: speed. In order to reduce the number of errors in primitives and to make programming them easier, we provide a tool, the primitive generator ('`prims2x.fs`' aka Vmgen, see section "Introduction" in *Vmgen*), that automatically generates most (and sometimes all) of the C code for a primitive from the stack effect notation. The source for a primitive has the following form:

Forth-name (*stack-effect*) *category* [*pronounc.*]
[""*glossary entry*""]
C code
[:
Forth code]

The items in brackets are optional. The category and glossary fields are there for generating the documentation, the Forth code is there for manual implementations on machines without GNU C. E.g., the source for the primitive `+` is:

```
+    ( n1 n2 -- n )   core    plus
n = n1+n2;
```

This looks like a specification, but in fact `n = n1+n2` is C code. Our primitive generation tool extracts a lot of information from the stack effect notations[1]: The number of items popped from and pushed on the stack, their type, and by what name they are referred to in the C code. It then generates a C code prelude and postlude for each primitive. The final C code for `+` looks like this:

```
I_plus: /* + ( n1 n2 -- n ) */  /* label, stack effect */
/*  */                          /* documentation */
NAME("+")                       /* debugging output (with -DDEBUG) */
{
DEF_CA                          /* definition of variable ca (indirect threading) */
Cell n1;                        /* definitions of variables */
Cell n2;
Cell n;
NEXT_P0;                        /* NEXT part 0 */
n1 = (Cell) sp[1];              /* input */
n2 = (Cell) TOS;
sp += 1;                        /* stack adjustment */
{
n = n1+n2;                      /* C code taken from the source */
}
NEXT_P1;                        /* NEXT part 1 */
TOS = (Cell)n;                  /* output */
NEXT_P2;                        /* NEXT part 2 */
}
```

This looks long and inefficient, but the GNU C compiler optimizes quite well and produces optimal code for `+` on, e.g., the R3000 and the HP RISC machines: Defining the `n`s does not produce any code, and using them as intermediate storage also adds no cost.

There are also other optimizations that are not illustrated by this example: assignments between simple variables are usually for free (copy propagation). If one of the stack items is not used by the primitive (e.g. in `drop`), the compiler eliminates the load from the stack (dead code elimination). On the other hand, there are some things that the compiler does not do, therefore they are performed by '`prims2x.fs`': The compiler does not optimize code away that stores a stack item to the place where it just came from (e.g., `over`).

[1] We use a one-stack notation, even though we have separate data and floating-point stacks; The separate notation can be generated easily from the unified notation.

While programming a primitive is usually easy, there are a few cases where the programmer has to take the actions of the generator into account, most notably `?dup`, but also words that do not (always) fall through to `NEXT`.

For more information

14.3.2 TOS Optimization

An important optimization for stack machine emulators, e.g., Forth engines, is keeping one or more of the top stack items in registers. If a word has the stack effect *in1...inx* `--` *out1...outy*, keeping the top *n* items in registers

- is better than keeping *n-1* items, if *x*>=*n* and *y*>=*n*, due to fewer loads from and stores to the stack.
- is slower than keeping *n-1* items, if *x*<>*y* and *x*<*n* and *y*<*n*, due to additional moves between registers.

In particular, keeping one item in a register is never a disadvantage, if there are enough registers. Keeping two items in registers is a disadvantage for frequent words like `?branch`, constants, variables, literals and `i`. Therefore our generator only produces code that keeps zero or one items in registers. The generated C code covers both cases; the selection between these alternatives is made at C-compile time using the switch `-DUSE_TOS`. `TOS` in the C code for `+` is just a simple variable name in the one-item case, otherwise it is a macro that expands into `sp[0]`. Note that the GNU C compiler tries to keep simple variables like `TOS` in registers, and it usually succeeds, if there are enough registers.

The primitive generator performs the TOS optimization for the floating-point stack, too (`-DUSE_FTOS`). For floating-point operations the benefit of this optimization is even larger: floating-point operations take quite long on most processors, but can be performed in parallel with other operations as long as their results are not used. If the FP-TOS is kept in a register, this works. If it is kept on the stack, i.e., in memory, the store into memory has to wait for the result of the floating-point operation, lengthening the execution time of the primitive considerably.

The TOS optimization makes the automatic generation of primitives a bit more complicated. Just replacing all occurrences of `sp[0]` by `TOS` is not sufficient. There are some special cases to consider:

- In the case of `dup ( w -- w w )` the generator must not eliminate the store to the original location of the item on the stack, if the TOS optimization is turned on.
- Primitives with stack effects of the form `--` *out1...outy* must store the TOS to the stack at the start. Likewise, primitives with the stack effect *in1...inx* `--` must load the TOS from the stack at the end. But for the null stack effect `--` no stores or loads should be generated.

14.3.3 Produced code

To see what assembly code is produced for the primitives on your machine with your compiler and your flag settings, type `make engine.s` and look at the resulting file '`engine.s`'. Alternatively, you can also disassemble the code of primitives with `see` on some architectures.

14.4 Performance

On RISCs the Gforth engine is very close to optimal; i.e., it is usually impossible to write a significantly faster threaded-code engine.

On register-starved machines like the 386 architecture processors improvements are possible, because `gcc` does not utilize the registers as well as a human, even with explicit register declarations; e.g., Bernd Beuster wrote a Forth system fragment in assembly language and hand-tuned it for the 486; this system is 1.19 times faster on the Sieve benchmark on a 486DX2/66 than Gforth compiled with `gcc-2.6.3` with `-DFORCE_REG`. The situation has improved with gcc-2.95 and gforth-0.4.9; now the most important virtual machine registers fit in real registers (and we can even afford to use the TOS optimization), resulting in a speedup of 1.14 on the sieve over the earlier results. And dynamic superinstructions provide another speedup (but only around a factor 1.2 on the 486).

The potential advantage of assembly language implementations is not necessarily realized in complete Forth systems: We compared Gforth-0.5.9 (direct threaded, compiled with `gcc-2.95.1` and `-DFORCE_REG`) with Win32Forth 1.2093 (newer versions are reportedly much faster), LMI's NT Forth (Beta, May 1994) and Eforth (with and without peephole (aka pinhole) optimization of the threaded code); all these systems were written in assembly language. We also compared Gforth with three systems written in C: PFE-0.9.14 (compiled with `gcc-2.6.3` with the default configuration for Linux: `-O2 -fomit-frame-pointer -DUSE_REGS -DUNROLL_NEXT`), ThisForth Beta (compiled with `gcc-2.6.3 -O3 -fomit-frame-pointer`; ThisForth employs peephole optimization of the threaded code) and TILE (compiled with `make opt`). We benchmarked Gforth, PFE, ThisForth and TILE on a 486DX2/66 under Linux. Kenneth O'Heskin kindly provided the results for Win32Forth and NT Forth on a 486DX2/66 with similar memory performance under Windows NT. Marcel Hendrix ported Eforth to Linux, then extended it to run the benchmarks, added the peephole optimizer, ran the benchmarks and reported the results.

We used four small benchmarks: the ubiquitous Sieve; bubble-sorting and matrix multiplication come from the Stanford integer benchmarks and have been translated into Forth by Martin Fraeman; we used the versions included in the TILE Forth package, but with bigger data set sizes; and a recursive Fibonacci number computation for benchmarking calling performance. The following table shows the time taken for the benchmarks scaled by the time taken by Gforth (in other words, it shows the speedup factor that Gforth achieved over the other systems).

relative time	Gforth	Win32-Forth	NT Forth	eforth	eforth +opt	PFE	This-Forth	TILE
sieve	1.00	2.16	1.78	2.16	1.32	2.46	4.96	13.37
bubble	1.00	1.93	2.07	2.18	1.29	2.21		5.70
matmul	1.00	1.92	1.76	1.90	0.96	2.06		5.32
fib	1.00	2.32	2.03	1.86	1.31	2.64	4.55	6.54

You may be quite surprised by the good performance of Gforth when compared with systems written in assembly language. One important reason for the disappointing performance of these other systems is probably that they are not written optimally for the 486 (e.g., they use the `lods` instruction). In addition, Win32Forth uses a comfortable, but costly method for relocating the Forth image: like `cforth`, it computes the actual addresses at run time, resulting in two address computations per `NEXT` (see Section 13.2 [Image File Background], page 187).

The speedup of Gforth over PFE, ThisForth and TILE can be easily explained with the self-imposed restriction of the latter systems to standard C, which makes efficient threading impossible (however, the measured implementation of PFE uses a GNU C extension: see section "Defining Global Register Variables" in *GNU C Manual*). Moreover, current C compilers have a hard time optimizing other aspects of the ThisForth and the TILE source.

The performance of Gforth on 386 architecture processors varies widely with the version of `gcc` used. E.g., `gcc-2.5.8` failed to allocate any of the virtual machine registers into real machine registers by itself and would not work correctly with explicit register declarations, giving a significantly slower engine (on a 486DX2/66 running the Sieve) than the one measured above.

Note that there have been several releases of Win32Forth since the release presented here, so the results presented above may have little predictive value for the performance of Win32Forth today (results for the current release on an i486DX2/66 are welcome).

In *Translating Forth to Efficient C* (`http://www.complang.tuwien.ac.at/papers/ertl&maierhofer95.ps.gz`) by M. Anton Ertl and Martin Maierhofer (presented at EuroForth '95), an indirect threaded version of Gforth is compared with Win32Forth, NT Forth, PFE, ThisForth, and several native code systems; that version of Gforth is slower on a 486 than the version used here. You can find a newer version of these measurements at `http://www.complang.tuwien.ac.at/forth/performance.html`. You can find numbers for Gforth on various machines in '`Benchres`'.

15 Cross Compiler

The cross compiler is used to bootstrap a Forth kernel. Since Gforth is mostly written in Forth, including crucial parts like the outer interpreter and compiler, it needs compiled Forth code to get started. The cross compiler allows to create new images for other architectures, even running under another Forth system.

15.1 Using the Cross Compiler

The cross compiler uses a language that resembles Forth, but isn't. The main difference is that you can execute Forth code after definition, while you usually can't execute the code compiled by cross, because the code you are compiling is typically for a different computer than the one you are compiling on.

The Makefile is already set up to allow you to create kernels for new architectures with a simple make command. The generic kernels using the GCC compiled virtual machine are created in the normal build process with `make`. To create a embedded Gforth executable for e.g. the 8086 processor (running on a DOS machine), type

```
make kernl-8086.fi
```

This will use the machine description from the 'arch/8086' directory to create a new kernel. A machine file may look like that:

```
\ Parameter for target systems                         06oct92py

    4 Constant cell             \ cell size in bytes
    2 Constant cell<<           \ cell shift to bytes
    5 Constant cell>bit         \ cell shift to bits
    8 Constant bits/char        \ bits per character
    8 Constant bits/byte        \ bits per byte [default: 8]
    8 Constant float            \ bytes per float
    8 Constant /maxalign        \ maximum alignment in bytes
false Constant bigendian        \ byte order
( true=big, false=little )

include machpc.fs               \ feature list
```

This part is obligatory for the cross compiler itself, the feature list is used by the kernel to conditionally compile some features in and out, depending on whether the target supports these features.

There are some optional features, if you define your own primitives, have an assembler, or need special, nonstandard preparation to make the boot process work. `asm-include` includes an assembler, `prims-include` includes primitives, and `>boot` prepares for booting.

```
: asm-include    ." Include assembler" cr
  s" arch/8086/asm.fs" included ;

: prims-include  ." Include primitives" cr
  s" arch/8086/prim.fs" included ;

: >boot          ." Prepare booting" cr
```

```
s" ' boot >body into-forth 1+ !" evaluate ;
```

These words are used as sort of macro during the cross compilation in the file '`kernel/main.fs`'. Instead of using these macros, it would be possible — but more complicated — to write a new kernel project file, too.

'`kernel/main.fs`' expects the machine description file name on the stack; the cross compiler itself ('`cross.fs`') assumes that either `mach-file` leaves a counted string on the stack, or `machine-file` leaves an address, count pair of the filename on the stack.

The feature list is typically controlled using `SetValue`, generic files that are used by several projects can use `DefaultValue` instead. Both functions work like `Value`, when the value isn't defined, but `SetValue` works like `to` if the value is defined, and `DefaultValue` doesn't set anything, if the value is defined.

```
\ generic mach file for pc gforth                       03sep97jaw

true DefaultValue NIL  \ relocating

>ENVIRON

true DefaultValue file          \ controls the presence of the
                                \ file access wordset
true DefaultValue OS            \ flag to indicate a operating system

true DefaultValue prims         \ true: primitives are c-code

true DefaultValue floating      \ floating point wordset is present

true DefaultValue glocals       \ gforth locals are present
                                \ will be loaded
true DefaultValue dcomps        \ double number comparisons

true DefaultValue hash          \ hashing primitives are loaded/present

true DefaultValue xconds        \ used together with glocals,
                                \ special conditionals supporting gforths'
                                \ local variables
true DefaultValue header        \ save a header information

true DefaultValue backtrace     \ enables backtrace code

false DefaultValue ec
false DefaultValue crlf

cell 2 = [IF] &32 [ELSE] &256 [THEN] KB DefaultValue kernel-size

&16 KB          DefaultValue stack-size
&15 KB &512 +   DefaultValue fstack-size
&15 KB          DefaultValue rstack-size
&14 KB &512 +   DefaultValue lstack-size
```

15.2 How the Cross Compiler Works

Appendix A Bugs

Known bugs are described in the file 'BUGS' in the Gforth distribution.

If you find a bug, please submit a bug report through `https://savannah.gnu.org/bugs/?func=addbug&group=gforth`.

- A program (or a sequence of keyboard commands) that reproduces the bug.
- A description of what you think constitutes the buggy behaviour.
- The Gforth version used (it is announced at the start of an interactive Gforth session).
- The machine and operating system (on Unix systems `uname -a` will report this information).
- The installation options (you can find the configure options at the start of '`config.status`') and configuration (`configure` output or '`config.cache`').
- A complete list of changes (if any) you (or your installer) have made to the Gforth sources.

For a thorough guide on reporting bugs read section "How to Report Bugs" in *GNU C Manual*.

Appendix B Authors and Ancestors of Gforth

B.1 Authors and Contributors

The Gforth project was started in mid-1992 by Bernd Paysan and Anton Ertl. The third major author was Jens Wilke. Neal Crook contributed a lot to the manual. Assemblers and disassemblers were contributed by Andrew McKewan, Christian Pirker, and Bernd Thallner. Lennart Benschop (who was one of Gforth's first users, in mid-1993) and Stuart Ramsden inspired us with their continuous feedback. Lennart Benshop contributed '`glosgen.fs`', while Stuart Ramsden has been working on automatic support for calling C libraries. Helpful comments also came from Paul Kleinrubatscher, Christian Pirker, Dirk Zoller, Marcel Hendrix, John Wavrik, Barrie Stott, Marc de Groot, Jorge Acerada, Bruce Hoyt, Robert Epprecht, Dennis Ruffer and David N. Williams. Since the release of Gforth-0.2.1 there were also helpful comments from many others; thank you all, sorry for not listing you here (but digging through my mailbox to extract your names is on my to-do list).

Gforth also owes a lot to the authors of the tools we used (GCC, CVS, and autoconf, among others), and to the creators of the Internet: Gforth was developed across the Internet, and its authors did not meet physically for the first 4 years of development.

B.2 Pedigree

Gforth descends from bigFORTH (1993) and fig-Forth. Of course, a significant part of the design of Gforth was prescribed by ANS Forth.

Bernd Paysan wrote bigFORTH, a descendent from TurboForth, an unreleased 32 bit native code version of VolksForth for the Atari ST, written mostly by Dietrich Weineck.

VolksForth was written by Klaus Schleisiek, Bernd Pennemann, Georg Rehfeld and Dietrich Weineck for the C64 (called UltraForth there) in the mid-80s and ported to the Atari ST in 1986. It descends from F83.

Henry Laxen and Mike Perry wrote F83 as a model implementation of the Forth-83 standard. !! Pedigree? When?

A team led by Bill Ragsdale implemented fig-Forth on many processors in 1979. Robert Selzer and Bill Ragsdale developed the original implementation of fig-Forth for the 6502 based on microForth.

The principal architect of microForth was Dean Sanderson. microForth was FORTH, Inc.'s first off-the-shelf product. It was developed in 1976 for the 1802, and subsequently implemented on the 8080, the 6800 and the Z80.

All earlier Forth systems were custom-made, usually by Charles Moore, who discovered (as he puts it) Forth during the late 60s. The first full Forth existed in 1971.

A part of the information in this section comes from *The Evolution of Forth* (`http://www.forth.com/Content/History/History1.htm`) by Elizabeth D. Rather, Donald R. Colburn and Charles H. Moore, presented at the HOPL-II conference and preprinted in SIGPLAN Notices 28(3), 1993. You can find more historical and genealogical information about Forth there.

Appendix C Other Forth-related information

There is an active news group (comp.lang.forth) discussing Forth (including Gforth) and Forth-related issues. Its FAQs (`http://www.complang.tuwien.ac.at/forth/faq/faq-general-2.html`) (frequently asked questions and their answers) contains a lot of information on Forth. You should read it before posting to comp.lang.forth.

The ANS Forth standard is most usable in its HTML form (`http://www.taygeta.com/forth/dpans.html`).

Appendix D Licenses

D.1 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 the terms 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 "Acknowledgments" or "Dedications", preserve the section's title, and preserve in the section all the substance and tone of each of the contributor acknowledgments 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 "Acknowledgments", 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.

D.1.1 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.

D.2 GNU GENERAL PUBLIC LICENSE

Version 2, June 1991

Preamble

The licenses for most software are designed to take away your freedom to share and change it. By contrast, the GNU General Public License is intended to guarantee your freedom to share and change free software—to make sure the software is free for all its users. This General Public License applies to most of the Free Software Foundation's software and to any other program whose authors commit to using it. (Some other Free Software Foundation software is covered by the GNU Library General Public License instead.) You can apply it to your programs, too.

When we speak of free software, we are referring to freedom, not price. Our General Public Licenses are designed to make sure that you have the freedom to distribute copies of free software (and charge for this service if you wish), that you receive source code or can get it if you want it, that you can change the software or use pieces of it in new free programs; and that you know you can do these things.

To protect your rights, we need to make restrictions that forbid anyone to deny you these rights or to ask you to surrender the rights. These restrictions translate to certain responsibilities for you if you distribute copies of the software, or if you modify it.

For example, if you distribute copies of such a program, whether gratis or for a fee, you must give the recipients all the rights that you have. You must make sure that they, too, receive or can get the source code. And you must show them these terms so they know their rights.

We protect your rights with two steps: (1) copyright the software, and (2) offer you this license which gives you legal permission to copy, distribute and/or modify the software.

Also, for each author's protection and ours, we want to make certain that everyone understands that there is no warranty for this free software. If the software is modified by someone else and passed on, we want its recipients to know that what they have is not the original, so that any problems introduced by others will not reflect on the original authors' reputations.

Finally, any free program is threatened constantly by software patents. We wish to avoid the danger that redistributors of a free program will individually obtain patent licenses, in effect making the program proprietary. To prevent this, we have made it clear that any patent must be licensed for everyone's free use or not licensed at all.

The precise terms and conditions for copying, distribution and modification follow.

TERMS AND CONDITIONS FOR COPYING, DISTRIBUTION AND MODIFICATION

0. This License applies to any program or other work which contains a notice placed by the copyright holder saying it may be distributed under the terms of this General Public License. The "Program", below, refers to any such program or work, and a "work based on the Program" means either the Program or any derivative work under copyright law: that is to say, a work containing the Program or a portion of it, either verbatim or with modifications and/or translated into another language. (Hereinafter, translation is included without limitation in the term "modification".) Each licensee is addressed as "you".

   Activities other than copying, distribution and modification are not covered by this License; they are outside its scope. The act of running the Program is not restricted, and the output from the Program is covered only if its contents constitute a work based on the Program (independent of having been made by running the Program). Whether that is true depends on what the Program does.

1. You may copy and distribute verbatim copies of the Program's source code as you receive it, in any medium, provided that you conspicuously and appropriately publish on each copy an appropriate copyright notice and disclaimer of warranty; keep intact all the notices that refer to this License and to the absence of any warranty; and give any other recipients of the Program a copy of this License along with the Program.

   You may charge a fee for the physical act of transferring a copy, and you may at your option offer warranty protection in exchange for a fee.

2. You may modify your copy or copies of the Program or any portion of it, thus forming a work based on the Program, and copy and distribute such modifications or work under the terms of Section 1 above, provided that you also meet all of these conditions:

   a. You must cause the modified files to carry prominent notices stating that you changed the files and the date of any change.

b. You must cause any work that you distribute or publish, that in whole or in part contains or is derived from the Program or any part thereof, to be licensed as a whole at no charge to all third parties under the terms of this License.

c. If the modified program normally reads commands interactively when run, you must cause it, when started running for such interactive use in the most ordinary way, to print or display an announcement including an appropriate copyright notice and a notice that there is no warranty (or else, saying that you provide a warranty) and that users may redistribute the program under these conditions, and telling the user how to view a copy of this License. (Exception: if the Program itself is interactive but does not normally print such an announcement, your work based on the Program is not required to print an announcement.)

These requirements apply to the modified work as a whole. If identifiable sections of that work are not derived from the Program, and can be reasonably considered independent and separate works in themselves, then this License, and its terms, do not apply to those sections when you distribute them as separate works. But when you distribute the same sections as part of a whole which is a work based on the Program, the distribution of the whole must be on the terms of this License, whose permissions for other licensees extend to the entire whole, and thus to each and every part regardless of who wrote it.

Thus, it is not the intent of this section to claim rights or contest your rights to work written entirely by you; rather, the intent is to exercise the right to control the distribution of derivative or collective works based on the Program.

In addition, mere aggregation of another work not based on the Program with the Program (or with a work based on the Program) on a volume of a storage or distribution medium does not bring the other work under the scope of this License.

3. You may copy and distribute the Program (or a work based on it, under Section 2) in object code or executable form under the terms of Sections 1 and 2 above provided that you also do one of the following:

   a. Accompany it with the complete corresponding machine-readable source code, which must be distributed under the terms of Sections 1 and 2 above on a medium customarily used for software interchange; or,

   b. Accompany it with a written offer, valid for at least three years, to give any third party, for a charge no more than your cost of physically performing source distribution, a complete machine-readable copy of the corresponding source code, to be distributed under the terms of Sections 1 and 2 above on a medium customarily used for software interchange; or,

   c. Accompany it with the information you received as to the offer to distribute corresponding source code. (This alternative is allowed only for noncommercial distribution and only if you received the program in object code or executable form with such an offer, in accord with Subsection b above.)

   The source code for a work means the preferred form of the work for making modifications to it. For an executable work, complete source code means all the source code for all modules it contains, plus any associated interface definition files, plus the scripts used to control compilation and installation of the executable. However, as a special exception, the source code distributed need not include anything that is normally distributed (in either source or binary form) with the major components (compiler, kernel, and so on) of the operating system on which the executable runs, unless that component itself accompanies the executable.

If distribution of executable or object code is made by offering access to copy from a designated place, then offering equivalent access to copy the source code from the same place counts as distribution of the source code, even though third parties are not compelled to copy the source along with the object code.

4. You may not copy, modify, sublicense, or distribute the Program except as expressly provided under this License. Any attempt otherwise to copy, modify, sublicense or distribute the Program 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.
5. You are not required to accept this License, since you have not signed it. However, nothing else grants you permission to modify or distribute the Program or its derivative works. These actions are prohibited by law if you do not accept this License. Therefore, by modifying or distributing the Program (or any work based on the Program), you indicate your acceptance of this License to do so, and all its terms and conditions for copying, distributing or modifying the Program or works based on it.
6. Each time you redistribute the Program (or any work based on the Program), the recipient automatically receives a license from the original licensor to copy, distribute or modify the Program subject to these terms and conditions. You may not impose any further restrictions on the recipients' exercise of the rights granted herein. You are not responsible for enforcing compliance by third parties to this License.
7. If, as a consequence of a court judgment or allegation of patent infringement or for any other reason (not limited to patent issues), conditions are imposed on you (whether by court order, agreement or otherwise) that contradict the conditions of this License, they do not excuse you from the conditions of this License. If you cannot distribute so as to satisfy simultaneously your obligations under this License and any other pertinent obligations, then as a consequence you may not distribute the Program at all. For example, if a patent license would not permit royalty-free redistribution of the Program by all those who receive copies directly or indirectly through you, then the only way you could satisfy both it and this License would be to refrain entirely from distribution of the Program.

   If any portion of this section is held invalid or unenforceable under any particular circumstance, the balance of the section is intended to apply and the section as a whole is intended to apply in other circumstances.

   It is not the purpose of this section to induce you to infringe any patents or other property right claims or to contest validity of any such claims; this section has the sole purpose of protecting the integrity of the free software distribution system, which is implemented by public license practices. Many people have made generous contributions to the wide range of software distributed through that system in reliance on consistent application of that system; it is up to the author/donor to decide if he or she is willing to distribute software through any other system and a licensee cannot impose that choice.

This section is intended to make thoroughly clear what is believed to be a consequence of the rest of this License.

8. If the distribution and/or use of the Program is restricted in certain countries either by patents or by copyrighted interfaces, the original copyright holder who places the Program under this License may add an explicit geographical distribution limitation excluding those countries, so that distribution is permitted only in or among countries not thus excluded. In such case, this License incorporates the limitation as if written in the body of this License.
9. The Free Software Foundation may publish revised and/or new versions of the General Public 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.

   Each version is given a distinguishing version number. If the Program specifies a version number of this License which applies to it and "any later version", you have the option of following the terms and conditions either of that version or of any later version published by the Free Software Foundation. If the Program does not specify a version number of this License, you may choose any version ever published by the Free Software Foundation.
10. If you wish to incorporate parts of the Program into other free programs whose distribution conditions are different, write to the author to ask for permission. For software which is copyrighted by the Free Software Foundation, write to the Free Software Foundation; we sometimes make exceptions for this. Our decision will be guided by the two goals of preserving the free status of all derivatives of our free software and of promoting the sharing and reuse of software generally.

NO WARRANTY

11. BECAUSE THE PROGRAM IS LICENSED FREE OF CHARGE, THERE IS NO WARRANTY FOR THE PROGRAM, TO THE EXTENT PERMITTED BY APPLICABLE LAW. EXCEPT WHEN OTHERWISE STATED IN WRITING THE COPYRIGHT HOLDERS AND/OR OTHER PARTIES PROVIDE THE PROGRAM "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESSED OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. THE ENTIRE RISK AS TO THE QUALITY AND PERFORMANCE OF THE PROGRAM IS WITH YOU. SHOULD THE PROGRAM PROVE DEFECTIVE, YOU ASSUME THE COST OF ALL NECESSARY SERVICING, REPAIR OR CORRECTION.
12. IN NO EVENT UNLESS REQUIRED BY APPLICABLE LAW OR AGREED TO IN WRITING WILL ANY COPYRIGHT HOLDER, OR ANY OTHER PARTY WHO MAY MODIFY AND/OR REDISTRIBUTE THE PROGRAM AS PERMITTED ABOVE, BE LIABLE TO YOU FOR DAMAGES, INCLUDING ANY GENERAL, SPECIAL, INCIDENTAL OR CONSEQUENTIAL DAMAGES ARISING OUT OF THE USE OR INABILITY TO USE THE PROGRAM (INCLUDING BUT NOT LIMITED TO LOSS OF DATA OR DATA BEING RENDERED INACCURATE OR LOSSES SUSTAINED BY YOU OR THIRD PARTIES OR A FAILURE OF THE PROGRAM TO OPERATE WITH ANY OTHER PROGRAMS), EVEN IF SUCH

HOLDER OR OTHER PARTY HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

END OF TERMS AND CONDITIONS

How to Apply These Terms to Your New Programs

If you develop a new program, and you want it to be of the greatest possible use to the public, the best way to achieve this is to make it free software which everyone can redistribute and change under these terms.

To do so, attach the following notices to the program. It is safest to attach them to the start of each source file to most effectively convey the exclusion of warranty; and each file should have at least the "copyright" line and a pointer to where the full notice is found.

```
one line to give the program's name and a brief idea of what it does.
Copyright (C) year  name of author

This program is free software; you can redistribute it and/or modify
it under the terms of the GNU General Public License as published by
the Free Software Foundation; either version 2 of the License, or
(at your option) any later version.

This program is distributed in the hope that it will be useful,
but WITHOUT ANY WARRANTY; without even the implied warranty of
MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
GNU General Public License for more details.

You should have received a copy of the GNU General Public License
along with this program; if not, write to the Free Software
Foundation, Inc., 59 Temple Place - Suite 330,
Boston, MA 02111-1307, USA.
```

Also add information on how to contact you by electronic and paper mail.

If the program is interactive, make it output a short notice like this when it starts in an interactive mode:

```
Gnomovision version 69, Copyright (C) year  name of author
Gnomovision comes with ABSOLUTELY NO WARRANTY; for details
type `show w'.
This is free software, and you are welcome to redistribute it
under certain conditions; type `show c' for details.
```

The hypothetical commands '`show w`' and '`show c`' should show the appropriate parts of the General Public License. Of course, the commands you use may be called something other than '`show w`' and '`show c`'; they could even be mouse-clicks or menu items—whatever suits your program.

You should also get your employer (if you work as a programmer) or your school, if any, to sign a "copyright disclaimer" for the program, if necessary. Here is a sample; alter the names:

```
Yoyodyne, Inc., hereby disclaims all copyright interest in the program
`Gnomovision' (which makes passes at compilers) written by James Hacker.

signature of Ty Coon, 1 April 1989
Ty Coon, President of Vice
```

This General Public License does not permit incorporating your program into proprietary programs. If your program is a subroutine library, you may consider it more useful to permit linking proprietary applications with the library. If this is what you want to do, use the GNU Library General Public License instead of this License.

Word Index

This index is a list of Forth words that have “glossary” entries within this manual. Each word is listed with its stack effect and wordset.

:

;

<

=

>

?

@

[

]

\

~

0

1

2

A

B

C

D

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W

X

Concept and Word Index

Not all entries listed in this index are present verbatim in the text. This index also duplicates, in abbreviated form, all of the words listed in the Word Index (only the names are listed for the words here).

.

/

:

;

<

=

>

?

@

[

]

\

~

0

1

2

A

B

C

D

E

J

K

L

M

N

O

P

Q

R

S

T

X

Z