Gforth Manual

for version 0.4.0

Anton Ertl
Bernd Paysan
Jens Wilke

This manual is permanently under construction

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) 19*yy* *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., 675 Mass Ave, Cambridge, MA 02139, 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) 19*yy* *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.

Preface

This manual documents Gforth. The reader is expected to know Forth. This manual is primarily a reference manual. See Chapter 2 [Other Books], page 10 for introductory material.

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 Forth standard (ANS Forth).
- 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 11.4 [Performance], page 102). It is free and available on many machines.

2 Other books on ANS Forth

As the standard is relatively new, there are not many books out yet. It is not recommended to learn Forth by using Gforth and a book that is not written for ANS Forth, as you will not know your mistakes from the deviations of the book. However, books based on the Forth-83 standard should be ok, because ANS Forth is primarily an extension of Forth-83.

There is, of course, the standard, the definite reference if you want to write ANS Forth programs. It is available in printed form from the National Standards Institute Sales Department (Tel.: USA (212) 642-4900; Fax.: USA (212) 302-1286) as document *X3.215-1994* for about $200. You can also get it from Global Engineering Documents (Tel.: USA (800) 854-7179; Fax.: (303) 843-9880) for about $300.

dpANS6, the last draft of the standard, which was then submitted to ANSI for publication is available electronically and for free in some MS Word format, and it has been converted to HTML (`http://www.taygeta.com/forth/dpans.html`; this is my favourite format); this HTML version also includes the answers to Requests for Interpretation (RFIs). Some pointers to these versions can be found through
`http://www.complang.tuwien.ac.at/projects/forth.html`.

Forth: The New Model by Jack Woehr (Prentice-Hall, 1993) is an introductory book based on a draft version of the standard. It does not cover the whole standard. It also contains interesting background information (Jack Woehr was in the ANS Forth Technical Committee). It is not appropriate for complete newbies, but programmers experienced in other languages should find it ok.

Forth Programmer's Handbook by Edward K. Conklin, Elizabeth D. Rather and the technical staff of Forth, Inc. (Forth, Inc., 1997; ISBN 0-9662156-0-5) contains little introductory material. The majority of the book is similar to Chapter 4 [Words], page 13, but the book covers most of the standard words and some non-standard words (whereas this manual is quite incomplete). In addition, the book contains a chapter on programming style. The major drawback of this book is that it usually does not identify what is standard and what is specific to the Forth system described in the book (probably one of Forth, Inc.'s systems). Fortunately, many of the non-standard programming practices described in the book work in Gforth, too. Still, this drawback makes the book hardly more useful than a pre-ANS book.

3 Invoking Gforth

You will usually just say `gforth`. In many other cases the default Gforth image will be invoked like this:

```
gforth [files] [-e forth-code]
```

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

In general, the command line looks like this:

```
gforth [initialization options] [image-specific options]
```

The initialization 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 10 [Image Files], page 92).

`--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 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 10.4 [Data-Relocatable Image Files], page 93).

`--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 10.4 [Data-Relocatable Image Files], page 93).

`--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.

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 several `-e`s. 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. You may want to use the option `--path`, if your environment contains the variable `GFORTHPATH`.

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 first searched in '`.`', then in '`~`', then in the normal path (see above).

4 Forth Words

4.1 Notation

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

word *Stack effect* *wordset* *pronunciation*

Description

word The name of the word. BTW, Gforth is case insensitive, so you can type the words in in lower case (However, see Section 6.1.1 [core-idef], page 75).

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 wordsets. A standard system need not support all of them. So, the fewer wordsets your program uses the more portable it will be in theory. However, we suspect that most ANS Forth systems on personal machines will feature all wordsets. Words that are not defined in the ANS standard have `gforth` or `gforth-internal` as wordset. `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 wordset 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`	Wordlist ID, same size as Cell
`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.

4.2 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 4.2.3 [Mixed precision], page 15).

4.2.1 Single precision

`+`	*n1 n2 – n*	core	“plus”
`-`	*n1 n2 – n*	core	“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`	*n1 – n2*	core	“abs”
`min`	*n1 n2 – n*	core	“min”
`max`	*n1 n2 – n*	core	“max”

4.2.2 Bitwise operations

`and` *w1 w2 – w* core "and"

`or` *w1 w2 – w* core "or"

`xor` *w1 w2 – w* core "xor"

`invert` *w1 – w2* core "invert"

`2*` *n1 – n2* core "two-star"

`2/` *n1 – n2* core "two-slash"

4.2.3 Mixed precision

`m+` *d1 n – d2* double "m-plus"

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

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

`m*` *n1 n2 – d* core "m-star"

`um*` *u1 u2 – ud* core "u-m-star"

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

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

`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

4.2.4 Double precision

The outer (aka text) interpreter converts numbers containing a dot into a double precision number. Note that only numbers with the dot as last character are standard-conforming.

`d+` *d1 d2 – d* double "d-plus"

`d-` *d1 d2 – d* double "d-minus"

`dnegate` *d1 – d2* double "dnegate"

`dabs` *d1 – d2* double "dabs"

`dmin` *d1 d2 – d* double "dmin"

`dmax` *d1 d2 – d* double "dmax"

4.2.5 Floating Point

The format of floating point numbers recognized by the outer (aka text) interpreter is: a signed decimal number, possibly containing a decimal point (`.`), followed by `E` or `e`, optionally followed by a signed integer (the exponent). E.g., `1e` is the same as `+1.0e+0`. Note that a number without `e` is not interpreted as floating-point number, but as double (if the number contains a `.`) or single precision integer. Also, conversions between string and floating point numbers always use base 10, irrespective of the value of `BASE` (in Gforth; for

the standard this is an ambiguous condition). If `BASE` contains a value greater then 14, the `E` may be interpreted as digit and the number will be interpreted as integer, unless it has a signed exponent (both `+` and `-` are allowed as signs).

Angles in floating point operations are given in radians (a full circle has 2 pi radians). Note, that Gforth has a separate floating point stack, but we use the unified notation.

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, ACM Computing Surveys 23(1):5–48, March 1991* (`http://www.validgh.com/goldberg/paper.ps`).

`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 "fnegate"

`fabs` *r1 – r2* float-ext "fabs"

`fmax` *r1 r2 – r3* float "fmax"

`fmin` *r1 r2 – r3* float "fmin"

`floor` *r1 – r2* float "floor"

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

`fround` *r1 – r2* float "fround"

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 "fsqrt"

`fexp` *r1 – r2* float-ext "fexp"

`fexpm1` *r1 – r2* float-ext "fexpm1"

r2=*e****r1*−1

`fln` *r1 – r2* float-ext "fln"

`flnp1` *r1 – r2* float-ext "flnp1"

r2=ln(*r1*+1)

`flog` *r1 – r2* float-ext "flog"

the decimal logarithm

`falog` *r1 – r2* float-ext "falog"

r2=10***r1*

`fsin` *r1 – r2* float-ext "fsin"

`fcos` *r1 – r2* float-ext "fcos"

`fsincos` *r1 – r2 r3* float-ext "fsincos"

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

`ftan` *r1 – r2* float-ext "ftan"
`fasin` *r1 – r2* float-ext "fasin"
`facos` *r1 – r2* float-ext "facos"
`fatan` *r1 – r2* float-ext "fatan"
`fatan2` *r1 r2 – r3* float-ext "fatan2"

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

`fsinh` *r1 – r2* float-ext "fsinh"
`fcosh` *r1 – r2* float-ext "fcosh"
`ftanh` *r1 – r2* float-ext "ftanh"
`fasinh` *r1 – r2* float-ext "fasinh"
`facosh` *r1 – r2* float-ext "facosh"
`fatanh` *r1 – r2* float-ext "fatanh"

4.3 Stack Manipulation

Gforth has a data stack (aka parameter stack) for characters, cells, addresses, and double cells, a floating point stack for floating point numbers, a return stack for storing the return addresses of colon definitions and other data, and a locals stack for storing local variables. Note that while every sane Forth has a separate floating point stack, this 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 FP stack.

Also, a Forth system is allowed to keep the 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 complying program and if you are using local variables in a word, forget about return stack manipulations in that word (see the standard document for the exact rules).

4.3.1 Data stack

`drop` *w –* core "drop"
`nip` *w1 w2 – w2* core-ext "nip"
`dup` *w – w w* core "dup"
`over` *w1 w2 – w1 w2 w1* core "over"
`tuck` *w1 w2 – w2 w1 w2* core-ext "tuck"
`swap` *w1 w2 – w2 w1* core "swap"
`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"
`pick` *u – w* core-ext "pick"

`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”

4.3.2 Floating point stack

`fdrop` *r –* float “fdrop”
`fnip` *r1 r2 – r2* gforth “fnip”
`fdup` *r – r r* float “fdup”
`fover` *r1 r2 – r1 r2 r1* float “fover”
`ftuck` *r1 r2 – r2 r1 r2* gforth “ftuck”
`fswap` *r1 r2 – r2 r1* float “fswap”
`frot` *r1 r2 r3 – r2 r3 r1* float “frot”

4.3.3 Return stack

`>r` *w –* core “to-r”
`r>` *– w* core “r-from”
`r@` *– w ; R: w – w* core “r-fetch”
copy w from the return stack to the data stack
`rdrop` *–* gforth “rdrop”
`2>r` *w1 w2 –* core-ext “two-to-r”
`2r>` *– w1 w2* core-ext “two-r-from”
`2r@` *– w1 w2* core-ext “two-r-fetch”
`2rdrop` *–* gforth “two-r-drop”

4.3.4 Locals stack

4.3.5 Stack pointer manipulation

`sp@` *– a-addr* gforth “spat”
`sp!` *a-addr –* gforth “spstore”
`fp@` *– f-addr* gforth “fp-fetch”
`fp!` *f-addr –* gforth “fp-store”
`rp@` *– a-addr* gforth “rpat”
`rp!` *a-addr –* gforth “rpstore”
`lp@` *– addr* gforth “l-p-fetch”
`lp!` *c-addr –* gforth “lp-store”

4.4 Memory

4.4.1 Memory Access

`@`	*a-addr – w*	core	“fetch”
`!`	*w a-addr –*	core	“store”
`+!`	*n a-addr –*	core	“plus-store”
`c@`	*c-addr – c*	core	“cfetch”
`c!`	*c c-addr –*	core	“cstore”
`2@`	*a-addr – w1 w2*	core	“two-fetch”
`2!`	*w1 w2 a-addr –*	core	“two-store”
`f@`	*f-addr – r*	float	“f-fetch”
`f!`	*r f-addr –*	float	“f-store”
`sf@`	*sf-addr – r*	float-ext	“s-f-fetch”
`sf!`	*r sf-addr –*	float-ext	“s-f-store”
`df@`	*df-addr – r*	float-ext	“d-f-fetch”
`df!`	*r df-addr –*	float-ext	“d-f-store”

4.4.2 Address arithmetic

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. Basically, 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 systems where it is a noop, it compiles to nothing).

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.

The standard 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 standard defines a word `char`, which has nothing to do with address arithmetic.

`chars`	*n1 – n2*	core	“chars”
`char+`	*c-addr1 – c-addr2*	core	“care-plus”
`cells`	*n1 – n2*	core	“cells”
`cell+`	*a-addr1 – a-addr2*	core	“cell-plus”

`cell` – *u* gforth "cell"
`align` – core "align"
`aligned` *c-addr* – *a-addr* core "aligned"
`floats` *n1* – *n2* float "floats"
`float+` *f-addr1* – *f-addr2* float "float-plus"
`float` – *u* gforth "float"
`falign` – float "falign"
`faligned` *c-addr* – *f-addr* float "f-aligned"
`sfloats` *n1* – *n2* float-ext "s-floats"
`sfloat+` *sf-addr1* – *sf-addr2* float-ext "s-float-plus"
`sfalign` – float-ext "s-f-align"
`sfaligned` *c-addr* – *sf-addr* float-ext "s-f-aligned"
`dfloats` *n1* – *n2* float-ext "d-floats"
`dfloat+` *df-addr1* – *df-addr2* float-ext "d-float-plus"
`dfalign` – float-ext "d-f-align"
`dfaligned` *c-addr* – *df-addr* float-ext "d-f-aligned"
`maxalign` – float "maxalign"
`maxaligned` *addr* – *f-addr* float "maxaligned"
`cfalign` – gforth "cfalign"
`cfaligned` *addr1* – *addr2* gforth "cfaligned"
`ADDRESS-UNIT-BITS` – *n* environment "ADDRESS-UNIT-BITS"

4.4.3 Memory Blocks

`move` *c-from c-to ucount* – core "move"
`erase` *addr len* – core-ext "erase"

While the previous words work on address units, the rest works on characters.

`cmove` *c-from c-to u* – string "cmove"
`cmove>` *c-from c-to u* – string "c-move-up"
`fill` *c-addr u c* – core "fill"
`blank` *addr len* – string "blank"

4.5 Control Structures

Control structures in Forth cannot be used in interpret state, only in compile state[1], i.e., in a colon definition. We do not like this limitation, but have not seen a satisfying way around it yet, although many schemes have been proposed.

[1] More precisely, they have no interpretation semantics (see Section 4.7.5 [Interpretation and Compilation Semantics], page 38)

4.5.1 Selection

flag
IF
  code
ENDIF

or

flag
IF
  code1
ELSE
  code2
ENDIF

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 plain standard 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
  ...
ENDCASE

Executes the first *codei*, where the *ni* is equal to *n*. A 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.

4.5.2 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.

```
BEGIN
  code
AGAIN
```

This is an endless loop.

4.5.3 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, aka index, can be accessed with `i`. E.g., 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` – *n* core "i"

`j` – *n* core "j"

`k` – *n* 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. E.g., if you put values on the return stack outside the loop, you cannot read them inside the loop. 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.

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.

`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 in the ANS Forth standard. However, an implementation for these words that uses only standard words is provided in '`compat/loops.fs`'.

`?DO` can also 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.

`UNLOOP` is used to prepare 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.

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. 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.

4.5.4 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 “CS-PICK”

`CS-ROLL` *destu/origu .. dest0/orig0 u – .. dest0/orig0 destu/origu* tools-ext “CS-ROLL”

On many systems control-flow stack items take one word, in Gforth they currently take three (this may change in the future). Therefore it is a really good idea to manipulate the control flow stack with `cs-pick` and `cs-roll`, not with data stack manipulation words.

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` – core “unloop”

`DONE` *compilation orig – ; run-time –* gforth “DONE”

The standard does not allow using `cs-pick` and `cs-roll` on *do-sys*. Our system 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`.

4.5.4.1 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.

E.g., 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.

4.5.5 Calls and returns

A definition can be called simply be writing the name of the definition to be called. Note that normally a definition is invisible during its definition. If you want to write a directly recursive definition, you can use `recursive` to make the current definition visible.

`recursive` *compilation – ; run-time –* gforth "recursive"

makes the current definition visible, enabling it to call itself recursively.

Another way to perform a recursive call is

`recurse` *compilation – ; run-time ?? – ??* core "recurse"

calls 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
```

When the end of the definition is reached, it returns. An earlier return can be forced using

`EXIT` *compilation – ; run-time nest-sys –* core "EXIT"

Don't forget to clean up the return stack and `UNLOOP` any outstanding `?DO`...`LOOP`s before `EXIT`ing. The primitive compiled by `EXIT` is

`;s` – gforth "semis"

4.5.6 Exception Handling

`catch` *... xt – ... n* exception "catch"

`throw` *y1 .. ym error/0 – y1 .. ym / z1 .. zn error* exception "throw"

4.6 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 the paper *Automatic Scoping of Local Variables* by M. Anton Ertl, presented at EuroForth '94; it is available at `http://www.complang.tuwien.ac.at/papers/ertl94l.ps.gz`.

4.6.1 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:

4.6.1.1 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[2]. 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 }
  ...
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

[2] In compiler construction terminology, all places dominated by the definition of the local.

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
```

4.6.1.2 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).

4.6.1.3 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 `PICK`s and `ROLL`s 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` in Section 4.6.1 [Gforth locals], page 26 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.

4.6.1.4 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#` – *w* gforth "fetch-local-number"

`f@local#` – *r* gforth "f-fetch-local-number"

`laddr#` – *c-addr* gforth "laddr-number"

`lp+!#` – 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-@local` *n* – gforth "compile-fetch-local"

`compile-f@local` *n* – gforth "compile-f-fetch-local"

`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 wordlist (aka vocabulary) has its own methods for searching etc. (see Section 4.11 [Wordlists], page 64). For the present purpose we defined a wordlist 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 wordlist containing `}`, `W:` etc., and then the wordlist 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 wordlist 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 wordlists:

`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 wordlist implementation make these operations easy to implement: The locals wordlists 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.

4.6.2 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 this way behave like `VALUE`s (See Section 4.7.1 [Simple Defining Words], page 34). I.e., they are initialized from the stack. Using their name produces their value. Their value can be changed using `TO`.

Since this syntax 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 the following word

`(local)` *addr u* – local "paren-local-paren"

The ANS Forth locals extension wordset defines a syntax, 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.

4.7 Defining Words

4.7.1 Simple Defining Words

`Constant` *w* "*name*" – core "Constant"

Defines constant *name*

name execution: – *w*

`2Constant` *w1 w2* "*name*" – double "2Constant"

`fconstant` *r* "*name*" – float "fconstant"

`Variable` "*name*" – core "Variable"

`2Variable` "*name*" – double "2Variable"

`fvariable` "*name*" – float "fvariable"

`Create` "*name*" – core "Create"

`User` "*name*" – gforth "User"

`Value` *w* "*name*" – core-ext "Value"

`TO` *addr* "*name*" – core-ext "TO"

`Defer` "*name*" – gforth "Defer"

`IS` *addr* "*name*" – gforth "IS"

4.7.2 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. See Section 4.7.5 [Interpretation and Compilation Semantics], page 38 for an in-depth discussion of some of the issues involved.

`:` "*name*" – *colon-sys* core "colon"

`;` *compilation colon-sys* – *; run-time nest-sys* core "semicolon"

4.7.3 User-defined Defining Words

You can create new defining words simply by wrapping defining-time code around existing defining words and putting the sequence in a colon definition.

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
```

Technically, this fragment defines a defining word `def-word`, and a word `name`; 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`).

In other words, if you make the following definitions:

```
: def-word1 ( "name" -- )
    Create code1 ;

: action1 ( ... -- ... )
    code2 ;

def-word name1
```

Using `name1` `action1` is equivalent to using `name`.

E.g., you can implement `Constant` in this way:

```
: constant ( w "name" -- )
    create ,
DOES> ( -- w )
    @ ;
```

When you create a constant with `5 constant five`, first a new word `five` is created, then the value 5 is laid down in the body of `five` with `,`. When `five` is invoked, the address of the body is put on the stack, and `@` retrieves the value 5.

In the example 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).

4.7.3.1 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 -- )
    create ,
DOES> ( n2 -- n1+n2 )
    @ + ;

 3 curry+ 3+
-2 curry+ 2-
```

4.7.3.2 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; E.g., 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 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:

```
CREATE name ( ... -- ... )
  initialization
DOES>
  code ;
```

This is equivalent to the standard

```
:noname
DOES>
    code ;
CREATE name EXECUTE ( ... -- ... )
    initialization
```

You can get the address of the body of a word with

`>body` *xt – a-addr* core "to-body"

4.7.4 Supplying names for the defined words

By default, defining words take the names for the defined words 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"

E.g.,

```
s" foo" nextname create
```

is equivalent to

```
create foo
```

Sometimes you want to define a word without a name. You can do this with

`noname` – gforth "noname"

To make any use of the newly defined word, you need its execution token. You can get it with

`lastxt` – *xt* gforth "lastxt"

E.g., you can initialize a deferred word with an anonymous colon definition:

```
Defer deferred
noname : ( ... -- ... )
  ... ;
lastxt IS deferred
```

`lastxt` also works when the last word was not defined as `noname`.

The standard has also recognized the need for anonymous words and provides

`:noname` – *xt colon-sys* core-ext "colon-no-name"

This leaves the execution token for the word on the stack after the closing `;`. You can rewrite the last example with `:noname`:

```
Defer deferred
:noname ( ... -- ... )
  ... ;
IS deferred
```

4.7.5 Interpretation and Compilation Semantics

The *interpretation semantics* of a 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 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[3] 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.[4]

You can change the compilation semantics into `execute`ing the execution semantics with

`immediate` – core "immediate"

You can remove the interpretation semantics of a word with

`compile-only` – gforth "compile-only"

`restrict` – gforth "restrict"

Note that ticking (`'`) compile-only words gives an error ("Interpreting a compile-only word").

Gforth also allows you to define words with arbitrary combinations 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 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 also might try to provide an optimizing implementation of the default compilation semantics with this feature, like this:

```
:noname
   foo bar ;
:noname
   POSTPONE foo POSTPONE bar ;
interpret/compile: foobar
```

as an optimizing version of

[3] In standard terminology, "appends to the current definition".

[4] 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.

```
: 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] foobar` would compile the compilation semantics for the optimizing `foobar`, whereas it would compile the interpretation semantics for the non-optimizing `foobar`.

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
```

While this works if `foobar` is processed only 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!

It is also possible to write defining words that define words with arbitrary combinations of interpretation and compilation semantics. In general, this looks like:

```
: def-word
    create-interpret/compile
    code1
interpretation>
    code2
<interpretation
compilation>
    code3
<compilation ;
```

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”

Note that 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`.

4.8 Structures

This section presents the structure package that comes with Gforth. A version of the package implemented in plain 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 ???. Marcel Hendrix provided helpful comments.

4.8.1 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 4.4.2 [Address arithmetic], page 19). 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 `+`, using a defining word `cfield` that includes the `+` in the action of the defined word offers itself:

```
: 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.

4.8.2 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 also get the the size of a list node with `list% %size` and it's 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
```

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.

You can also include a structure `foo%` as field of another structure, with:

```
struct
...
    foo% field ...
...
end-struct ...
```

Instead of starting with an empty structure, you can also 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:[5]

```
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 and compiling such words does not generate any code. So, in the interest of readability and maintainability you should include the word for the field when accessing the field.

4.8.3 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:

[5] 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 requires a few lines of code.

- 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 about field words as converting converts 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`.

4.8.4 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

```
+
```

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.

4.8.5 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`s 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”

`name` execution: `addr1 -- addr1+offset1`
create a field `name` with offset `offset1`, and the type given by `size align`. `offset2` is the offset of the next field, and `align2` is the alignment of all fields.

`field` *align1 offset1 align size* "*name*" – *align2 offset2* gforth “field”

name execution: (addr1 – addr2)

`float%` – *align size* gforth “float%”

`nalign` *addr1 n – addr2* gforth “nalign”

`addr2` is the aligned version of `addr1` wrt 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.

4.9 Object-oriented Forth

Gforth comes with three packets 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 packets (and others) are discussed in Section 4.9.1.13 [Comparison with other object models], page 52. All packets are written in ANS Forth and can be used with any other ANS Forth.

4.9.1 Objects

This section describes the ‘`objects.fs`’ packet. This material also has been published in *Yet Another Forth Objects Package* by Anton Ertl and appeared in Forth Dimensions 19(2), pages 37–43 (`http://www.complang.tuwien.ac.at/forth/objects/objects.html`).

This section assumes (in some places) that you have read Section 4.8 [Structures], page 40.

Marcel Hendrix provided helpful comments on this section. Andras Zsoter and Bernd Paysan helped me with the related works section.

4.9.1.1 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, I did not intend to support information hiding with this model (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 4.9.1.4 [Basic Objects Usage], page 46, See Section 4.9.1.5 [The class Object], page 47, See Section 4.9.1.6 [Creating objects], page 47.), the others are optional and independent of each other.
- An implementation in ANS Forth is available.

I have used the technique, on which this model is based, for implementing the parser generator Gray; we have also used this technique in Gforth for implementing the various flavours of wordlists (hashed or not, case-sensitive or not, special-purpose wordlists for locals etc.).

4.9.1.2 Why object-oriented programming?

Often we have to deal with several data structures (*objects*), that have to be treated similarly in some respects, but differ 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, and the object-oriented package I present here also solves this problem (and not much else).

4.9.1.3 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`) for performing 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 our model it is the object that is on the TOS when the selector is invoked. (*Receiving* comes from Smalltalks *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.

4.9.1.4 Basic Objects 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 a field `circle-radius` (which behaves just like a field in see Section 4.8 [Structures], page 40); it defines 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 invoke a selector only 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`.

4.9.1.5 The class `object`

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 can use it as ancestor for all classes. It is the only class that has no parent. It has two selectors: `construct` and `print`.

4.9.1.6 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.

4.9.1.7 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).

4.9.1.8 Class Binding

Normal selector invocations determine the method at run-time depending on the class of the receiving object (late binding).

Sometimes we want to invoke a different method. E.g., assume that you 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.

4.9.1.9 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`.[6]

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
```

4.9.1.10 Classes and Scoping

Inheritance is frequent, unlike structure extension. This exacerbates the problem with the field name convention (see Section 4.8.3 [Structure Naming Convention], page 42): 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

[6] 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 ;`

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`) wordlist that was in effect before. If you have several `protected`s without an intervening `public` or `set-current`, `public` will restore the compilation wordlist in effect before the first of these `protected`s.

4.9.1.11 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` foo that implements it. The code for this would look like this:

```
interface
  selector write ( file object -- )
  selector read1 ( file object -- )
end-interface storage

bar class
  storage implementation

... overrides write
... overrides read
...
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.

4.9.1.12 '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*[7] is an array that contains the execution tokens (XTs) of the methods for the object's class. Each selector contains an offset into the method maps.

`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:

```
( 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).

At the start, a new class has 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 wordlist 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 wordlists of the class an all its ancestors on the search order, 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).

[7] This is Self terminology; in C++ terminology: virtual function table.

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, assuming a cell size of 4.

4.9.1.13 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). Here I'll discuss the relation of '`objects.fs`' to two well-known and two closely-related (by the use of method maps) models.

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). The Neon model uses a *selector object* syntax, which makes it unnatural to pass objects on the stack. It also requires that the selector parses the input stream (at compile time); this leads to reduced extensibility and to bugs that are hard to find. Finally, it allows using every selector to every object; this eliminates the need for classes, but makes it harder to create efficient implementations. A longer version of this critique can be found in *On Standardizing Object-Oriented Forth Extensions* (Forth Dimensions, May 1997) by Anton Ertl.

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?* (Forth Dimensions ??? 1996) 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 the Zsoter's model, because the receiving object is the active object already; OTOH, 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 at `http://www.forth.org/fig/oopf.html`.

The '`oof.fs`' model combines information hiding and overloading resolution (by keeping names in various wordlists) 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 model is quite similar to a very stripped-down version of the Objects model, but syntactically it is a mixture of the Objects and the OOF model.

4.9.1.14 '`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”

used as `class-inst-size 2 ( class -- align size )`, gives the size specification for an instance (i.e. an object) of `class`.

`class-override!` *xt sel-xt class-map –* objects “class-override!”

`xt` is the new method for the selector `sel-xt` in `class-map`.

`construct` *... object –* objects “construct”

initializes 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”

this 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.

`drop-order` *class –* objects “drop-order”

drops `class`’s wordlists from the search order. No checking is made whether `class`’s wordlists are actually on the search order.

`end-class` *align offset "name" –* objects “end-class”

`name` execution: `-- class`
ends a class definition. The resulting class is `class`.

`end-class-noname` *align offset – class* objects “end-class-noname”

ends a class definition. The resulting class is `class`.

`end-interface` *"name" –* objects “end-interface”

`name` execution: `-- interface`
ends an interface definition. The resulting interface is `interface`.

`end-interface-noname` *– interface* objects “end-interface-noname”

ends an interface definition. The resulting interface is `interface`.

`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”

initializes 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"

starts an interface definition.

`;m` *colon-sys –; run-time:* – objects ";m"

end a method definition; restore old `this`.

`m:` – *xt colon-sys; run-time: object* – objects "m:"

start a method definition; `object` becomes new `this`.

`method` *xt* "*name*" – objects "method"

`name` execution: `... object -- ...`
creates selector `name` and makes `xt` its method in the current class.

`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"

prints 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.

`push-order` *class* – objects "push-order"

add `class`'s wordlists to the search-order (in front)

`selector` "*name*" – objects "selector"

`name` execution: `... object -- ...`
creates 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"

sets `this` (used internally, but useful when debugging).

`xt-new` *... class xt – object* objects "xt-new"

makes a new object, using `xt ( align size -- addr )` to get memory.

4.9.2 OOF

This section describes the '`oof.fs`' packet. This section uses the same rationale why using object-oriented programming, and the same terminology.

The packet described in this section is 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.

4.9.2.1 Properties of the OOF 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 straight forward, 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.

4.9.2.2 Basic OOF Usage

Here, I use the same example as for `objects` (see Section 4.9.1.4 [Basic Objects Usage], page 46).

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 invoke a selector only 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.

4.9.2.3 The base class 'object'

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 destroctor 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 expicit scoping. You should use expicit scoping only for super classes or classes with the same set of instance variables. Explicit 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 enabling it's scope. Using `with` and `endwith` also allows to create code using selector `postpone` without being trapped bye the state-smart objects.

  `with` *o* – oof "with"

  `endwith` – oof "endwith"

4.9.2.4 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, casted 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

4.9.2.5 Class Implementation

4.9.3 Mini-OOF

Gforth's third object oriented Forth package is a 12-liner. It uses a bit of a mixture of the '`object.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.arch.

4.9.3.1 Usage

Basically, there are seven words, to define a method, a variable, a class; to end a class, to define a method, to allocate an object, to resolve binding, and the base class (which allocates one cell for the object pointer).

`method` *m v – m' v* unknown "method"

Defines a method

`var` *m v size – m v'* unknown "var"

Defines a variable with size bytes

`class` *class – class methods vars* unknown "class"

Starts the definition of a sub-class

`end-class` *class methods vars* – unknown "end-class"

Ends the definition of a class

`defines` *xt class* "*name*" – unknown "defines"

Binds the xt to the method name in the class

`new` *class* – *o* unknown "new"

Creates a new incarnation of the class

`::` *class* "*name*" – unknown "::"

Compiles the method name of the class (not immediate!)

`object` unknown "object"

Is the base class of all objects

4.9.3.2 Mini-OOF Example

A short example shows how to use this package.

```
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
```

4.9.3.3 Mini-OOF Implementation

Object oriented system 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. This vtable may contain some other informations, too.

So first, let's declare methods:

```
: method ( m v -- m' v ) Create  over , swap cell+ swap
  DOES> ( ... o -- ... )  over  +  execute ;
```

During method declaration, the number of methods and instance variables is on the stack (in address units). `method` creates one method and increments the method number. To execute a method, it takes the object, fetches the vtable pointer, adds the offset, and executes the xt stored there. Each method takes the object it is invoked from as top of stack parameter. The method itself should consume that object.

Now, we also have to declare instance variables

```
: var ( m v size -- m v' ) Create  over , +
  DOES> ( o -- addr )  + ;
```

Same as above, 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, it will adjust the variable offset. That's why it is on the top of stack.

We need a starting point (the empty object) and some syntactic sugar:

```
Create object  1 cells , 2 cells ,
: class ( class -- class methods vars ) dup 2 ;
```

Now, 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 methods vars -- )
  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 `noop`s. 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 -- ) ' >body  + ! ;
```

To allocate a new object, we need a word, too:

```
: new ( class -- o )  here over  allot swap over ! ;
```

And sometimes derived classes want to access the method of the parent object. There are two ways to achieve this with this OOF: first, you could use named words, and second, you could look up the vtable of the parent object.

```
: :: ( class "name" -- ) ' >body  +  compile, ;
```

<H2>An Example</H2>

Nothing can be more confusing than a good example, so here is one. First let's declare a text object (further 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
```

For inheritance, we define a class `bold-button`, with no new data and no new methods.

```
button class
end-class bold-button

: bold   27 emit ." [1m" ;
: normal 27 emit ." [0m" ;

:noname bold [ button :: draw ] normal ; bold-button defines draw
```

And finally, some code to demonstrate how to create objects and apply methods:

```
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
```

4.10 Tokens for Words

This chapter describes the creation and use of tokens that represent words on the stack (and in data space).

Named words have interpretation and compilation semantics. Unnamed words just have execution semantics.

An *execution token* represents the execution semantics of an unnamed word. An execution token occupies one cell. As explained in section Section 4.7.4 [Supplying names], page 37, the execution token of the last words defined can be produced with

`lastxt` – *xt* gforth “lastxt”

You can perform the semantics represented by an execution token with

`execute` *xt* – core “execute”

You can compile the word with

`compile,` *xt* – core-ext "compile-comma"

In Gforth, the abstract data type *execution token* is implemented as CFA (code field address).

The interpretation semantics of a named word are also represented by an execution token. You can get it with

`[']` *compilation.* "*name*" – ; *run-time.* – *xt* core "bracket-tick"

xt represents *name*'s interpretation semantics. Performs `-14 throw` if the word has no interpretation semantics.

`'` "*name*" – *xt* core "tick"

xt represents *name*'s interpretation semantics. Performs `-14 throw` if the word has no interpretation semantics.

For literals, you use `'` in interpreted code and `[']` in compiled code. Gforth's `'` and `[']` behave somewhat unusual by complaining about compile-only words. To get an execution token for a compiling word *X*, use `COMP' X drop` or `[COMP'] X drop`.

The compilation semantics are represented 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.

`[COMP']` *compilation* "*name*" – ; *run-time* – *w xt* gforth "bracket-comp-tick"

w xt represents *name*'s compilation semantics.

`COMP'` "*name*" – *w xt* gforth "comp-tick"

w xt represents *name*'s compilation semantics.

You can compile the compilation semantics with `postpone,`. I.e., `COMP'` *word* `POSTPONE,` is equivalent to `POSTPONE` *word*.

`postpone,` *w xt* – gforth "postpone-comma"

Compiles the compilation semantics represented by *w xt*.

At present, the *w* part of a compilation token is an execution token, and the *xt* part represents either `execute` or `compile,`. However, don't rely on that knowledge, unless necessary; we may introduce unusual compilation tokens in the future (e.g., compilation tokens representing the compilation semantics of literals).

Named words are also represented by the *name token*. The abstract data type *name token* is implemented as NFA (name field address).

`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*. Produces `' compile-only-error` if *nt* is compile-only.

`name?int` *nt* – *xt* gforth "name?int"

Like name>int, but throws an error if compile-only.

`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*.

4.11 Wordlists

4.12 Files

This chapter describes how to operate on files from Forth.

Files have the following types for opening and creating:

`r/o` – *fam* file "r-o"

`r/w` – *fam* file "r-w"

`w/o` – *fam* file "w-o"

`bin` *fam1 – fam2* file "bin"

Files are opened/created by name and type, and return a file identifier.

`open-file` *c-addr u ntype – w2 wior* file "open-file"

`create-file` *c-addr u ntype – w2 wior* file "create-file"

This identifier is used for all other file commands.

`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* file "read-line"

`write-file` *c-addr u1 wfileid – wior* file "write-file"

`emit-file` *c wfileid – wior* gforth "emit-file"

`flush-file` *wfileid – wior* file-ext "flush-file"

`file-status` *c-addr u – ntype 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"

4.13 Including Files

4.13.1 Words for Including

`include-file` *i*x fid – j*x* file "include-file"

interpret (process using the text interpreter) the contents of the file *fid*.

`included` *i*x addr u – j*x* file "included"

`include-file` the file whose name is given by the string *addr u*.

`include` ... "*file*" – ... gforth "include"

includes *file*

Usually you want to include a file only if it is not included already (by, say, another source file):

`required` *i*x addr u – j*x* gforth "required"

include 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"

includes *file* only if it is not included already

`needs` ... "*name*" – ... gforth "needs"

an alias for `require`; exists on other systems (e.g., Win32Forth).

I recommend that you write your source files such that interpreting them does not change the stack. This allows using these files with `required` and friends without complications. E.g.,

```
1 require foo.fs drop
```

4.13.2 Search Path

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.

For relative filenames, Gforth uses a search path similar to Forth's search order (see Section 4.11 [Wordlists], page 64). It tries to find the given filename in the directories present in the path, and includes the first one it finds.

If the search path contains the directory '`.`' (as it should), 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.

Use '`~+`' to refer to the current working directory (as in the `bash`).

If the filename starts with '`./`', the search path is not searched (just as with absolute filenames), and the '`.`' has the same meaning as described above.

4.13.3 Changing the Search Path

The search path is initialized when you start Gforth (see Chapter 3 [Invoking Gforth], page 11). You can display it with

`.fpath` – gforth ".fpath"

displays the contents of the forth search patch

You can change it later with the following words:

`fpath+` "*dir*" gforth "fpath+"

adds a directory to the forth search path

`fpath=` "*dir1*|*dir2*|*dir3*" gforth "fpath="

makes complete new searchpath, serpeator is `|`

Using fpath and require would look like:

```
fpath= /usr/lib/forth/|./

require timer.fs
```

If you have the need to look for a file in the Forth search path, you could use this Gforth feature in your application.

`open-fpath-file` *adr len – fd adr1 len2 0 | ior* gforth "open-fpath-file"

looks in the forth search path for the file specified by adr len when found the resulting path and an open file descriptor is returned. If the file is not found ior is non zero

4.13.4 General Search Paths

Your application may need to search files in sevaral directories, like `included` does. For this purpose you can define and use your own search paths. Create a search path like this:

```
Make a buffer for the path:
create mypath   100 chars ,     \ maximum length (is checked)
                0 ,             \ real len
                100 chars allot \ space for path
```

You have the same functions for the forth search path in a generic version for different paths.

`path+` *path-addr* "*dir*" – gforth "path+"

adds a directory to the search path path-addr

`path=` *path-addr* "*dir1*|*dir2*|*dir3*" gforth "path="

makes complete new searchpath, seperator is `|`

`.path` *path-addr* – gforth ".path"

displays the contents of the search path path-addr

`open-path-file` *adr len path-addr – fd adr1 len2 0 | ior* gforth "open-path-file"

looks in path path-addr for the file specified by adr len when found the resulting path and an open file descriptor is returned. If the file is not found ior is non zero

4.14 Blocks

This chapter describes how to use block files within Gforth.

Block files are traditionally means of data and source storage in Forth. They have been very important in resource-starved computers without OS in the past. Gforth doesn't encourage to use blocks as source, and provides blocks only for backward compatibility. The ANS standard requires blocks to be available when files are.

`open-blocks` *addr u* – gforth "open-blocks"

use the file, whose name is given by *addr u*, as blocks file

`use` "*file*" – gforth "use"

use *file* as blocks file

`get-block-fid` – *fid* gforth "get-block-fid"

`block-position` *u* – block "block-position"

positions the block file to the start of block u

`update` – block "update"

`save-buffer` *buffer* – gforth "save-buffer"

`empty-buffer` *buffer* – gforth "empty-buffer"

`flush` – block "flush"

`get-buffer` *n* – *a-addr* gforth "get-buffer"

`block` unknown "block"

`buffer` *u* – *a-addr* block "buffer"

`updated?` *n* – *f* gforth "updated?"

`list` *u* – block "list"

`load` *i*x n* – *j*x* block "load"

`thru` *i*x n1 n2* – *j*x* block "thru"

`+load` *i*x n* – *j*x* block "+load"

`+thru` *i*x n1 n2* – *j*x* block "+thru"

`-->` – block "–>"

`block-included` *addr u* – gforth "block-included"

4.15 Other I/O

4.16 Programming Tools

4.16.1 Debugging

The simple debugging aids provided in '`debugs.fs`' are meant to support a different style of debugging than the tracing/stepping debuggers used in languages with long turn-around times.

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 word `~~` is easy to insert. It just prints debugging information (by default the source location and the stack contents). It is also easy to remove (`C-x ~` in the Emacs Forth mode to query-replace them with nothing). The deferred words `printdebugdata` and `printdebugline` 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).

Note that the default actions clobber the contents of the pictured numeric output string, so you should not use `~~`, e.g., between `<#` and `#>`.

`~~` *compilation* – ; *run-time* – gforth “tilde-tilde”

`printdebugdata` – gforth “printdebugdata”

`printdebugline` *addr* – gforth “printdebugline”

4.16.2 Assertions

It is a good idea to make your programs self-checking, in particular, if you use an assumption (e.g., that a certain field of a data structure is never zero) that may become wrong during maintenance. 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

`Assert(` is the same as `assert1(`. 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.

Note that the `assert-level` is evaluated at compile-time, not at run-time. I.e., 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 several pieces of code at several `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).

4.16.3 Singlestep Debugger

When a new word is created there's often the need to check whether it behaves correctly or not. You can do this by typing `dbg badword`. This might look like:

```
: 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:

<*return*> 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 Stopp; Abort immediately.

Debugging large application with this mechanism is very difficult, because you have to nest very deep 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.

4.17 Assembler and Code Words

Gforth provides some words for defining primitives (words written in machine code), and for defining the 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,`.

`assembler` – tools-ext "assembler"

`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.

These words are rarely used. Therefore they reside in `code.fs`, which is usually not loaded (except `flush-icache`, which is always present). You can load them with `require code.fs`.

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.

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: adding the wanted functionality to the source of Gforth. For normal words you just have to edit '`primitives`' (see Section 11.3.1 [Automatic Generation], page 100), defining words (equivalent to `;CODE` words, for fast defined words) may require changes in '`engine.c`', '`kernal.fs`', '`prims2x.fs`', and possibly '`cross.fs`'.

4.18 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.

`>code-address` *xt – c-addr* gforth "to-code-address"

c_addr is the code address of the word xt

`>does-code` *xt – a-addr* gforth "to-does-code"

If xt ist the execution token of a defining-word-defined word, a_addr is the start of the Forth code after the DOES>; Otherwise a_addr is 0.

`code-address!` *c-addr xt –* gforth "code-address-store"

Creates a code field with code address c_addr at xt

`does-code!` *a-addr xt* – gforth "does-code-store"

creates a code field at xt for a defining-word-defined word; a_addr is the start of the Forth code after DOES>

`does-handler!` *a-addr* – gforth "does-handler-store"

creates a DOES>-handler at address a_addr. a_addr usually points just behind a DOES>.

`/does-handler` – *n* gforth "slash-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`

You can recognize words defined by a `CREATE...DOES>` word with `>DOES-CODE`. If the word was defined in that way, the value returned is different from 0 and identifies the `DOES>` used by the defining word.

5 Tools

See also Chapter 9 [Emacs and Gforth], page 91.

5.1 '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 :
(
```

5.1.1 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.

6 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 `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.

6.1 The Core Words

6.1.1 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).

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 charcter can be used in a name. Matching is case-insensitive (except in `TABLE`s). The matching is performed using the C 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. `PARSE-WORD` 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 ports, 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:
: 31

maximum string length for `ENVIRONMENT?`*, in characters:*
: 31

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 ports.

number representation and arithmetic:
: Processor-dependent. Binary two's complement on all current ports.

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 ports.

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), although a `-10 throw` (divide by zero) would be more appropriate.

whether the current definition can be found after `DOES>`*:*
: No.

6.1.2 Ambiguous conditions

a name is neither a word nor a number:
: `-13 throw` (Undefined word). Actually, `-13 bounce`, which preserves the data and FP stack, so you don't lose more work than necessary.

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 name 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:
: typically results in a `-55 throw` (floating point unidentified fault), although a `-10 throw` (divide by zero) would be more appropriate.

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 `-9 throw` (Invalid memory address) as soon as the overflow happens. If it is not check, 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:
: Not checked. Runs into the dictionary and destroys it (at least, partially).

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 `-55 throw` (floatingpoint unidentified fault), although a `-10 throw` (divide by zero) would be more appropriate. `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. The consequences of stack underflows are similar to the consequences of stack overflows. 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).

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.

6.1.3 Other system documentation

nonstandard words using `PAD`*:*
None.

operator's terminal facilities available:
: After processing the 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 Chapter 3 [Invoking Gforth], page 11) 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 Chapter 3 [Invoking Gforth], page 11).

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 Chapter 3 [Invoking Gforth], page 11).

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.

6.2 The optional Block word set

6.2.1 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.

6.2.2 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.

6.2.3 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.

6.3 The optional Double Number word set

6.3.1 Ambiguous conditions

d outside of range of n in `D>S`*:*
: The least significant cell of *d* is produced.

6.4 The optional Exception word set

6.4.1 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.

6.5 The optional Facility word set

6.5.1 Implementation Defined Options

encoding of keyboard events (`EKEY`*):*
: Not yet implemented.

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.

6.5.2 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.

6.6 The optional File-Access word set

6.6.1 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.

6.6.2 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?)

6.7 The optional Floating-Point word set

6.7.1 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 Chapter 3 [Invoking Gforth], page 11).

width of floating-point stack:
: `1 floats`.

6.7.2 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 `-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:
: `-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`*):*
: `-55 throw` (Floating-point unidentified fault)

float=<-*1 (*`FLNP1`*):*
: `-55 throw` (Floating-point unidentified fault). On IEEE-FP systems negative infinity is typically produced for *float*=-1.

float=<*0 (*`FLN`, `FLOG`*):*
: `-55 throw` (Floating-point unidentified fault). On IEEE-FP systems negative infinity is typically produced for *float*=0.

float<*0 (*`FASINH`, `FSQRT`*):*
: `-55 throw` (Floating-point unidentified fault). `fasinh` produces values for these inputs on my Linux box (Bug in the C library?)

|*float*|>*1 (*`FACOS`, `FASIN`, `FATANH`*):*
: `-55 throw` (Floating-point unidentified fault).

integer part of float cannot be represented by d in `F>D`*:*
: `-55 throw` (Floating-point unidentified fault).

string larger than pictured numeric output area (`f.`, `fe.`, `fs.`*):*
: This does not happen.

6.8 The optional Locals word set

6.8.1 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.

6.8.2 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)

6.9 The optional Memory-Allocation word set

6.9.1 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*.

6.10 The optional Programming-Tools word set

6.10.1 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 intermediate code used by the inner interpreter. The current `see` tries to output Forth source code as well as possible.

6.10.2 Ambiguous conditions

deleting the compilation wordlist (`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).

6.11 The optional Search-Order word set

6.11.1 Implementation Defined Options

maximum number of word lists in search order:
: `s" wordlists" environment? drop` .. Currently 16.

minimum search order:
: `root root`.

6.11.2 Ambiguous conditions

changing the compilation wordlist (during compilation):
: The word is entered into the wordlist that was the compilation wordlist 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 wordlist.

search order empty (`previous`):
: `abort" Vocstack empty"`.

too many word lists in search order (`also`):
: `abort" Vocstack full"`.

7 Model

This chapter has yet to be written. It will contain information, on which internal structures you can rely.

8 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 ATLASTs 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

9 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 (but still not perfect) handling of indentation. I have also added comment paragraph filling (*M-q*), commenting (*C-x *) and uncommenting (*C-u C-x *) regions and removing debugging tracers (*C-x ~*, see Section 4.16.1 [Debugging], page 68). I left the stuff I do not use alone, even though some of it only makes sense for TILE. To get a 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).

Also, if you `include` '`etags.fs`', a new '`TAGS`' file (see section "Tags Tables" in *Emacs Manual*) will be produced 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 all these benefits, add the following lines to your '`.emacs`' file:

```
(autoload 'forth-mode "gforth.el")
(setq auto-mode-alist (cons '("\\.fs\\'" . forth-mode) auto-mode-alist))
```

10 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.

10.1 Image Licensing Issues

An image created with `gforthmi` (see Section 10.5.1 [gforthmi], page 94) or `savesystem` (see Section 10.3 [Non-Relocatable Image Files], page 93) 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 [GNU General Public License (Section 3)], page 1.

If you create an image with `cross` (see Section 10.5.2 [cross.fs], page 94), 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.

10.2 Image File Background

Our Forth system consists not only of primitives, 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 primitives and the Forth source alone. Therefore we provide the Forth code as an image file in nearly executable form. At the start of the system 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 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 Bradleys `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, our loader performs relocation at image load time. The loader also has to replace tokens standing for 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). And 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.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.

10.3 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"

10.4 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 10.5 [Fully Relocatable Image Files], page 94).

[1] In my opinion, though, you should think thrice before using a doubly-linked list (whatever implementation).

10.5 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:

10.5.1 '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`' 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 nonrelocatable 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 the following one:

```
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).

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 nonrelocatable 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).

10.5.2 'cross.fs'

You can also use `cross`, a batch compiler that accepts a Forth-like programming language. This `cross` language has to be documented yet.

`cross` also 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.

10.6 Stack and Dictionary Sizes

If you invoke Gforth with a command line flag for the size (see Chapter 3 [Invoking Gforth], page 11), 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).

10.7 Running Image Files

You can invoke Gforth with an image file *image* instead of the default '`gforth.fi`' with the `-i` flag (see Chapter 3 [Invoking Gforth], page 11):

```
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*.

`#!` – gforth "hash-bang"

an alias for `\`

10.8 Modifying the Startup Sequence

You can add your own initialization to the startup sequence through the deferred word

`'cold` – gforth "tick-cold"

`'cold` is invoked just before the image-specific command line processing (by default, 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 conventient access to `argv`.

`argc` – *addr* gforth "argc"

contains the number of command-line arguments (including the command name)

`argv` – *addr* gforth “argv”

contains 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”

returns the string for the *n*th command-line argument.

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.

11 Engine

Reading this section 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 papers *ANS fig/GNU/??? Forth* (in German) by Bernd Paysan, presented at the Forth-Tagung '93 and *A Portable Forth Engine* by M. Anton Ertl, presented at EuroForth '93; the latter is available at `http://www.complang.tuwien.ac.at/papers/ertl93.ps.Z`.

11.1 Portability

One of the main goals of the effort 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[1]. GNU C is available for free 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

[1] Unfortunately, long longs are not implemented properly on all machines (e.g., on alpha-osf1, long longs are only 64 bits, the same size as longs (and pointers), but they should be twice as long according to see section "Double-Word Integers" in *GNU C Manual*). So, we had to implement doubles in C after all. Still, on most machines we can use long longs and achieve better performance than with the emulation package.

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.

11.2 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).

11.2.1 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 the loading part (`NEXT_P1`) and the goto part (`NEXT_P2`). `+` now looks like:

```
n=sp[0]+sp[1];
sp++;
```

```
NEXT_P1;
sp[0]=n;
NEXT_P2;
```

This can be scheduled optimally by the compiler.

This division can be turned off with the switch `-DCISC_NEXT`. This switch is on by default on machines that do not profit from scheduling (e.g., the 80386), in order to preserve registers.

11.2.2 Direct or Indirect Threaded?

Both! After packaging the nasty details in macro definitions we realized that we could switch between direct and indirect threading by simply setting a compilation flag (`-DDIRECT_THREADED`) and defining a few machine-specific macros for the direct-threading case. On the Forth level we also offer access words that hide the differences between the threading methods (see Section 4.18 [Threading Words], page 71).

Indirect threading is implemented completely machine-independently. Direct threading needs routines for creating jumps to the executable code (e.g. to docol or dodoes). These routines are inherently machine-dependent, but they do not amount to many source lines. I.e., even porting direct threading to a new machine is a small effort.

The default threading method is machine-dependent. You can enforce a specific threading method when building Gforth with the configuration flag `--enable-direct-threaded` or `--enable-indirect-threaded`. Note that direct threading is not supported on all machines.

11.2.3 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. 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 for the indirect threaded version and for direct threading on some machines. Leaving a cell unused in most words is a bit wasteful, but on the machines we are targeting this is hardly a problem. The other reason for having a code field size of two cells is to avoid having different image files for direct and indirect threaded systems (direct threaded systems require two-cell code fields on many machines).

The other approach is that the code field points or jumps to the cell after `DOES`. In this variant there is a jump to `dodoes` at this address (the DOES-handler). `dodoes` can then get the DOES-code address by computing the code address, i.e., the address of the jump to dodoes, and add the length of that jump field. A variant of this is to have a call to `dodoes` after the `DOES>`; then the return address (which can be found in the return register

on RISCs) is the DOES-code address. Since the two cells available in the code field are used up by the jump to the code address in direct threading on many architectures, we use this approach for direct threading on these architectures. We did not want to add another cell to the code field.

11.3 Primitives

11.3.1 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`'), 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[2]: 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 */
{
DEF_CA                          /* definition of variable ca (indirect threading) */
Cell n1;                        /* definitions of variables */
Cell n2;
Cell n;
n1 = (Cell) sp[1];              /* input */
n2 = (Cell) TOS;
sp += 1;                        /* stack adjustment */
NAME("+")                       /* debugging output (with -DDEBUG) */
{
n = n1+n2;                      /* C code taken from the source */
```

[2] 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.

```
}
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`).

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.

11.3.2 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.

11.3.3 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`'.

11.4 Performance

On RISCs the Gforth engine is very close to optimal; i.e., it is usually impossible to write a significantly faster 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`.

However, this potential advantage of assembly language implementations is not necessarily realized in complete Forth systems: We compared Gforth (direct threaded, compiled with `gcc-2.6.3` and `-DFORCE_REG`) with Win32Forth 1.2093, 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      Win32-    NT       eforth       This-
  time  Gforth Forth Forth eforth  +opt   PFE Forth  TILE
sieve     1.00  1.39  1.14   1.39  0.85  1.58  3.18  8.58
```

```
bubble    1.00  1.31  1.41   1.48  0.88  1.50        3.88
matmul    1.00  1.47  1.35   1.46  0.74  1.58        4.09
fib       1.00  1.52  1.34   1.22  0.86  1.74  2.99  4.30
```

You may find the good performance of Gforth compared with the systems written in assembly language quite surprising. One important reason for the disappointing performance of these 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 10.2 [Image File Background], page 92).

Only Eforth with the peephole optimizer performs comparable to Gforth. The speedups achieved with peephole optimization of threaded code are quite remarkable. Adding a peephole optimizer to Gforth should cause similar speedups.

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.

Note that 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 1.3 times slower engine (on a 486DX2/66 running the Sieve) than the one measured above.

Note also that there have been several releases of Win32Forth since the release presented here, so the results presented here may have little predictive value for the performance of Win32Forth today.

In *Translating Forth to Efficient C* by M. Anton Ertl and Martin Maierhofer (presented at EuroForth '95), an indirect threaded version of Gforth is compared with Win32Forth, NT Forth, PFE, and ThisForth; that version of Gforth is 2%–8% slower on a 486 than the direct threaded version used here. The paper available at `http://www.complang.tuwien.ac.at/papers/ertl&maierhofer95.ps.gz`; it also contains numbers for some native code systems. 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`'.

12 Binding to System Library

13 Cross Compiler

Cross Compiler

13.1 Using the Cross Compiler

13.2 How the Cross Compiler Works

14 Bugs

Known bugs are described in the file BUGS in the Gforth distribution.

If you find a bug, please send a bug report to `bug-gforth@gnu.ai.mit.edu`. A bug report should describe the Gforth version used (it is announced at the start of an interactive Gforth session), the machine and operating system (on Unix systems you can use `uname -a` to produce this information), the installation options (send the '`config.status`' file), and a complete list of changes you (or your installer) have made to the Gforth sources (if any); it should contain a program (or a sequence of keyboard commands) that reproduces the bug and a description of what you think constitutes the buggy behaviour.

For a thorough guide on reporting bugs read section "How to Report Bugs" in *GNU C Manual*.

15 Authors and Ancestors of Gforth

15.1 Authors and Contributors

The Gforth project was started in mid-1992 by Bernd Paysan and Anton Ertl. The third major author was Jens Wilke. 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, and Jorge Acerada. 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 have not met physically for the first 4 years of development.

15.2 Pedigree

Gforth descends from bigFORTH (1993) and fig-Forth. Gforth and PFE (by Dirk Zoller) will cross-fertilize each other. 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 descends from F83. It 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.

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* 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.

Word Index

This index is as incomplete as the manual. Each word is listed with stack effect and wordset.

~

+

>

<

2

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

R

S

T

U

V

W

X

Concept and Word Index

This index is as incomplete as the manual. Not all entries listed are present verbatim in the text. Only the names are listed for the words here.

;

?

@

[

"

~

+

>

\

<

2

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

R

S

T

U

V

W

X

Z

Table of Contents