/im/tt/cam/fire/12x8/fire.0-16a.ps

Contents

Contents

1 Introduction

STEP is an indefinitely scalable multiprocessor architecture optimized for the fine-grained simulations of physical systems and other spatially-extended systems. From a functional viewpoint, the STEP architecture can be thought of as a spatially-extended, uniform, fine-grained medium of a rather general and abstract nature. Specific behavior is obtained by specifying the medium's connectivity and dynamical properties, and assigning its initial state. In this sense, STEP embodies the metaphor of "programmable matter."

This manual describes the software environment provided with the STEP machine. At the moment, only rather low-level software resources are available; however, they are sufficient to exercise and evaluate all of the architecture's features. Thus, the present sofware should be viewed as an "evaluation kit" rather than a mature, comprehensive package.

It is hoped that higher-level software tools, both general-purpose and application-specific, will arise out of the collaboration between the architecture designers and a variety of users.

The present *Software Reference* is one of three companion volumes. The scope of the three manuals is as follows.

- *Architecture Reference:* Describes the STEP architecture, stressing structural and methodological aspects.
- *Software Reference:* Presents and documents the current STEP software, primarily addressing the needs of the applications programmer.
- *Hardware Reference:* Documents the four levels of the STEP hardware hierarchy, namely, *machine*, *box*, *module*, and STEP*chip*, specifying in detail the physical interface between adjacent levels.

All three manuals are undergoing continual revision and expansion. Their PostScript sources, as well as other documentation, are available via anonymous `ftp`. The latest version number of each manual is listed in the `README` file in the same `ftp` directory. Please compare this number with the version number of your copy, written right under the title on the cover page, to know whether your documentation is up-to-date.

1.1 System overview

The activity of the assembly of modules that make up a STEP **machine** is governed and monitored by a general-purpose computer called the **host**. In the current release, the host is a SUN SPARCstation running under UNIX; all references to the 'host' in this manual will imply such a configuration.

All exchanges between the host and the STEP machine are mediated by the STEP **controller** (Fig. ??), which communicates with the host directly via the **SBus**—the SPARCstation's high-performance external bus—and drives the modules via the **CBus**—a tree-like bus having the controller at its root and the individual modules at its leaves. Physically, the controller consists of a small board plugged into one of the SBus slots.

In what follows, the STEP controller and the STEP machine will simply be called 'the controller' and 'the machine'.

Physically, a STEP machine consists of three-dimensional array of **modules**. The array can have an arbitrary extension along each of the dimensions. Typically, each dimension is wrapped around, yielding for the module-array the topology of a three-dimensional torus. A module contains all the resources (state variables and updating machinery) needed to simulate a certain portion (a **sector**) of the system under study. The modules all work in parallel and in lockstep, directly exchanging information only with contiguous modules.

The system being modeled is represented by a collection of discrete **cells** arranged in a regular **lattice**. This lattice may have more or fewer than three dimensions, it may have an arbitrary crystallographic structure (in particular, it need not be a cubic lattice), and may or may not be wrapped around (independently of whether the module array is wrapped around).

An equal amount of data storage capacity is allocated to every lattice cell, for representing in a distributed way the system's state variables, its parameters, and other information useful for processing and analyzing the system's state. Depending on the nature of the model, the appropriate cell capacity may range from a few bits to thousands of bits.

Cells within each sector are updated serially: on each updating pass through the sector a single processor is time-shared between the cells in such a way as to simulate the parallel operation of an equivalent number of identical virtual processors. A complete updating **step**, in which all cell state components are appropriately updated, may require several **passes**. Since the results of each pass are, in ordinary circumstances, independent of the order in which the cells are scanned, the scanning order itself can be chosen so as to accomodate particular display or analysis needs. Moreover, the updating algorithm may change from step to step, so as to implement a *time-dependent* dynamics.

Data communication between cells during each updating pass is not restricted to the immediate neighbors in the lattice. Indeed, one of the features that distinguish the STEP architecture from more "obvious" cellular automata architectures is that essentially instantaneous communication can be achieved between cells that are hundreds or thousands of lattice sites apart. Thus, short- and long-range interactions are efficiently handled at the same time by the updating machinery.

Pipelining of address arithmetic, read, update, and write operations, together with flash updating based on lookup tables, yield a high rate of serial updating: for 16-bit cells, 25 Megacells/sec per module in the current implementation.

The basic building block of a STEP machine is an assembly of eight modules called a **box**. Besides mechanical support, shielding, and power distribution, the box provides signal buffering, clock distribution, CBus routing and fan-out, a frame buffer, and a video interface. All of these functions are handled by a board called the **box interface** (a motherboard with eight slots for the modules can be treated, for electrical purpose, as part of the box interface). Boxes are physically stacked in a three-dimensional array, and exchange information only with adjacent boxes.

The smallest viable STEP machine consists of a single box populated by a single module. Conceptually, there is no upper limit to machine size; in practice, a limit is set by economic factors sooner than by engineering constraints. Installations consisting of hundreds or thousands of modules are feasible and, in terms of useful modeling applications, desirable.

Within each module, ordinary dynamic RAM chips (DRAM) are used to store cell data. The sytem's dynamics is defined by lookup tables, which are stored in ordinary static RAM chips (SRAM). The "glue" that turns these generic commodities into a powerful and flexible modeling environment is represented by a custom VLSI chip—called the STEP**chip**—of which there are a set of sixteeen in each module. This set works as a tightly integrated team, and is indeed called a STEPchip **team**.

A faithful and efficient software simulator for the STEP architecture is available. The current version runs on a SPARCstation, but is easily portable to other platforms. For many applications, a viable approach is to develop an experiment on the software simulator—on a lattice of only tens of thousands of sites—and then run the *identical* experiment on a lattice of millions or billions of sites on a STEP machine.

1.2 STEP's machine language

The STEP architecture gives one great flexibility in specifying the features of the conceptual model that is the object of the simulation. Subject to a single quantitative constraint, i.e., the number of STEP modules available, the user has great latitude in choosing the dimensionality of the lattice, its crystallographic group, and its extension in each of the dimensions; the size and the internal structure of the cells, as well as the interconnectivity of the cells within the lattice; the sequencing of updating steps and the algorithmic details of each step; and, finally, the strategies for real-time data analysis and display and the related input/output tactics.

It must be understood that this great variety of high-level features are synthesized from a basic set of low-level, native features of the STEP architecture—a simple, well-structured, highly orthogonal set of machine primitives.

The main goal of this manual is to provide a "programmer's model" of STEP appropriate to this synthesis taks. That is, our chief concern will be to introduce, explain, and document those aspects of the hardware that are meant to be visible to the programmer. In this respect, we shall deal with primitive concepts such as *sites* and *layers*, *kicks* and *glue*, *scans*, *lookup tables* and *events*. Common STEP-programming constructs such as *lattice*, *cells*, and *steps* will be mentioned, but they properly belong to the application-language level rather than machine-language level.

Ultimately, STEP programming is reduced to writing to (and occasionally reading from) a set of hardware registers. Only about a dozen of these registers are of direct concern to the user, when programming at the machine-language level. Even if programming is done through a compiler, the user should be generally aware of the existence and function of these registers. The other registers—used for setup, interrupt control, and diagnostics—are handled by system calls and are of concern only to systems programmers.

1.3 Organization of this manual

[to be written]

1.4 Conventions

Here we list a number of general conventions followed throughout this manual.

1.4.1 Emphasis and quoting

Italics are used for *emphasis*, i.e., to stress a particular word or phrase. Boldface indicates an **inline definition**, i.e., a term that is formally being assigned a specific technical meaning by the surrounding text. Double quotes are used, as customary, for quoting short pieces of text inline. Double quotes are also used to indicate that an expression is deliberately used in some informal, allegorical, or humorous way, as when one says that "the computer got confused". We follow the continental style of putting *outside* the quotes punctuation that belongs to the main text rather than the quoted text (such as the previous period). Finally, single quotes are used for literally exhibiting a string, as when we say that 'one word' is *two* words.

1.4.2 Number representation

Numerals in typewriter font are in base sixteen; thus `40`=64 and `ff`=255. The advantages of using base sixteen in many hardware and software technical discussions need not be belabored.

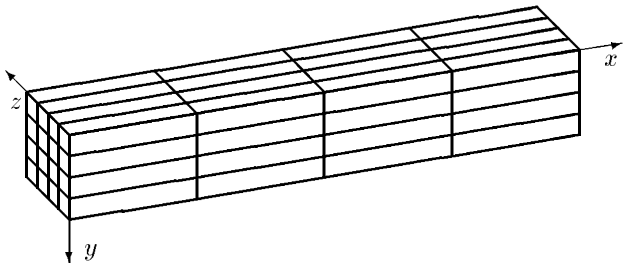
1.4.3 Data and variable ordering

In listings, tables, and figures, we had to fight a hard struggle to reconcile conflicting conventions concerning the ordering of items and the orientation of axes.

There are two basic sources of conflict. One conflict is between the "most significant first" approach, typified by the way we write a number such as '1992' starting with the thousands and ending with the units, and the "first causes first" approach, typified by the way the electronic engineer draws the stages of digital counter, starting with the units stage on the left and making the carry propagate left-to-right. The other conflict is between the way a mathematician would write an i, j array, with j increasing left-to-rigth and i top-to-bottom, in agreement with western-style writing, and the way a physicist would draw an x, y or an x, t diagram on the page, with x pointing rightwards but y or t pointing *upwards*. Note that the "mathematician's" approach is also the one used in scanning an ordinary CRT monitor—left-to-right and then top-to-bottom.

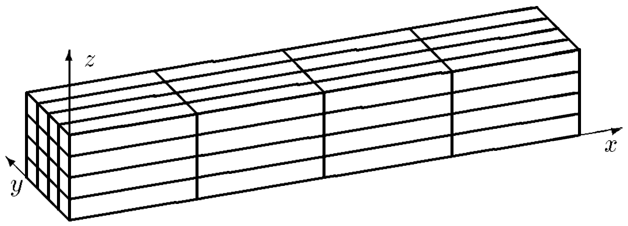
After much soul-searching, we took the responsibility to enforce a consistent, universal convention: in any enumeration, the natural ordering of the items (i.e., increasing values of the enumeration index) will be reflected by the typographical ordering (left-to-right, and, for a second index, top-to-bottom).

Thus, an array indexed by x and y will be displayed with x increasing from left to right and y from top to bottom, as in the following diagram



Here, the diagram's third axis, z, is represented as pointing *into* the page).

Note that, with these conventions, the "handedness" of the (x, y, z) frame is the same as that that results from the more conventional ordering of axes, illustrated by the following diagram



Similarly, a 16-bit computer register will invariably be represented as follows

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

and a text description of this register will list individual bits or fields in same order, as in the following example:

0–6	ASCII character
7	Parity bit
8–15	Not used

When the contents of the register is interpreted as a binary-encoded integer by the computer's arithmetic-logic unit or its address-arithmetic unit, bit 0 will represent the least significant binary digit (units), and in general bit n will represent the coefficient of 2^n.

The above ordering convention will be broken in *only one circumstance*. In deference to universally ingrained practice, ordinary numerals in the text will be written as usual with the most significant digit first. Thus, the number represented by the following register contents

1	0	0	0	1	1	1	1	0	0	0	0	0	0	0	0
0															15

will be written, in text form, as hexadecimal `00f1` or decimal 237.

4 STEP device driver

All interaction between a program running on the host under UNIX and the controller is mediated by an *installable device driver*[1] called the STEP *driver*, which acts as a custom extension of the UNIX kernel.

UNIX treats devices as a special files, to which certain special-purpose operations (`ioctl`) are applicable in addition to the standard open, close, read, and write calls. Devices appear in the `/dev` directory; the STEP device driver is listed as `/dev/step`.

NOTE: In its current version, the STEP driver is called `/dev/cam`, and in its nomenclature uses the string `cam` rather than `step` throughout.

[This appendix to be greatly expanded.]

4.1 Using the driver

In order for the driver's entry points to be accessible by a user program, the `fcntl.h` and `errno.h` system header files as well as the custom header file `camio.h` must be included. The latter declares the entry points to the driver, defines the structure `sl_element` of a step list instruction, and also defines the arguments for the different `ioctl` calls permitted on the driver.

The `cam` device is opened with the standard `open` call. The cam device must be opened for read/write access (`O_RDWR`) in order to function correctly. Similarly, `cam` is closed with the standard `close` call. All meaningful interactions with the device are done through `ioctl` calls. The standard `read` and `write` calls are not used; though entry points for these calls are provided, they result simply in an error being returned, with no effect on the state of the STEP controller or of the driver itself.

The macros for the `ioctl` calls, defined in the `camio.h` header file, are described in the following section; subsection headings are identical to the corresponding call names. (Eventually, both call names and subsection headings should be updated to reflect the actual register names.)

4.2 Driver entry points

4.2.1 CIOSTEP

This call is used to schedule a step list. Basically, the call receives a pointer to the beginning of a step list, ideally in both the user and kernel address spaces. The call prepends a no-op instruction with a soft interrupt to the beginning of the step list, sets the Next List Pointer on the controller, and then waits for the hardware to perform the interrupt. The call returns as soon as the soft interrupt is received. (This is the typical behavior expected from a read/write device driver call.)

4.2.2 CIOSTOP

This call is essentially the scheduling of a stop step list. The stop call generates a no-op instruction with a soft interrupt and a wait for host. The host next pointer on the interface is set appropriately, and the call waits for the hardware to perform the interrupt. The call returns as soon as the soft interrupt is received.

4.2.3 RESET-INTF

This call is used to reset the controller. The associated bit of `cam_rstintr` automatically clears itself when the controller reset sequence is complete.

4.2.4 RESET-CAM

This call is used to reset the STEP hardware. Unlike the controller reset, this reset is not self-clearingd; the associated bit of `cam_rstintr` needs to be explicitly cleared. This bit should not be cleared until the driver has waited a sufficient time for the STEP hardware to have completed the reset. (For the actual timing of the STEP reset sequence, consult Tom D.)

4.2.5 SPARC-2?

Returns the type of SPARCstation in which the board is installed. This call exists so that the hardware may take advantage of the ability of newer SPARCstations to use 16-word bus transfers in addition to 4-word transfers. (Actually, it may not be necessary to make this a call, as it is probably not necessary for the higher-level software to be concerned with this distinction. Basically, the purpose of this is to permit 16-word transfers on the SPARC-2 and limit transfers to 4-words on the SPARC-1.)

4.2.6 Get-status-interrupt

Senses the interrupt status of the device. It returns a `u_int` which encodes the interrupt status. This is the mechanism for passing interrupt information to the user program, since the `cam_intr` routine, being a "bottom

[1] *Writing SBus Device Drivers*, SUN Microsystems (...).

half" routine[], cannot directly pass information to the user program.

4.2.7 `Clear-status-interrupt`

Clears the interrupt status of the device as seen by a user program.

4.2.8 `MEM`

This kernel call is supposed to allocate memory for sharing between the user program and the device. This needs to allocate memory that is mapped in the kernel context (as well as the user context.)

4.2.9 `GET-HNP`

Returns the value of the Next List Pointer.

4.2.10 `GET-CURR-INST`

Returns the value of the Current Instruction Pointer..

4.2.11 `GET-PREV-INST`

Returns the value of the Previous Instruction Pointer.

5 A Forth tutorial

The main purpose of this tutorial is to give you an overall *reading* familiarity with Forth—enough to follow the STEP programming examples given in this book and in the accompanying software.

Relatively little knowledge of Forth is needed to compose full-fledged STEP experiments: the same few constructs appear over and over with minor variations, while many features of the Forth environment that are prominent in other programming contexts are not required at all.

However, these few constructs must be understood well. In many cases an intuitive presentation will be sufficient, but we shall not hesitate to give the appropriate amount of technical detail in those few cases where this is necessary to insure exact comprehension.

To practice this tutorial, you have two choices. You can work with the bare Forth system (`/usr/bin/forth`), or you can run **StepForth** (`/usr/bin/step`)—the STEP machine *monitor*—which is an extension of Forth and provides a complete programming environment for the STEP machine.

By 'Forth' in this chapter we mean the particular Forth dialect on which StepForth is built. We'll explicitly say 'StepForth' when we want to make clear that a certain feature belongs to the STEP extension of Forth rather that to the underlying language.

Historically, Forth started out with 16-bit data **cells**, corresponding to a 16-bit memory word. The present Forth uses 32-bit cells; all our examples will be based on cells of this size. (There is no connection between Forth cells and STEP cells; a Forth cell is a unit of storage of the size of a "machine word" of the Forth processor (which resides in the host), while a STEP cell is an assembly of data bits that are logically associated with a given spatial site of the STEP machine.)

5.1 The command interpreter

You may visualize the Forth *command interpreter* as a competent but not-too-literate technician who sits in the machine room of your computer and has access to all the levers and dials. From the deck, you speak to him through an "intercom"—i.e., your terminal—issuing orders and receiving reports and acknowledgements (see §?? and §?? for how to summon the command interpreter). For instance, if you say

```
beep
```

(type it at the terminal, followed by a carriage return) the terminal will respond with a "beep"; if you say

```
0 100 dump
```

the contents of the first 100 memory locations (starting from location 0) will be dumped on the screen. After that, the interpreter will say

```
ok
```

to tell you it's done and ready for a new command. (From now on, we'll take this carriage-return and `ok` business for granted.)

Saying `bye` dismisses the interpreter; you are then automatically returned to the process from which you had gained access to the interpreter itself.

The interpreter's "ears" are conditioned to break up the input character stream into *tokens*, using "blank space" (one or more consecutive spaces) and *only* blank space as the token separator.[1] Even though some Forth tokens consist of a single character (including comma, period, semicolon, etc., which in other computer languages are often used as "punctuation marks") one should be careful not to omit the blank space between them and adjacent tokens. The tokens are passed on one-by-one to the interpreter's "brain" and the spaces are discarded. Thus, the interpreter hears the above command as a sequence of three tokens—`0`, `100`, and `dump`—and would hear the same thing if you typed, say,

```
0 100 dump
```

In order to be understood by the interpreter, the two of you must share a dictionary of terms and a set of grammatical conventions, which together make up the Forth language. The Forth grammar is simple and fixed; on the other hand, the **Forth dictionary**, which reflects the range of things that the interpreter currently knows about, typically grows with time. As you take command, you'll find that the Forth interpreter has already gone through "standard training"—and possibly some additional, more specialized training (for example, how to exchange data with a STEP machine). This standard training, which is documented in any good Forth manual, is more extensive than that of many common computer languages; in this sense, one speaks of Forth as a programming *environment* rather than just a programming *language*.

[1] At the interactive terminal, carriage return acts, of course, as a token terminator. For "canned" text submitted to the interpreter from a disk file, cf. footnote ??.

5.2 The compiler

The entries that make up the Forth dictionary are called, as you might expect, **words**. To *program* in Forth, you successively add new words to the dictionary—defining each new word in terms of existing ones. In this way you extend the interpreter's knowledge—and at the same time your expressive range—with regard to the set of activities you are interested in. At any stage of this construction you may say something that uses the new words, and check that its actual meaning (i.e., what the interpreter does in response to your words) is what *you* had in mind.

For example, to make up a new word for "beep three times" you type

```
: 3beep beep beep beep ;
```

As soon as it sees the colon token (`:`), the interpreter summons the aid of another technician, called the COLON *compiler*. This technician takes the token that immediately follows, in this case `3beep`, and starts a new dictionary entry under that name. After that, it expects a phrase (a sequence of tokens) describing the action of the new word; this phrase, namely `beep beep beep`, is not executed at this time, but is compiled in the dictionary as the meaning of `3beep`. The end of the phrase is marked by a semicolon token (`;`)—don't drop the space before it—which tells the COLON compiler to return control to the command interpreter.

The language understood by the COLON compiler is slightly different (and somewhat richer) than that understood by the command interpreter; some things only work with either the compiler or the interpreter, but not with both. Except when noted below, everything we will discuss works well either way.

If you now type

```
3beep
```

the interpreter will look up this word in the dictionary, execute it, and respond with the usual `ok`. If you type, say,

```
4beep
```

the interpreter will look it up but won't find it in the dictionary; it will then ask an assistant whether it might possibly be a number (see next section); and finally will print

```
4beep ?
```

to tell you it can't make sense of the token `4beep`.

5.3 The dictionary

If you type `forth words`, you'll get a listing of all the words currently contained in the main section of Forth's dictionary,[2] starting with the ones Forth has learned most recently. Thus, if you had just had the above conversation with the interpreter, the word `3beep` would be on top of the list. The older word `beep` would appear somewhere down the list.

The following entries

```
beep here + ' c@ ; : c! 0=
```

taken at random from the dictionary give you an idea of what typical Forth words look like. *Any* token can be entered in the dictionary as a word. In particular, characters that in other languages are used as punctuation marks, such as `:`, can appear as part of a Forth word, or even make up a word all by themselves; the word `c,` is very different than the two-word sequence `c ,`.

A word already present in the dictionary may be redefined by you. For instance, if the speaker in your terminal is dead, as a temporary fix you may use a version of `beep` that prints on the screen, on a new line, the message `This IS a beep!`. This is done by redefining `beep` as follows

```
: beep    cr ." This IS a beep!" ;
```

Type the above line exactly as it appears (there must be at least one space after `beep`, after `cr`, after `."`, and before the `;`). the word `cr` will execute a "carriage return" (moving the terminal's "printing head" to the beginning of a fresh line); the construct `."` ⟨text⟩`"` will print ⟨text⟩ on the screen (this construct can only be used within a COLON definition). If you now define

```
: 4beep beep beep beep beep ;
```

this word will be compiled using the *new* version of `beep`, and when executed will print

```
This IS a beep!
This IS a beep!
This IS a beep!
This IS a beep!
```

The old version of `beep` is not deleted from the dictionary, and the previously defined word `3beep` *will retain its original meaning*—which is tied to the *old* version of `beep`). If you say `3beep` now, the three beeps will still be routed to the (dead) speaker.

It's all right, and often useful, to redefine a word in terms of its previous namesake. For instance, if you discover that beeps in your terminal have such a long decay time that three consecutive beeps sound more like a long one, you may redefine `beep` as

```
: beep beep 10 ticks ;
```

(where `10 ticks` means "Wait for ten ticks of an internal computer clock"), and a sequence of beeps will now sound "staccato" rather than "legato".

In some English dictionaries, words belonging to certain specialized areas of discourse are listed in separate sections (e.g., geographical names, measurement units,

[2] In addition to this main section, called `forth`, the dictionary may contain some additional specialized sections, as will be explained in a moment.

abbreviations). In Forth, these sections are called **vocabularies**; in addition to the main `forth` vocabulary there may be an `assembler` vocabulary containing machine-language op-codes and other assembler-specific terms, an `editor` vocabulary containing editor-specific terms, etc. As explained in §5.18, there are commands available for specifying which vocabularies should be searched at any particular moment, and in what order.

5.4 Numbers

If in an ordinary piece of English text you find a phrase such as "the motion was passed with 371 votes in favor", you won't look up '371' in the dictionary—and for that matter you wouldn't find it there if you did. The meaning of a number derives from its make-up; this makes it possible for any fool to produce more numbers than any one would want to list, but at the same time makes it unnecessary to list the meaning of individual numbers in a dictionary.

The situation is analogous in Forth: numbers are parsed as they are encountered and their meaning is reconstructed from their make-up by another technician, called `number`, summoned by the interpreter as the occasion arises. This "meaning" is nothing but an internal representation in binary form. For instance, if the interpreter sees the token `103`, by default this token will be recognized as a numeral and will be converted to and internally stored as a 32-bit pattern—namely 00...01100100 (this is the binary expression for the number 103). However, if you had told the interpreter to use base sixteen rather than base ten when reading input,[3] the same token `103` will be interpreted as a hexadecimal numeral and stored as the pattern 00...0100000011 (meaning $2^8 + 2^1 + 2^0$, or 259 in decimal).

Some numbers that for historical or practical reasons deserve explicit mention, such as 'three', *are* listed (in a spelled-out form) in the English dictionary. In an analogous way, some common numbers such as 0 and 1 have been entered (in this form, i.e., `0`—not `zero`) as *words* in the Forth dictionary.[4] This leads to more efficient execution (their meaning has been established in advance, once and for all, and there is no need to ask for the help of the `number` technician) and more compact code.

By default, Forth handles numbers as signed integers with a 32-bit range (only 2^{32}=4,294,967,295 different values are available) so that the ordinary counting sequence −2, −1, 0, 1, 2, 3, ... at a certain point "wraps around": ..., 2147483646, 2147483647, −2147483648, −2147483647, ..., −2, −1, 0 (cf. §5.16). We'll not try to convince you that what is most natural for a computer (or for a person that lives with computers) should be the most natural thing for you; indeed, while these features of Forth's "numbers" are actually an asset in low-level programming tasks, serious numerical applications require much more powerful number-handling facilities. The Forth philosophy is that specialized applications should be served by specialized extensions to the language (cf. §??).

5.5 The stack

Forth manages to achieve remarkable expressive power and efficiency; yet a Forth *system* (i.e., the language as implemented on a computer) can be amazingly simple and compact. These advantages are bought at the cost of ruthless standardization, in particular in the way nearby words in a phrase communicate to one another the information that binds them together as a semantic whole.

For the purpose of this communication, all data (characters, Boolean variables, numbers, addresses, etc.) are packaged in a standard-size "carton", called a **cell**, having a capacity of 32 bits, and all data exchanges take place—in a preordained choreography, as we shall see in a moment—through a single clearing-house called the **stack**. This is just a pile of cells that grows or shrinks according to the traffic.

Imagine a stage with this stack in the middle. The Forth interpreter is the ballet conductor; as he reads off the words that make up your phrase, the corresponding actors show up in sequence, do their thing, and disappear. If an actor's part tells him to leave behind certain data for later actors, he'll walk to the stack and pile these data, cell by cell, on top of it; if his part expects data from previous actors, he will walk to the stack and pick them up.

Some data that an actor needs may end up, say, buried two cells deep into the stack. The actor won't go fumbling through the stack looking for them (the cartons don't carry a label!); rather, his score will explicitly tell him to lift just the top two cells, grab the cell that is now on top, and put the first two back down (incidentally, the score notation specifying this sequence of actions is `rot`, discussed in §5.12).

With this scheme, there is no need for each piece of data to have an *absolute* address—a permanent mailbox with a distinguished name. Instead, all addressing is by position *relative* to the top of the stack. If a new, self-contained piece of choreography is inserted in the old score, at the moment of executing it one will find the stack already built up to a certain height; during execution of this piece one will see the stack grow more, shrink a bit,

[3] In Forth, numbers can be read and printed in any base you choose.

[4] Words of type CONSTANT; cf. §5.8.

etc., and by the end of the inserted piece of choreography return to its original height. The rest of the score will then resume, finding its own data where they had been left.

5.6 Expressions

The stack discipline is well suited to the communication needs of a hierarchically built program. It allows one to use a particularly simple scoring notation—called **reverse Polish notation**—by which arithmetical and logic expressions of arbitrary depth can be written without making recourse to parentheses or other place markers.

When you type a number to the command interpreter, this number is packaged in one cell and put on top of the stack. The one-character word '`.`' ("dot") picks up the top cell of the stack and prints its contents *as a number* on the screen. Thus, if you type

```
356 .
```

(where the dot '.' is part of what you type) the screen will respond with

```
356
```

(In a more conventional programming language, the equivalent of '`356 .`' would be something like '`print(356)`'.) Note that the stack went up one level with `356`, down one level with '`.`', and is now the same height as before.

The word '+' ("plus") gobbles up the top *two* cells of the stack, adds them together, and places the result—consisting of *one* cell—on top of the stack; thus, it leaves the stack one level lower than it found it. For example, the expression

```
2 3 +
```

will leave the result 5 on the stack (from where you can move it to the screen with '`.`'). If you want to see how much $1 + 2 + 4$ is, you type

```
1 2 + 4 + .
```

We can picture the evolution of the stack as follows

STACK	INPUT	OUTPUT
...	`1`	
... 1	`2`	
... 1 2	`+`	
... 3	`4`	
... 3 4	`+`	
... 7	`.`	`7`
...		

where each row displays (a) the current state of the stack (with the top element on the right), (b) the text to be interpreted, and (c) what is printed on the screen. The dots on the left indicate the part of the stack that we haven't touched; this indication will be dropped in the following stack examples.

Note that once 3 has been placed on the stack, it does not matter *how* it got there; from a functional viewpoint, the expression `1 2 +` is interchangeable with, say, `3`, or `1 1 + 1 +`, or anything that eventually bows out having left just a 3 on top of whatever else the stack contained before. Note also that the two different expressions

`1 1 + 1 + 1 +` and `1 1 1 1 + + +`

produce the same end result even though the second one temporarily builds up the stack to a higher level:

STACK	INPUT	STACK	INPUT
	`1`		`1`
1	`1`	1	`1`
1 1	`+`	1 1	`1`
2	`1`	1 1 1	`1`
2 1	`+`	1 1 1 1	`+`
3	`1`	1 1 2	`+`
3 1	`+`	1 3	`+`
4		4	

5.7 Editing and loading

Once you have said something to the Forth interpreter, you cannot take it back; if something goes wrong, you may not even remember exactly what you said.[5] While immediate interaction with the interpreter is very useful, there are times where you would like to carefully think out in advance a whole sequence of commands and definitions, review and edit it, and perhaps discuss it with somebody else before you give it to the interpreter. You want to be in a position to give *written* orders, and you might have a collection of different "orders of the day" to be handed to the interpreter according to the circumstances.

You can do all of this by first writing your text on a disk file, where it can be inspected and modified by means of an ordinary text editor such as EMACS. Then you can ask the interpreter to use this file as the input stream, instead of what comes from the keyboard; this process is called **loading**. When you load a file, everything (well, *almost* everything) works as if you were typing the file's contents from the keyboard—except that the interpreter now processes your tokens as they come, without waiting for a carriage return.

When you compose your text with the editor for subsequent loading, you may choose to format it in a way that facilitates comprehension, and here and there add a comment to yourself.

The Forth word `(` removes from the input stream everything that follows, up to and including the matching character ')'; thus, you may write a line as follows

```
beep beep ( two beeps ) beep beep
```

[5]But see `see` in §??.

and the interpreter will never hear what is "in parentheses".[6]

The word '\' ("backslash") treats as a comment the remainder of the line on which it appears

```
: beep            \ new version!
       beep       \ plain beep
       10 ticks ; \ insert delay
```

5.8 "Constants" and "variables"

In §5.2 we said that the interpreter "executes" the words you type. Actually, each word in the Forth dictionary carries a notice saying "I am to be executed by technician so-and-so, who knows how to handle me", and the interpreter will just pass the buck to this technician. For words that have been entered in the dictionary by the COLON compiler, the competent technician is the COLON interpreter. In general, each type of word has its own compiler and a corresponding interpreter. The buck stops with words that have been compiled by the CODE "compiler"; this technician actually produces code written directly in machine language (i.e., your microprocessor's native language), and is more properly termed an *assembler*. At this point the hardware takes over.

All of this works much more simply than it sounds. Suppose you are writing a telescope-driving program that needs to know your town's latitude, say, 43°. It is good programming practice to give this number a *name*—say, `latitude`—so that whenever this name appears in your program it will have the same effect as if you had typed '43'. To do this, in Forth you say

```
43 constant latitude
```

the word `latitude` will be entered in the dictionary as a constant, and when executed it will place the number 43 on the stack.

What happens is that as soon as it sees the word `constant` the command interpreter summons the aid of the CONSTANT compiler, (cf. §5.9) who gobbles up the next token—namely `latitude`—and starts a new dictionary entry under that name. The entry will consist of two parts: the first (**code field**) contains a notice saying "I am to be executed by the CONSTANT interpreter"; the second (**data cell**) is reserved for the value of the constant. At this point the CONSTANT compiler takes the top cell of the stack—with the 43 you had just put there—and moves it to the data cell in the dictionary. At execution time, the CONSTANT interpreter will look at the data cell and place a copy of it on the stack.

With the above definition of `latitude`, the command

```
latitude 7 + .
```

will print `50` on the screen.

The term 'CONSTANT' is somewhat of a misnomer (though it is retained for historical reasons), since the contents of the data cell may be altered at will; in the present implementation of Forth, to change the value of `latitude` to 45° you say

```
45 is latitude
```

The most relevant aspect of a Forth CONSTANT is that it returns the *value* of its data cell, rather than a *pointer* to it (cf. `variable`) below).

In STEP, FIELD words, defined by the defining word `==` or derivatives therof (cf. §??), act like Forth CONSTANTs insofar as they return a value; however, this value will change very many times during the construction of a rule table (§??).

Forth provides another mechanism for accessing a piece of data, namely by its *address*[7] rather than by its *value*. The VARIABLE compiler, used in a construct such as

```
variable time
```

is analogous to the CONSTANT compiler insofar as it constructs a dictionary entry, namely `time`, with a data cell in it. However,

- This data cell is not initialized to a particular value (and therefore the defining word `variable`, unlike `constant`, does *not* expect a value on the stack).
- When `time` is executed, the VARIABLE interpreter puts on the stack the *address* of the data cell, rather than its contents.

Thus, if we type

```
time
```

what will be placed on the stack is not the current value of `time` (which may change several times during execution of the program), but its address (which is always the same).

To get the *value* of `time` you use the word '`@`' (pronounced "fetch"), as in

```
time @ ( data )
```

(i.e., '`@`' expects an address on the stack, and replaces it with the data at that address), and to set it you use the word '`!`' ("store"), as in

```
( data ) time !
```

(i.e., '`!`' expects a piece of data *and* an address, and ships the data to that address). For example, to increment time by one unit you write

[6] Observe that the following spacing is correct too

```
beep beep ( two beeps)beep beep
```

even though there is no intervening space between '`)`' and `beep`, since the effect of the word `(` is precisely to throw away the string '`two beeps)`'. On the other hand, the following spacing

```
beep beep (two beeps) beep beep
```

won't do, since it will make the interpreter think that '`(two`' is a token to be processed.

[7] In an ordinary computer, memory locations are sequentially numbered; the address of a piece of data is the number of its location.

```
time @ 1 + time !
```

Supposing that the `time` data cell is at location 1000 and its initial contents is 5, the evolution of the relevant data is the following

`time`'s contents	STACK	INPUT
5		`time`
5	1000	`@`
5	5	`1`
5	5 1	`+`
5	6	`time`
5	6 1000	`!`
6		

(The word `+!` allows you to use the more efficient construct `1 time +!`, where the cell's value never appears on the stack. Similarly, `time off` will store all `0`s in the cell at the address supplied by `time`, and `time on` will store all `1`s, corresponding to the logic values `false` and `true`—cf. §5.16.)

From the viewpoint of STEP's user, Forth VARIABLEs need seldom be used. In this book, the term 'variable' always has the usual meaning of "a generic quantity to which we may assign an arbitrary value" rather the the more technical Forth meaning, for which we reserve the small-caps term VARIABLE.

5.9 Defining words

The words `:` ('colon'), `constant`, and `variable` belong to the class of **defining words**, whose action is to create new dictionary entries.

In Forth, a number of defining words of common utility are built-in; each one activates a miniature compiler (and, at execution time, a miniature interpreter) for a specific data structure. If an application is going to make regular use of other data structures, it is possible (and quite easy) to introduce new, appropriate defining words. Similar considerations apply to *control constructs*, discussed in the following sections.

In other words, while, for the casual user, programming will consist merely of extending the *lexicon* of the received Forth system, the more sophisticated user will find it both expedient and easy to extend Forth's very *syntax*. Practically every major programming effort can benefit from a moderate investment in "language development" of this kind.

The developers of StepForth have made extensive use of the above features of Forth.

5.10 Iteration

Once you've stored a program in the computer's memory, portions of it can be executed over and over, perhaps with some variations.

For instance, you can define

```
: beep-stuck   begin beep again ;
```

When this word is called, once the execution reaches `again` it jumps back to `begin`, producing an endless series of beeps. The only way to break the loop is through a keyboard interrupt (typically `^C` to abort the loop, or `^Z` to suspend the Forth process).

The above `begin` / `again` pair delimits a phrase somewhat like a pair of parentheses: the phrase in between gets iterated forever. Note that for readability we vertically aligned[**revise**] the two elements of the pair, flush on the *right* (this is recommended in reverse-Polish-notation style), and indented the phrase inside.

A more flexible pair is `do` / `loop`; the word

```
: 100beeps   100 0 do beep loop ;
```

will beep 100 times.[8] What happens in detail is that `do` gobbles up the top two numbers on the stack, 100 (the loop LIMIT) and 0 (the loop INDEX), and saves them for later use. The execution proceeds until `loop` is encountered. At this point, INDEX is incremented by one and compared with LIMIT; if INDEX=LIMIT the loop is terminated; otherwise, execution jumps back to `do`.[9]

Inside a `do` loop, the word `i` returns on the stack the current value of the index. Thus, the word

```
: print-ascii   127 32 do i emit loop ;
```

will produce all printable ASCII characters. (Character 0–31 are control characters; all others are printable except for the last character of the ASCII set, namely character 127, which on old-fashioned paper tape "prints" 7 holes on the last punched character, effectively DELETEing it.)

Pairs of "parentheses" such as `begin` / `again` and `do` / `loop` can be nested as ordinary parentheses, and a lot more bells and whistles are available. You can look up the details of these and other flow-control constructs in a Forth manual. Here we shall only mention that, in traditional Forth, flow-control constructs can only appear inside a COLON definition; that is, you cannot say

```
100 0 do beep loop
```

at the command-interpreter level. In StepForth, flow-control constructs encountered outside a colon definition are colon-compiled "on-the-fly" and then executed as if they were part of a COLON word. However, the material thus compiled is not enclosed in the dictionary and is immediately discarded.

[8] Of course, if you had typed `999beeps` rather than `100beeps` you still would have gotten a word that beeps 100 times. A name is a name is a name…

[9] The loop index is a modulo-2^{32} counter. The minimum number of iterations, namely 1, is achieved when the loop is entered with INDEX just one less than LIMIT; the maximum, when the loop is entered with INDEX *equal* to LIMIT: `0 0 loop` will cycle 2^{32} times!

5.11 Stack comments

By now, we have encountered many words that expect arguments on the stack or leave results on the stack. Since breaches of stack discipline may send a computation berserk (if you leave an extra item on the stack everyone after you will get his data wrong), it will be convenient to have a notation to remind us of just how many items a word takes from the stack or leaves on it.

The following are examples of *stack comments*:

```
do ( n1 n2 -- )
do ( limit index --)
beep ( -- )
latitude ( -- n)
time ( -- addr)
+ ( n1 n2 -- n3)
+ ( m n -- m+n)
2 ( -- n)
2 ( -- 2)
```

The general convention is as follows. We put in parentheses a' double-dash to indicate the word in question. Before the dash we write a list of what the word *expects* on the stack; after the dash we write a list of what the word *leaves* on it. If nothing appears *after* the dash, the dash itself is usually omitted.

What really counts is the *number* of items in each list—which corresponds to the number of cells taken from or given to the stack; but the items themselves may be elaborated upon a little, for extra clarity.

For example, `do` takes two items and leaves none. A minimal notation is

```
DO ( n1 n2 -- )
```

which just tells us that `do` expects two items. A better mnemonic is provided by

```
DO ( limit index --)
```

which reminds us that the first item is used as the *limit* and the second as (the initial value of) the *index* of the loop.

As a final grand example, let us load the following three definitions from a disk file

```
  100 constant hundred  ( -- n)
10000 constant a-lot-of ( -- n)
             : beeps    ( n --)
       0 do
    beep loop ;
```

The words `hundred` and `a-lot-of` are introduced by the defining word `constant`; the word `beeps`, by the defining word `:` ('colon'). The vertical alignment of the three defined words—`hundred`, `a-lot-of`, and `beeps`—is for clarity; so is the right-justified alignment of `do` and `loop` (cf. §5.15).

Now type the following three commands

```
3 beeps
hundred beeps
a-lot-of beeps
```

We have seen above that `do` wants two arguments. When we type `3 beeps`, the first argument is left on the stack by the `3` we typed, while the second is placed on the stack by the `0` appearing within the definition of `beeps`: `do`'s hunger is satisfied.

Thus, if we define words giving a little thought to the **stack interface** (i.e., the conventions about *who* should supply or consume *what* and *when*) and to choosing appropriate names, the flow of a Forth phrase can be given a natural-language flavor that is hard to achieve in other programming languages. Properly trained Forth words can talk to one another under the surface of the phrase, without bothering us with their chatter.

A generally obeyed convention in Forth is to make words "use up" their arguments rather than leave them on the stack. If an object on the stack is needed as an argument by a given word and also by another word that closely follows, a second copy of this object is made—using the word `dup` introduced below—before the first copy is used up.

5.12 `dup`, `drop`, and all that

The Forth word '`*`' ("times") takes two numbers and returns their product; to compute the square of 3 you have to type 3 twice: `3 3 *`. How about a word `sqr` that will take a *single* argument and multiply it by itself?

Forth provides a number of general-purpose words for manipulating the stack; one of these is `dup` (pronounced "dupe"), which looks at the cell on top of the stack and puts a *duplicate* copy of it on top of it; e.g.,

STACK	INPUT
5	`dup`
5 5	

Thus, `sqr` can be simply

```
: sqr ( n -- n*n)   dup * ;
```

since here '`*`' will see two copies of the argument.

Related to `dup` are `drop` (which drops the top item from the stack), `swap` (which swaps the top two items), `over` (which makes a copy of the next-to-the-top stack item), `rot` (which pulls the third item from underneath and puts in on top of the first two), and a few more. Words of this kind act somewhat like *pronouns* in English ('this', 'that', 'one another', etc.), in that they allow one to refer by *position* rather than by *name* to objects introduced in a different part of the sentence. As an exercise, verify that the function $y(m,n) = (m+n)(m-n)$ is computed by the following Forth expression

```
( m n ) over over + rot rot - * ( y )
```

(where the comments in parentheses tell you what's on the stack before and after).

5.13 Case selection

A flow-control construct that is supported and encouraged by StepForth is the FLASH CASE construct, which allows one to select for execution one of several alternative actions.[10] Suppose we have three words called `beep`, `honk`, and `whistle`; we can then make up a new word, called `sound`, which will take an integer argument from the stack, with value 0, 1, or 2, and respectively beep, honk, or whistle:

```
: sound ( n --)   { beep honk whistle } ;
```

That is, `0 sound` will execute `beep`, `1 sound` will execute `honk`, and so on. The selection list may consist of any number n of entries, which are to be thought of as consecutively numbered from 0 upwards, and is delimited by the two "brace" words, namely '`{`' and '`}`'. If you attempt to execute the case statement with an argument that is less than 0 or greater than $n-1$ you get an error message.

Suppose you want to make up a word that returns the number of days in a month. You'd probably try the following

```
: days ( month -- days)
           1 -  { 31 28 31 30 31 30
                  31 31 30 31 30 31 } ;
```

Assuming that months are numbered 1, 2, ..., 11, 12, you subtract 1 in order to have the numbering 0, 1, ..., 10, 11—better suited to the case statement—and then you look up the number of days. The reasoning is correct, but there is one minor catch: in STEP Forth the case statement only accepts individual dictionary words as items in the selection list; with a few exceptions (mentioned in §5.4) numbers are *not* in the dictionary. There is an easy fix to this problem: *before* defining `days`, enter the desired numbers in the dictionary as CONSTANTS

```
28 constant 28 ( -- 28)
30 constant 30 ( -- 30)
31 constant 31 ( -- 31)
```

From this moment the token `28` (for one) will be recognized as a *word*—one that leaves a 28 on the stack just as the *number* `28` used to do before. The case statement will now accept it as a list item.

5.14 Conditional statements

The phrase between a `begin` and `again` pair is iterated forever, that between `do` and `loop` is iterated a number of times as specified by the two arguments that `do` finds on the stack. A phrase between the words `if` and `then` is executed only if the argument found on the stack by `if` has the logic value 'true'.[11]

The word

```
= ( m n -- flag)
```

compares the two arguments m and n and returns a *logic flag* having the value 'true' if they are equal and 'false' if different. Thus, the following word will beep only when the top two stack items are equal

```
: beep-if-equal ( m n --)   = if beep then ;
```

A richer construct is the `if` / `else` / `then`, used as follows

```
: beep-or-whistle ( m n --)
      = if beep else whistle then
      honk honk honk ;
```

This word will beep if m and n are equal, and whistle if they are different; after that, it will honk three times.

There also exist two *conditional iteration* statements. The first has the form '`begin` ⟨body⟩ `until`'; the ⟨body⟩ (which returns a logic flag) is iterated *until* the value of this flag becomes 'true'. The other has the form '`begin` ⟨setup⟩ `while` ⟨body⟩ `repeat`'; in which, after the first ⟨setup⟩ (which is always executed), the sequence ⟨body⟩ ⟨setup⟩ is iterated *as long as* the value of the logic flag returned by ⟨setup⟩ remains 'true'.

5.15 Visual structuring

Visual structuring of Forth code confronts us with an unusual problem. In the following code fragment, for each ASCII character code (from 0 through 127) we print the character itself if the code corresponds to a printable character, and an appropriate comment otherwise.

Since Forth uses reverse Polish notation (each operator *follows* its arguments), the most natural way to indent nested flow-control constructs would be "from the right" rather than "from the left", as shown below.

```
                128 0 do \ for all 128 codes
            cr i .     \   newline, print code
i 32 126 between if     \   if printable
        i emit else     \   then print it
     i 127 = if         \   else if 127
   ." DEL" else         \        then is DEL
  ." CTRL" then         \        else is CTRL
                        \        endif
                then    \   endif
                    loop \ endfor
```

Here `do` matches `loop`; this is shown by placing the *right* edges of these two words on the same indentation level. One level lower (and thus to the *left*) we find an `if` / `else` / `then` group, with these three keywords on the same level. The phrase associated with the `if` operator is the arithmetic condition `i 32 126 between`;

[10] Another type of case construct, called simply `case`, is available; it is more flexible but less compact and efficient than the FLASH CASE, and is not discussed here.

[11] How logic values are encoded in a Forth cell doesn't matter at this point, and is discussed in the next section.

that associated with the **else** is what is executed if the condition is true, namely **i emit**; that associated with the **then**—executed if the condition is false—is another **if / else / then** group on an inner level. The comments on the right show conventional (left-to-right) indentation.

It will up to the user to decide to what extent visual layout and structure should be correlated.

5.16 Logic expressions

In defining a STEP rule, sometimes it is convenient to treat the value of a STEP field as a logic quantity ('on' or 'off', 'true' or 'false') and sometimes as a number (0 or 1, for a two-state field; 0, 1, 2, or 3, for a four-state field; etc.). To understand precisely what is passed on the stack by one word to another in these cases, it is important to be aware of the coding conventions employed in StepForth concerning arithmetic and logic expressions.

This lengthy section is meant as a reference for cases where doubts might arise; you may quickly go over it (or skip it altogether) on first reading.

We have seen that the contents of a Forth cell consists of a 32-bit pattern. The same pattern can have different meanings, depending on agreed-upon conventions. For instance, it can be used to encode an integer from 0 to 4,294,967,295 ("unsigned number"), an integer from −2,147,483,648 to 2,147,483,647 ("signed number"), four ASCII characters (8 bits each) or, quite commonly, a single ASCII character (using only the lowest 8 bits), etc. The cell does not carry a label telling what kind of encoding was used: it is up to the programmer to arrange things so that any "user" of the pattern will know what conventions to use in interpreting it.

For example, suppose that the top cell of the stack contains the pattern 100...00101010; the three words '**.**', **u.** , and **emit** will all print on the terminal the contents of this cell. However, '**.**' will treat it as a signed number, and print '**-2147483606**'; **u.** will treat it as un unsigned number, and print '**2147483690**'; and **emit** will treat it as a character, and print '*****' (since the lower eight bits of the pattern, namely 00101010, make up the ASCII code for '*****').

In many cases it is convenient to treat the cell pattern just as a collection of separate bits—each one representing an individual binary choice. The words

```
not ( p -- r)
and ( p q -- r)
or ( p q -- r)
xor ( p q -- r)
```

are useful in this context, since they allow one to individually or jointly manipulate these bits. For instance, one can "turn off" the upper twenty-four bits of a pattern by ANDing it with an appropriate *mask* pattern, namely 00...0011111111, in which the upper twenty-four bits are "off" and the lower eight are "on".

As a reminder, the logic operations NOT, AND, OR, and XOR are defined as follows:

NOT	AND	OR	XOR
$0 \mapsto 1$	$00 \mapsto 0$	$00 \mapsto 0$	$00 \mapsto 0$
$1 \mapsto 0$,	$01 \mapsto 0$,	$01 \mapsto 1$,	$01 \mapsto 1$.
	$10 \mapsto 0$	$10 \mapsto 1$	$10 \mapsto 1$
	$11 \mapsto 1$	$11 \mapsto 1$	$11 \mapsto 0$

In particular, the logic operation NOT complements its one-bit argument, and thus the Forth word **not** complements *each* of the 32 bits of a cell. The other three logic operators act on corresponding bits of two input cells to produce a 32-bit result.

To drive a conditional statement along one or the other of two possible paths (§5.14) all one needs is a binary "flag"—with values 'true' and 'false'.[12] For this purpose, a one-bit cell would be sufficient; but Forth cells come in a standard size of 32 bits, and one must have an agreement on which 32-bit pattern(s) should mean 'true' and which 'false'.

The Forth-83 standard stipulates that words that *return* a logic flag (such as **=** and similar "compare" words) should never put on the stack anything but the patterns 11...11 for 'true' or 00...00 for 'false'; for convenience, these patterns have been entered in the dictionary, as CONSTANT words, under the names **true** and **false**.[13] On the other hand, words that *expect* a logic flag (such as **if** and similar "conditional" words) will treat as 'false' the pattern 0000000000000000 and as 'true' *any other pattern*.

As long as one uses only the **true** and **false** patterns for 'true' and 'false', the bitwise logic operations **not** , **and** , etc. can also be used to manipulate such logic flags. However, if one tries to take advantage in an indiscriminate fashion of the wider "catching range" of **if**[14] some subtle problems may arise. We shall give just one example, as a warning to the reckless programmer.

Consider the word

```
: beep-if-not-equal ( m n --)
                  = not if
                 beep then ;
```

(cf. §5.14), which beeps only when m and n are *not* equal. This word would work the same if one replaced '**= not**' by just '**-**'; In fact, if m and n are equal their difference

[12] When more than two choices present themselves, it is usually more natural to use a case statement (cf. §5.13) rather than many nested **if** statements.

[13] Note that, when printed as signed numbers, **true** will yield **-1** and **false** will yield **0**; as an unsigned number, **true** will yield **4294967295** (**ffff** in hexadecimal).

[14] As when using arithmetic as a shortcut to logic.

$m-n$ will be 0, and will be seen as 'false' by `if`; on the other hand, if they are different $m-n$ will be a pattern containing at least one non-zero bit, and will be seen as 'true' by `if`.

Well, if '`-`' works "the same" as '`= not`', won't '`- not`' work the same as '`=`' in `beep-if-equal` of the previous section? If $m = n$, their difference $m - n$ is 0 and its complement as given by `not` is the pattern of all `1`s (the `true` pattern)—which of course is recognized as 'true' by `if`, as we intended. If $m \neq n$, the difference pattern *will* contain some 1's but *may* also contain some 0's; thus the complementary pattern returned by `not` may contain some `1`s—in which case it will again be recognized by `if` as 'true', which is *not* what we intended.

♠ **[adapt to field words]** Since neighbor words such as `center`, `north`, etc. return `1` or `0` as a value, we shall use these values as respectively 'true' and 'false' whenever expedient. Logic operations involving such 1-bit flags work well, except for `not` (since it complements all 32 bits). The two word sequence '`1 xor`' can be used to get the 1-bit complement. Alternatively, comparisons such as '`=`' '`>`' '`<`' and '`<>`' (not equal) can be used to convert 1-bit flags into standard logic flags.[15]

Finally, it will be useful to remember that word a defined by `==`, and thus associated with a certain subset of n STEP layers, will care only about the n least significant bits of a Forth cell (cf. §??). Any garbage that may have accumulated in the remaining bits of the cell as a result of arithmetic/logic manipulation tricks will be ignored.

5.17 Numerical tasks

No floating-point package is provided with the current version of STEP Forth. It is recommended that other programs be used for intensive numerical tasks. Data files can be used for exchanging data between STEP Forth and these programs. Moreover, separately compiled C procedures can be directly linked to the Forth code.

5.18 Search order

We have seen that, in addition to the main section, called `forth`, the Forth dictionary may contain a number of more specialized sections (`assembler`, `editor`, etc.). *Search order* commands allow one to do the necessary vocabulary switching. At any moment, the CURRENT vocabulary is the one in which new definitions are inserted, while the CONTEXT vocabulary is the one where old definitions are looked up. (Of course, most of the time both `current` and `context` pointers point to the `forth` vocabulary.) Just mentioning a vocabulary, such as `editor`, makes it the CONTEXT vocabulary, superseding the former CONTEXT; the word `definitions` makes CURRENT be whatever the CONTEXT is at that moment. Thus, after saying

```
editor definitions forth
```

`forth` is CONTEXT but `editor` is CURRENT: the interpreter will keep looking words up in the `forth` vocabulary, but any new definitions will be added to the `editor` vocabulary.

The vocabulary organization provides a way to use the same name with different meanings in different contexts, or to make a word unavailable in a certain context. In STEP Forth, for instance, words that are meaningful only for manipulating certain registes are segregated in special vocabularies and made visible only in the appropriate context.

The concept of "context vocabulary" is extended by providing a number of optional "second (third, fourth, etc.) choices" when a given token is not found in the primary context vocabulary. Although most STEP users will not need to explicitly use this mechanism, it is an integral part of the way that register setup options and menus are defined in the STEP software, and so we briefly discuss it here.

The additional vocabulary choices are arranged stack fashion—the *search-order* stack. In addition, there is a minimal `root` vocabulary which normally is always available as a last-resort choice, and which contains the search-order words described below. The command `only` empties the search stack and makes `root` the CONTEXT vocabulary. After saying `only`, the search order will be

$$\text{CONTEXT:}\ \begin{array}{c}\text{root}\\ \hline \text{root}\end{array}$$

(where the black line represents the search-order stack, now empty). Naming a vocabulary makes that vocabulary the CONTEXT; if you say `forth` now, the search order becomes

$$\text{CONTEXT:}\ \begin{array}{c}\text{forth}\\ \hline \text{root}\end{array}$$

`also` pushes a copy of the CONTEXT vocabulary onto the stack; if you said `also` now the search order would become

$$\text{CONTEXT:}\ \begin{array}{c}\text{forth}\\ \boxed{\text{forth}}\\ \text{root}\end{array}$$

meaning, "Look it up in `forth`; if that fails, try `forth` again; if that fails, you've got a last chance with `root`; if *that* fails, then give up!"

[15] The words '`0=`' '`0>`' etc. also exist as abbreviations for '`0 =`' etc.

If you further say `meta also assembler`, the search order (which is displayed if you type `order`) will become

CONTEXT:	assembler
	meta
	forth
	root

Finally, the command `previous` pops the top of the search stack into CONTEXT, dropping the previous CONTEXT.

This may sound confusing, but is in practice quite straightforward if you don't try to take advantage of all the tricks that are in principle possible. Usually you put in the search stack *all* of the vocabularies to be searched in a certain situation, without worrying much about their order and without relying on CONTEXT (so that the latter can be switched rather freely, without going every time through the `also`–`previous` business). You explicitly make a vocabulary the CONTEXT—by just naming it—only (a) when it's a rather secret one that you want to occasionally pull out of your hat to give a special meaning to the word that follows, or (b) though it's already in the back-up stack, you want to make sure that it will be searched *first*.

The CONTEXT vocabulary and the rest of the search-order machinery only have to do with where a word will be looked up. As mentioned above, newly defined word are added to the *current* vocabulary—and there is only one CURRENT vocabulary at any moment. The word `definitions` makes the CURRENT vocabulary be whatever CONTEXT is. Thus, if you say

```
editor definitions forth
```

the CONTEXT switches to `editor`, then `editor` is made the current vocabulary (by `definitions`), and finally the CONTEXT reverts to `forth`. Though the search order still starts with Forth, the next definitions will be added to the `editor` vocabulary.

5.18.1 The Forth thesaurus

Traditionally, a Forth system is accompanied by a **glossary**, i.e., a plain English description, arranged in dictionary form, of the words that make-up the system itself. We have found the **thesaurus** format much more compact, informative, and pedagogical. In the thesaurus, all the StepForth words are arranged and discussed *by subject*; an alphabetical index allows one to locate the thesaurus section where the desired word and related words are located.

5.19 Further reading

[revise]

Starting FORTH by Leo Brodie (Prentice-Hall, 1981) is an excellent practical introduction to Forth, while *Thinking FORTH*, by the same author (Prentice-Hall, 1984), discusses the methodology that inspires this programming language. *Inside F83*, by C. H. Ting (Offete Enterprises, 1306 South B. St., San Mateo, CA 94402; 1985) is a thorough description of F83's internal structure. The book *Mastering Forth*, by A. Anderson and M. Tracy (Brady Communications Co., 1984), describes Micromotion Forth—a commercial package based on the F83 model (and thus having a close family relationship with the present implementation).

The complete specifications for the *Forth-83 Standard* (cf. Appendix ??) are available from the Forth Interest Group (P.O. Box 8231, San Jose, CA 95155).

The periodical *FORTH Dimensions*, published by the Forth Interest Group, P.O. Box 8231, San Jose, CA, is a good source for news, applications, programming techniques, literature listings, and software and hardware developments; most of the above books can be purchased from the F.I.G. The *Journal of Forth Application and Research*, published by the Institute for Applied Forth Research, Inc., is a more academically oriented publication.

6 Guide to the thesaurus

6.1 About the thesaurus

The STEP software thesaurus consists of three parts:

- A **Guide to the thesaurus**—the present chapter.
- The **StepForth thesaurus**, describing, under individual entries arranged by topic, all the StepForth words that are intended for use by the STEP application programmer. The thesaurus is divided into two chapters: a **Forth thesaurus** (Chapter ??), devoted to generic Forth constructs, and a **Step thesaurus** (Chapter ??), devoted to STEP-specific constructs.
- The **Thesaurus index**—a single alphabetical cross-reference to all the thesaurus entries.

Since StepForth is a close relative of F83, for which much tutorial and reference material is already available, the Forth-generic entries of the thesaurus are conceived as a utilitarian reference—not an in-depth technical manual for the Forth language. On the other hand, the STEP-specific entries try to be fairly exhaustive, so as to provide a complete reference for STEP machine and support the needs of even the sophisticated user.

Moreover, through the `see` facility (Forth decompiler/disassembler) and the `view` facility (source-code definition locator), the systems programmer has direct access from StepForth to all the StepForth internals and the accompanying online documentation.

To every relevant StepForth word there corresponds an **entry** in the thesaurus. The entries are arranged by subject and grouped under separate **topics**. Entries are liberally cross-referenced to other entries and topics. All entries are listed in alphabetical order in the thesaurus index.

6.1.1 Thesaurus topics

Each thesaurus topic is in the form a a reasonably self-contained subsection of text, headed by a section number and a descriptive name. In turn, topics are arranged by subject and grouped within larger sections. For instance, the comparison of unsigned numbers is discussed in the section **Arithmetic and logic operations** under the topic

7.1.4 Unsigned comparison

The topic number ("7.1.4" in the above example) is the target of cross-references found in the Thesaurus index.

A topic starts with a brief summary of the subject at hand. This summary is followed by headings, collected in tabular form, for all the entries that are discussed under that topic. Each heading includes the entry's name as a Forth word, its stack and input-stream interfaces, and indications on the lexical/semantic type of the word (see §6.1.2). Finally, the entries are discussed in detail in discursive form, with notes on usage, examples, and cross-references.

Occasionally, when clarity demands, the listing of heading is broken into two or three separate segments, each followed by the text describing the entries in that segment.

6.1.2 Thesaurus entries

The structure of thesaurus entries is illustrated by the following example, where the headings for the three entries `forget`, `(forget)`, and `fence` are followed by the description of the entries themselves.

`forget`	(—) *name*	: I
`(forget)`	(*acf* —)	:
`fence`	(— *addr*)	`variable`

If *name* is found in the CURRENT vocabulary, `forget` deletes *name* from it, as well as all words added to the dictionary after *name*, regardless of their vocabulary. If *name* is not found, or the name of the CURRENT vocabulary has been deleted, an error condition occurs.

`forget` calls `(forget)`, which expects the *acf* of a word from the stack rather than its name from the input stream. If the *acf* of the specified word is lower than the value of the user variable `fence`, `(forget)` aborts with the message "`Below Fence`" and no forgetting will take place; this is to prevent accidentally erasing critical parts of the system.

The conventions used in the headings are explained immediately below.

Forth word naming conventions

Most Forth words that return a flag have a name that ends with a question mark. Most words whose action depends on a flag have a name that starts with a question mark. Words that start with a parenthesis are usually lower-level versions of a similar word without parenthesis. Other conventions used in naming words either are generic Forth conventions or are explained along with the relevant words.

All single-digit hex numbers (`0` through `f`) are defined as Forth constants; no two-digit hex numbers (i.e., `00` through `ff`) are used as names of words in the `forth` vocabulary. [How about `cd` and `df` ?]

Most Forth words are similar in spelling to ordinary English words and are pronounced accordingly. The words `dup` and `rot` are pronounced "dupe" and "rote". A question mark is pronounced "question"; a left or right parentesis, "paren"; a left or right bracket, "bracket". For example, `sys-(key?` is pronounced "sys-paren-key-question".

Stack and input stream

Following the entry name is the **stack interface**, in parentheses; this may be immediately followed by an **input-stream interface**. If *both* stack and input-stream interfaces are empty

Stack interface. In the above example, the word `(forget)` consumes one stack item, which will be called *acf* in the text that describes the word, and returns nothing on the stack. On the other hand, `fence` does not consume any stack items but returns one item, called *addr*. The long dash in the stack interface separates items consumed from items returned. The main entry `forget` does not affect the stack; in this case no stack interface is given, i.e. the notation (—) is suppressed.

Certain naming conventions are used within the stack interface; these conventions are made use of in the accompanying text that describes the behavior of an entry word. *n* stands for a generic signed number; *+n*, for a non-negative signed number; *-n*, for a positive one. *flag* stands for a logical flag (with value TRUE=−1 or FALSE=0); *mask*, for a bit mask; *len*, for a length (e.g., number of bytes in a string); *pstr*, for the address of a *packed string* (the characters that make up the string are preceded by a *count* byte); *addr*, for an address. A mnemonic such as *row#* means "row number" (i.e., which row), while *#rows* means "number of rows" (i.e., how many rows).

Input-stream interface. A Forth word may consume characters from the input stream, which will be parsed into one or more strings on the basis of delimiting characters that depend on the word itself. In the above thesaurus sample, `forget` consumes a *name*, i.e., a string of consecutive nonspace characters (see `blword` in the glossary); this string is terminated by a space. If the string is terminated by other characters, this is explicitly indicated; for instance, the entry

`abort"` *text*"

means that `abort"` expects a text string terminated by the character '"'; the entry

`sh` *rest-of-line*↩

means that `sh` consumes a string *rest-of-line* terminated by a carriage return.

Word type

The right end of each entry line is reserved for information about the *type* of the entry word, namely, its **definer** and its **attributes**, as explained below.

In the above thesurus sample, `forget` is a "COLON word", i.e., a word defined by the COLON compiler; `fence` is a Forth VARIABLE, defined by the word `variable`. Knowing by what defining word a given entry word was created gives useful clues about the latter's role in a program. Thus, a word created by the COLON compiler (`:`) is typically a procedure; a word created by `variable` is associated with a one-cell data-structure and returns on the stack the cell's address, while one created by `constant` returns the cell's contents; a word created by `defer` contains a pointer to another word, and executes that word when called; a word created by `create-buffer` is associated with a *buffer* data structure; etc.

The definer may be accompanied by superscripts that tell that the entry word has one or more of the attributes *immediate*, *deferred*, *compile-only*, or *interpret-only*, with the following meanings

†	immediate	The entry word is *immediate*, i.e., is immediately interpreted even during colon compilation.
*	deferred	The word is *deferred*, i.e., it contains a pointer to another word and executes that word when called.
C	compile	The word is intended for use only in COMPILE mode.
I	interpret	The word is intended for use only in INTERPRET mode.

Thus,

`literal` $(n$ —$)$: †C

means that `literal`, defined by the colon compiler, is an immediate word normally used only in COMPILE mode.

7 Forth thesaurus

7.1 Arithmetic and logic operations

7.1.1 Addition and subtraction

`+` *(n1 n2 — n3)*
`-` *(n1 n2 — n3)*
`1+` *(n — n')*
`1-` *(n — n')*
`2+` *(n — n')*
`2-` *(n — n')*
`+!` *(n addr —)*

`+` returns the arithmetic sum of *n1* and *n2*; `-` returns their difference *n1*−*n2*.

`1+` increments by 1 the top stack entry; `1-` decrements it by 1.

`2+` increments by 2 the top stack entry; `2-` decrements it by 2.

`+!` increments by *n* the Forth cell at *addr*.

7.1.2 Negation

`negate` *(n — −n)*
`?negate` *(n flag — n | −n)*
`abs` *(n — u)*

Returns the absolute value of *n*. The maximum negative number (hex `-8000`), which has no positive counterpart, is left unchanged.

`negate` returns −*n* (i.e., the *opposite*, *additive inverse*, *two's complement*, or *arithmetic negation* of *n*).

`?negate` negates *n* if flag is TRUE.

`abs` returns the absolute value of *n*. The maximum negative number (hex `-8000`), which has no positive counterpart, is left unchanged.

7.1.3 Signed comparison

In signed comparison, a Forth cell is interpreted as ranging from `-80000000` to `+7fffffff`.

`=` *(n1 n2 — flag)*
`<` *(n1 n2 — flag)*
`<>` *(n1 n2 — flag)*
`<=` *(n1 n2 — flag)*
`>` *(n1 n2 — flag)*
`>=` *(n1 n2 — flag)*

These words return TRUE if the numbers *n1*, *n2* stand in the following relation

word	relation
`=`	n1 = n2
`<`	n1 < n2
`<=`	n1 ≤ n2
`<>`	n1 ≠ n2
`>`	n1 > n2
`>=`	n1 ≥ n2

7.1.4 Unsigned comparison

In unsigned comparison, a Forth cell is interpreted as ranging from `0` to `ffffffff`.

`u<` *(u1 u2 — flag)*
`u<=` *(u1 u2 — flag)*
`u>` *(u1 u2 — flag)*
`u>=` *(u1 u2 — flag)*

These words return TRUE if the unsigned numbers *u1*, *u2* stand in the following relation

word	relation
`u<`	u1 < u2
`u<=`	u1 ≤ u2
`u>`	u1 > u2
`u>=`	u1 ≥ u2

7.1.5 Comparison with zero

`0=` *(n — flag)*
`0<` *(n — flag)*
`0<=` *(n — flag)*
`0<>` *(n — flag)*
`0>` *(n — flag)*
`0>=` *(n — flag)*

These words return TRUE if the number *n* stands in the following relation with 0

word	relation
`0=`	n = 0
`0<`	n < 0
`0<=`	n ≤ 0
`0<>`	n ≠
`0>`	n > 0
`0>=`	n ≥ 0

7.1.6 Min; max

`min` *(n1 n2 — n3)*
`max` *(n1 n2 — n3)*
`umax` *(u1 u2 — u3)*
`umin` *(u1 u2 — u3)*
`between` *(n lo hi — f)*
`within` *(n lo hi — f)*

`min` returns the lesser of *n1* and *n2* according to the operation of `<` (signed compare). Similarly, `max` returns the greater between the two according to the operation of `>`.

`umin` returns the lesser of *u1* and *u2* according to the operation of `u<` (unsigned compare). Similarly, `max` returns the greater between the two according to the operation of `u>`.

`between` returns TRUE if $lo \le n \le hi$. `within` returns TRUE if $lo \le n < hi$. Thus, both 9 and 10 are "`between`" 0 and 10, but only 9 is "`within`" them.

7.1.7 Multiplication

`*` *(n1 n2 — n3)*
`u*` *(n1 n2 — n3)*

`*` returns the product of *n1* and *n2* treated as signed numbers. In case of overflow, only the least significant 32 bits are kept.

Similarly, `u*` returns the unsigned product of *n1* and *n2*.

7.1.8 Division

`/` *(n1 n2 — n3)*
`/mod` *(n1 n2 — u n3)*
`u/mod` *(u1 u2 — u u3)*
`mod` *(n1 n2 — n3)*
`even` *(n — n')*

`/` returns the quotient *n3* of *n1* divided by *n2*. An error condition results if the divisor is zero. `/mod` also returns the remainder in *u*. `mod` only returns this remainder.

`u/mod`, which is the basic division word in Forth, performs *unsigned* division and leaves quotient and remainder much as `/mod` does.

`even` returns *n* itself if *n* is even, and one more than *n* if *n* is odd.

```
*/        (n1 n2 n3 — n4)
*/mod     (n1 n2 n3 — n4 n5)
```

`*/` multiplies *n1* by *n2* producing an intermediate result *n*, and then divides *n* by *n3*, yielding the quotient *n4*. In addition, `*/mod` returns the remainder *n5*. See `/mod` for division exceptions.

In the 1983 Forth Standard, which is oriented toward 16-bit Forth cells, the intermediate result *n* is kept in a higher precision than the two factors, and the division takes advantage of this higher precision. In this 32-bit system, the intermediate result has the same 32-bit precision.

7.1.9 Shifts

```
<<        (n +n — n')
>>        (n +n — n')
>>a       (n +n — n')
2*        (n — n')
4*        (n — n')
8*        (n — n')
2/        (n — n')
u2/       (n — n')
```

`<<` shifts *n* left by *+n* bit positions, filling least significant bits with zeros; *n'* is the result.

`>>` shifts *n* right by *+n* bit positions, filling the most significant bits with zeros; `>>a` (arithmetic right shift) does the same, but extends the sign of *n* into the most significant bits.

`2*` is equivalent to `1 <<`; `4*`, to `2 <<`; `8*`, to `3 <<`.

`2/` is equivalent to `1 >>a`; `u2/`, to `1 >>`.

7.1.10 Logic operations

```
and       (n1 n2 — n3)
or        (n1 n2 — n3)
xor       (n1 n2 — n3)
not       (n — n')
cnot      (n — n')
```

`and`, `or`, and `xor` respectively return the logical AND, OR, and XOR of *n1* and *n2*, operating bitwise on all 32 bits.

`not` complements each bit of *n*. `cnot` complements only its least-significant 8 bits.

7.1.11 Bit manipulation

```
creset    (byte-mask addr —)
cset      (byte-mask addr —)
ctoggle   (byte-mask addr —)
```

These words respectively set, reset, or complement selected bits of the byte at *addr*. The selection is performed by the mask *byte-mask*: only bit-positions where there is a 1 in the mask are affected.

7.1.12 Byte and word manipulations

```
flip      (wl — w2)
```

w2 is the result of exchanging the bytes in the lower 16 bits of the number wl.

```
l->w                                n --- w
```

Converts the 32-bit number *n* to a 16-bit number by truncating the high order bits.

```
lwsplit   (n — w0 w1)
wljoin    (w0 w1 — n)
wflip     (n0 — n1)
```

`lwsplit` Puts in *w0*, *w1* respectively the bottom and top 16-bit halves of *n*. Its inverse is `wljoin`, which concatenates into *n* two 16-bit numbers to form a 32-bit number.

`wflip` exchanges the two halves of `n0`, yielding *n1*.

7.2 Flow control

7.2.1 Begin constructs

```
begin                                  †C
while     (flag —)                     †C
repeat                                 †C
until     (flag —)                     †C
again                                  †C
```

The BEGIN constructs have the following form

`begin ... again`

`begin ...` *flag* `until`

`begin ...` *flag* `while ... repeat`

The BEGIN–AGAIN loop is re-executed indefinitely.

The BEGIN–UNTIL loop is re-executed as long as *flag* is TRUE.

The BEGIN–WHILE–REPEAT loop is re-executed as long as *flag* is TRUE. When *flag* becomes FALSE, the loop is exited from the `while` position, skipping the portion between `while` and `repeat`.

The above constructs can be nested.

7.2.2 Case constructs

```
case      (selector — selector)        †C
of        (selector test-value —)      †C
endcase   (selector —)                 C
endof                                  C
```

These words implement Charles Eaker's CASE construct, illustrated by the following example

```
                              : foo ( item -- )
                   case
    0 of ." it was 0" endof
    4 of ." it was 4" endof
    1 of ." it was 1" endof
  ( item)." it was " dup u.
                endcase ;
```

The default clause, after the last `endof`, is optional. When an `of` clause is executed, *item* is *not* on the stack. When a default clause is executed, *item is* on the stack. The default clause may use the selector, but must not remove it from the stack (it will be automatically removed by `endcase`). At run time, `of` tests the top of the stack against *item*. If they are the same, *item* is dropped and the following Forth code is executed. If they are not the same, execution continues at the point just following the matching `endof`.

7.2.3 If constructs

```
if        (flag —)                     †C
then                                   †C
else                                   †C
?pairs    (n0 n1 —)
```

The IF construct has one of the following two forms

flag `if` *true-clause* `else` *false-clause* `then`

flag `if` *true-clause* `then`

If *flag* is TRUE, the *true-clause* is executed and *false-clause* (if any) is skipped. Otherwise, *false-clause* (if any) is executed.

IF constructs may be nested.

`?pairs` aborts if $n0 \neq n1$, printing the message "`Conditionals not paired`". This is used to ensure that conditionals such as IF–THEN are properly matched and nested. The way this works is the following. The beginning word of a conditional (such as `if`) leaves a magic number *n0* on the stack; the ending word of the conditional uses leaves its own number *n1*, and calls `?pairs` to check that the two numbers on the stack match.

7.2.4 Do constructs

`do`	(*limit base* —)	†C
`?do`	(*limit base* —)	†C
`(do)`	(*limit base* —)	
`(?do)`	(*limit base* —)	
`loop`		†C
`(loop)`		
`+loop`	(*n* —)	†C
`(+loop)`	(*n* —)	
`i`	(— *n*)	
`j`	(— *n*)	
`k`	(— *n*)	
`leave`		†C
`(leave)`		
`?leave`	(*flag* —)	†C
`(?leave)`	(*flag* —)	
`bounds`	(*addr len* — *addr' addr*)	

The the two basic formats of DO construct are

limit base `do ... loop`

limit base `do ...` *incr* `+loop`

Briefly, the loop is executed a number of times, with an internal variable *index* stepping from the value of *base* to that of *limit*. The value of *index* is available within the body of the loop.

With `do`, the loop *body* is always executed at least once; with the `?do` variation, the loop is not executed at all if *base*=*limit*.

On the first iteration, *index*=*base*. At the end of the iteration, *index* is incremented by 1 (for `loop`) or by *incr* (for `+loop`). The value of *incr* may be positive, negative, or zero. If the increment moves *index* across the boundary between *limit*−1 and limit the loop is terminated. For example, `10 0 do ... loop` executes 10 times; `10 0 do ... loop` executes once. On the other hand, `0 0 do ... loop` executes 4,294,967,295 (hex `ffff`) times!

DO loops may be nested. However, note that they use the return stack to store *index* (one stack level per nesting level); cf. `exit`.

The words `i`, `j`, and `k` return the value of *index* from, respectively, the current DO level, the one right above it, and the one immediately above.

`leave` terminates the loop immediately. `?leave` is equivalent to `if leave then`.

`(leave)` is the run-time word compiled by `leave`; `(?leave)`, that compiled by `?leave`.

`(do)` is the run-time word word compiled by `do`. At run time, `end` and `start` are removed from the stack and used to control the execution of the DO-LOOP construct. The loop is always executed at least once.

`(?do)` is the run-time word word compiled by `?do`. It works like `(do)`, but if *limit*=*base* the loop is not executed at all.

`(loop)` is the run-time word compiled by `loop`. At run time, the loop index is incremented and compared to the end value saved by the previous `do` or `?do`. If the index has reached or exceeded the ending value, the loop is terminated, otherwise it is executed once more.

`(+loop)` is the run-time word compiled by `+loop`. It works like `(loop)`, but increments the loop index by *n* instead of by 1.

`bounds` is a general utility word that derives its name from its most frequent usage, i.e., to convert a starting address and a count into *bounds* appropriate for the DO construct—an ending address under a starting address. *addr'* is the sum of *addr* and *len*.

7.2.5 Branches

`branch`		C
`?branch`	(*flag* —)	C
`<mark`	(— *addr*)	C
`>mark`	(— *addr*)	C
`<resolve`	(*addr* —)	C
`>resolve`	(*addr* —)	C

The "branch" words are compiled into the dictionary by flow-control constructs such as DO–LOOP, BEGIN–UNTIL, etc. They may also be coded explicitly by the programmer; this, however, requires adequate care and knowledge.

`branch` is a literal: at execution time, the dictionary cell following a `branch` instruction is used as a branch address by the `branch` instruction itself. In this system, branch addresses are actually in the form of 16-bit offsets. The only difference between `branch` and `?branch` is that the first is unconditional, while the second is enabled only if there is a TRUE value on the stack.

The "mark" words are used during compilation by the flow-control constructs in order to mark the origin of forward branches (when the branch destination is still not known) and the destination of backward branches (when the branch origin is still not known).

The "resolve" words are used when the above origins and destinations become known, to compute branch addresses and write them in the branch literals.

7.3 Dictionary

7.3.1 Entry creation

`create`	*name*	*
`"create`	(*pstr* —)	
`(create`	*name*	
`last`	(— *addr*)	
`warning`	(— *addr*)	
`alias`	*name' name*	

`create` creates a dictionary entry for *name*; the next available dictionary location is the first byte of *name*'s parameter field. No space is allocated at this time in the parameter field. When *name* is executed, the address of the first byte of its parameter field is left on the stack.

`create` is deferred, so that the user may substitute another implementation of it if desired; it defaults to `(create`.

`"create` has similar behavior, but takes the name token from memory (typically from the stack) as a packed string at *pstr* instead of from the input stream.

`last` is a user variable containing the name field address of the last word created.

`warning` is a user variable controlling the "`isn't Unique`" message. If a word *name* is created but there is already a word with the same name in the CURRENT vocabulary, the message "*name* `isn't Unique`" is normally displayed. `warning off` disables this message, and `warning on` re-enables it.

`alias` adds a new word, *name'*, to the dictionary as a synonym for the existing word *name*. When *name'* is later encountered, the effect is the same as if *name* had been encountered instead. The replacement is done during text interpretation: the new word *name'* is always immediate.

7.3.2 Enclosing

Enclosing is the process of incrementally adding material to the Forth dictionary. A Forth word, as a dictionary construct, typically consists of a **header** and a **body**. In turn, header and body consist of collections of **fields** of various sizes. The "comma" words are the lowest-level primitives used to construct such fields.

`,` (n —)
`c,` (n —)
`w,` (n —)

`,` encloses n as a Forth cell in the dictionary. `c,` encloses the least-significant 8 bits of n as a byte in the dictionary (only one byte is allotted); similarly, `w,` encloses the least-significant 16 bits.

`,"` *text"* C
`",` (*addr len* —)

`,"` encloses *text* in the dictionary as a packed string. With `",`, the text to be enclosed is taken from memory at *addr* for a length *len*, instead of from the input stream.

7.3.3 Forgetting

`forget` *name*
`(forget)` (*acf* —)
`fence` (— *addr*)

If *name* is found in the CURRENT vocabulary, `forget` deletes *name* from it, as well as all words added to the dictionary after *name*, regardless of their vocabulary. If *name* is not found, or the name of the sc current vocabulary has been deleted, an error condition occurs.

`forget` calls `(forget)`, which expects the *acf* of a word from the stack rather than its name from the input stream. If the *acf* of the specified word is lover than the value of the user variable `fence`, `(forget)` aborts with the message "`Below Fence`" and no forgetting will take place. This is to prevent accidentally erasing critical parts of the system.

7.3.4 Dictionary listing

`words`
`another?` (— *anf T* | *F*)
`follow` (*threads* —)

`words` lists the names of the words in the topmost CONTEXT vocabulary starting with the most recent definition.

`another?` moves to the next word in the scanned vocabulary, leaving its *anf* and TRUE on the stack. If we are already at the end of the vocabulary, only FALSE is left.

`follow` initializes vocabulary scanning so that subsequent uses of `another?` will select words from the vocabulary whose threads are stored at *threads*.

Example: Here is a simple implementation of `words`, ignoring output formatting. `words-in` is similar to `words`, but it takes an explicit vocabulary name argument, as in `words-in assembler`. `words-in` illustrates the use of `>threads`.

```
: (words ( threads -- )
   follow  begin
  another while
      .id repeat ;
: words
   context @ (words ;
: words-in ( -- )
   [compile] '            \ vocabulary name
   >threads (words ;
```

7.3.5 Dictionary lookup

`'` (— *acf*) *name*
`[']` *name* †
`(')` (— *acf*)

The word `'` looks up *name* in the dictionary, using the current search order. If *name* is defined, it returns its compilation address *acf*; otherwise it aborts with an error message. This is an ordinary Forth word: if encountered during compilation, it is compiled; when the word is executed, *name* will be read from the input stream its *acf* returned.

The corresponding immediate word (normally to be used only during compilation) is `[']`, which looks up *name* and stores it as a literal governed by `(')`. When the latter executed, it will leave *acf* on the stack.

Also see `'user#`.

`find` (*pstr* — *addr n*)
`vfind` (*pstr voc-ptr* — *acf 1* | *acf -1* | *pstr F*)

`string` looks up in the current search order (see `order`) the word whose name is given by *pstr*. It the word is not found, *addr* is the string address, and *n* is zero. If the word is found, *addr* is its compilation address and *n* is set to 1 if the word has the immediate, and to -1 if non-immediate.

`vfind` is similar to `find`, except that instead of searching every vocabulary in the search order it only searches a single vocabulary, pointed to by `voc-ptr`. (A vocabulary pointer is the number which is stored in `context` by executing a vocabulary word.)

`defined` (— *pstr 0* | *acf n*) *name*

If *name* is found in the currently active search order, it returns its *acf* that word, followed by 1 if the word is immediate or -1 if the word is not immediate. If *name* is not found, it returns a packed string containing *name* as it was actually looked up (perhaps after case conversion, if this is enabled).

7.3.6 Dictionary pointer

`here` (— *addr*)
`dp` (— *addr*)
`allot` (n —)

`here` returns the address of the next available dictionary location. This address is kept in the variable `dp` (*dictionary pointer*), which is adjusted whenever material is added to or removed from the dictionary.

`allot` allocates n bytes in the dictionary (n may be negative). The address of the next available dictionary location, stored in `dp`, is adjusted accordingly.

7.3.7 Field traversing

`>body` `acf --- apf`
`>data` (*acf* — *addr*)
`>link` (*acf* — *alf*)
`>name` (*acf* — *anf*)
`>threads` (*acf* — *addr*)
`body>` (*apf* — *acf*)
`>user` (*apf* — *user-var-addr*)
`>user#` (*acf* — *user#*)
`word-bounds` (*acf* — *apf addr*)
`link>` (*alf* — *acf*)
`l>name` (*alf* — *anf*)
`n>link` (*anf* — *alf*)
`name>` (*anf* — *acf*)

Given a compilation address *acf*, `>body` returns the corresponding parameter field address *apf*; `>data` returns the address where the associated data are stored (the `pfa` for most words, but an address in the user area for user variables); `>link` returns the link-field address `alf`; `>name` returns the name-field address *anf*.

`>threads` assumes that *acf* is the compilation address of a vocabulary, and returns a pointer to where the threads are stored.

`body>` returns the *acf* corresponding to the parameter field address *apf*.

`link>` returns the *acf* corresponding to the given link field address.

`l>name` returns the name field address corresponding to the given link field address. `n>link` returns the link field address corresponding to the given name field address.

`name>` returns the compilation address corresponding to the given name field address.

`>user` returns in `user-var-addr` the address of the data storage area of the user variable whose parameter field address is *apf*. In this implementation, vocabularies and deferred words also keep their data in the user area, so `>user` also works for them.

`user#` is the user number (the byte offset into the user area) of the storage owned by the word at *acf*. If the word does not have any user area storage, the result is undefined.

`word-bounds`, given the *acf* of a word, returns in *apf* and *addr* the beginning and the end of the body of that word in the dictionary.

7.4 User area

7.4.1 User area

The **user area** (or **process control block**) contains important Forth system variables for which different copies need to be kept for different tasks.

`user`	(*offset* —) *name*
`nuser`	*name*
`tuser`	*name*
`user-size`	(— *n*)
`up@`	(— *addr*)
`up!`	(*addr* —)
`up0`	(— *addr*)
`#user`	(— *addr*)
`'user#`	(— *user#*) *name*
`ualloc`	(*len* — *new-user-number*)
`init-user`	

`user` is a defining word used in the form

offset `user` *name*

When *name* is later executed, it leaves the address of its data storage area which is at the given offset in the user area. Space for this in the user area is not allocated. The allocation must be done separately, for instance with `ualloc`.

`up@` returns the address of the currently active user area.

`user-size` is the size in bytes of the user area.

`up0` is a user variable containing the starting address of the given user area.

`up!` assigns *addr* as a new starting address of the user area. The new user area must have been already initialized before `up!` is executed, or the system may crash.

The variable `'user#` returns the user number (the byte offset into the user area) of the storage associated with the word *name*. If the word does not have any user area storage, the result is undefined.

`#user` is the address of a user variable containing the offset of the first location in the user area which has not yet been allocated.

The phrase

`nuser` *name*

defines a word *name* which, when later executed, leaves the address of its data storage area—which is in the user area. Enough space to store a 32-bit cell is allocated.

Similarly, the word defined by

`tuser` *name*

is associated to a data storage space large enough to contain an address.

`nuser` and `tuser` are similar to the standard word `user`, except that one doesn't have to keep track of which user area locations have already been allocated.

`ualloc` returns the next available user number, and allocates *len* bytes in the user area so that the next call to `ualloc` will return a number past the bytes just allocated.

`init-user` initializes the user area; it is automatically called when Forth is started.

7.5 Data structures

7.5.1 Variables; constants

`variable`	*name*
`off`	(*addr* —)
`on`	(*addr* —)
`constant`	(*n* —) *name*
`true`	(— -1)
`false`	(— *0*)

`variable` creates a dictionary entry for *name* and allots one Forth cell in its parameter field. This parameter field is to be used for the contents of the variable. The application is responsible for initializing the contents of the variable which it creates. When *name* is later executed, the address of its parameter field is placed on the stack.

`off` and `on` store, respectively, FALSE and TRUE in the data cell at *addr*. They are often used to initialize a variable.

`constant` creates a dictionary entry for *name* such that, when *name* is later executed, *n* will be left on the stack.

`true` and `false` return respectively the values TRUE , which is -1, or `ffffffff` in hex, and FALSE, which is 0.

Small numbers, namely, those corresponding to the first 10 hex digits `0 1 2 3 4 5 6 7 8 9 a b c d e f`, are defined as constants. Thus, for instance, `f` will return 15 in decimal and `f` in hex.

7.5.2 Value

`is`	(*value* —) *name*	†C
`(is)`	(*value* —)	
`(is`	(*value acf* —)	

Many Forth words (e.g., word defined by `constant`) return *data* without providing information as to the *address* where the data are kept. On the other hand, it is sometimes desirable to modify the data at that address. In many cases, the address in question can be figured from information available at compile time. The IS construct provides a mechanism for saving this information as a literal at compile time, so that it can later be used, at execute time, for accessing the location at that address.

The construct

value `is` *name*

is applicable only if *name* is an already defined word and belongs to one of certain classes (chiefly, `constant` and `defer`'ed words). The value *value* is stored in the appropriate data cell associated with *name*.

Examples:

```
5 constant foo 23 is foo
```

Now `foo` returns 23 instead of 5.

```
' (warm-code is warm-code
```

Assuming that `warm-code` is a deferred word, now `(warm-code` is installed as the "action word" of this deferred word.

`is` is a "state-smart" word. In interpret mode, *name* is "ticked" (see `'`) and the resulting *acf* is passed to `(is`, which does the work of storing *value* in the appropriate location. In compile mode, the word `(is)` is compiled and then the word *name* is compiled. When `(is)` is later executed, it treats the following dictionary cell, containing the *acf* of *name*, as a literal, and passes its value to `(is` who does the right thing.

7.5.3 Arrays

An **array** word accepts the base address of an array of data and an index; it performs the appropriate address arithmetic and returns the address of the array element corresponding to that index.

Many variations are possible. An array word that handles n-dimensional arrays will expect the array element to be specified by an n-tuple of indices. For an array that is to be used exclusively as a lookup table (i.e., read-only access), it may be more convenient to return the entry data itself rather than its address. To avoid proliferation of array words (for different element sizes, numbers of dimensions, ways of initializing the array, etc.), it is more practical to let the user define whatever array words are needed in the application, using a DOES **construct** (see). The current system provides a few array words as examples.

`array` (*n*) *name*
`string-array` *name*
`end-string-array`

`array` is a defining word. It associates with *name* a buffer of *n* entries of Forth-cell size. When *name* is later executed with *index* as an argument, it returns the address of the corresponding entry.

`string-array` is a defining word, defined by the following example

```
          string-array mystrings ( n -- pstr )
        ," a string"
  ," another string"
      end-string-array
```

The strings encountered in the body of the definition are saved in the dictionary. When `mystring` is later executed with *n* as an argument, it will return the address *pstr* of the *n*-th of those strings (counting from 0).

7.5.4 User stacks

`stack:` (*n* —) *name*
`push` (*n user-stack* —)
`pop` (*user-stack* — *n*)
`top@` (*user-stack* — *n*)
`top!` (*n user-stack* —)
`clearstack` (*user-stack* —)
`sdepth` (*user-stack* — *n*)
`?empty` (*user-stack* — *flag*)

The expression

`n stack:` *name*

defines as a new user stack called *name* with *n* bytes of storage allocated. When later executed, *name* leaves the address of the data structure decribing that stack. Note that the number of available items is *n* divided by the size of a cell (4 bytes).

`push` pushes *n* on the given user stack; an error abort occurs if the stack was already full. `pop` pops *n* from this stack; an error abort occurs if the stack waa already empty.

`top@` fetches the number which is on the top of the given user stack without popping it; `top!` replaces the number which was on the top of this stack. In either case, an error occurs if the stack was empty.

`empty?` returns TRUE if the given user stack is empty. `sdepth` returns the number of items that are currently on the given user stack. `clearstack` clears the given user stack.

7.6 Memory

Also see **Memory allocation** (§7.17.2).

7.6.1 Fetch and store

`@` (*addr* — *n*)
`c@` (*addr* — *n*)
`w@` (*addr* — *n*)
`2@` (*addr* — *n1 n2*)

`@` returns the value *n* at *addr*. `2@` returns the value of two consecutive cells starting at *addr*.

`c@` fetches the byte at *addr*, and returns it as a 32-bit stack cell, padding the remaining 24 bits with zeros. Similarly, `w@` fetches a 16-bit word.

`!` (*n addr* —)
`c!` (*n addr* —)
`w!` (*n addr* —)
`2!` (*n1 n2 addr* —)

`!` stores *n* at *addr*. `c!` stores the least significant 8 bits of *n* in the byte at *addr*. Similarly, `w!` stores the least significant 16 bits.

`2!` stores *n1* and *n2* in consecutive locations starting at *addr*.

7.6.2 Block moves

`move` (*addr addr' u* —)
`cmove` (*addr addr' u* —)
`cmove>` (*addr addr' u* —)

In the following words, nothing is moved if u=0.

`move` copies the *u* bytes at *addr* to *addr'*. It performs the correct operation even when the two string overlap: if *addr*>*addr'*, then the first byte of *addr* is moved first, otherwise the last byte is moved first. The *u* bytes remaining at address *addr'* are the same data as was originally at address *addr*.

Depending on the which of *addr* and *addr'* is lower, `move` performs the equivalent of one of the following "dumb" operations.

`cmove` moves *u* bytes from *addr* to *addr'*. The move begins by moving the byte at $addr+u-1$ to $addr'+u-1$ and proceeds to successively lower addresses for *u* bytes. Useful for sliding a string towards higher addresses when the ranges overlap.

`cmove>` moves *u* bytes from *addr* to *addr'*. The byte at *addr* is moved first, proceeding toward high memory.

`fill` (*addr u byte* —)
`blank` (*addr u* —)
`erase` (*addr u* —)

u bytes of memory beginning at *addr* are set to `byte` by `fill`, to 0 by `erase`, and to `bl` by `blank`.

7.6.3 Memory dump

`dump` (*addr len* —)
`du` (*addr* — *addr+64*)

`dump` outputs a formatted hex/ASCII dump of memory, starting at *addr* and for at least *len* bytes. Dump lines are aligned on 16-byte boundaries; a "downarrow" (\/) in the dump header marks the first requested byte (that at *addr*).

`du` is like `dump`, but assumes a length of 64 bytes. Moreover, it leaves the next address after the dump on the stack, so that successive `du`'s will advance through memory.

7.6.4 Memory search

`search`	(*n start end — loc T* | *F*)
`wsearch`	(*n start end — loc T* | *F*)
`tsearch`	(*addr start end — loc T* | *F*)

`search` searches memory between *start* and *end* for a 32-bit word matching *n*. If one is found, it returns the address of the first matching word and TRUE; otherwise, just FALSE is returned. Only word-aligned locations are searched.

`wsearch` likewise searches memory for a 16-bit word matching *n*. `tsearch` searches memory for a tokenized address matching *addr*.

7.7 Data stack and return stack

The **data stack**, usually simply called the **stack**, is the standard channel through which Forth words exchange arguments.

The **return stack** manages the recursive structure of words calling other words; in this fuction, it is not to be tampered with by the user. As long as control remains at the same recursion level, the return stack can be used (with caution!) as a "third hand" for temporarily holding a few data items; this approach has severe limitations and should be avoided by the casual user.

Also see **User stacks** (§7.5.4)

7.7.1 Stack operations

`dup`	(*n — n n*)
`?dup`	(*n — n n* | *0*)
`2dup`	(*a b — a b a b*)
`3dup`	(*a b c — a b c a b c*)
`4dup`	(*a b c d — a b c d a b c d*)
`drop`	(*n —*)
`swap`	(*n1 n2 — n2 n1*)
`over`	(*n1 n2 — n1 n2 n1*)
`tuck`	(*n1 n2 — n2 n1 n2*)
`nip`	(*n1 n2 — n2*)
`rot`	(*n1 n2 n3 — n2 n3 n1*)
`-rot`	(*n1 n2 n3 — n3 n1 n2*)
`pick`	(*+n — n*)
`roll`	(*+n —*)

`dup` duplicates the top stack cell; `?dup` duplicates it only if it is non-zero.

`2dup`, `3dup`, and `4dup` duplicate, respectively, the top 2, 3, or 4 stack cells.

`drop` removes *n* from the stack.

`swap` exchanges the top two stack entries.

`rot` rotates the top three stack entries, bringing the deepest to the top. `-rot` rotates the three entries in the opposite direction, putting the top number underneath the other two.

`over` puts on the stack a copy of the next-to-top element.

`tuck`, equivalent to `swap over`, copies the top of the stack underneath the second item.

`nip` discards the next-to-the-top element of the stack.

`pick` puts on the stack a copy of the *+n*-th stack element, not counting *+n* itself. `0 pick` is equivalent to `dup`; `1 pick` is equivalent to `over`

`roll` The *+n*-th stack value (not counting *+n* itself) transferred to the top of the stack, moving the remaining values into the vacated position. *+n* must be between 0 and the number of elements on the stack minus one. `2 roll` is equivalent to `rot`; `1 roll` is equivalent to `swap`; `0 roll` is a null operation.

Note: For readability's sake, the use of `pick` and `roll` should be minimized.

7.7.2 Stack management

`clear`	(*anything—*)
`.s`	
`(.s`	
`depth`	(*— +n*)
`ps-size`	(*— +n*)
`?enough`	(*n —*)
`sp@`	(*— addr*)
`sp!`	(*addr —*)
`sp0`	(*— addr*)

`clear` clears the stack (by executing `sp0 @ sp!`). `depth` returns the number of entries contained in the stack (not counting *+n* itself).

`.s` displays the contents of the stack in unsigned format in the current base; the top of the stack appears last. "`Empty`" is displayed if the stack is empty. `(.s`, a lower-level building block, does not display "`Empty`".

`ps-size` returns the maximum size in bytes of the stack (also called the "parameter stack").

`?enough` aborts if the stack depth (not including *n*) is less than *n*. A word may use this to make sure that it has been called with enough arguments.

`sp@` returns the address that the top of the stack had just before that word was executed. `sp!` resets to *addr* the address of the top of the stack. `sp0` is a variable which contains the address of the bottom of the stack.

7.7.3 Stack protection

`!csp`
`?csp`

`!csp` remembers the current value of the stack pointer. `?csp` compares the current value with the previouly-remembered one, and aborts if it has changed. Together, they are used to make sure that the stack depth does not change during the compilation of a word.

7.7.4 Return stack

`exit`		C
`unnest`		

When `exit` is encountered within a colon definition, the defined word will return control to the word that called it. This is done by jumping to the return address at the top of the return stack and popping the stack itself. An error condition exists if the top of the return stack does not contain a valid return point. Since a DO–LOOP uses the return stack, `exit` cannot be used within it.

`unnest` is compiled at the end of a colon definition to make the word terminate and return to the caller. `unnest` behaves exactly like `exit`; it is used in this position instead of `exit` so that the decompiler can easily distinguish the syntactical end of the colon definition from an `exit` that is explicitly compiled somewhere in the middle of the definition.

`>r`	(*n —*)	C
`r>`	(*— n*)	C
`r@`	(*— n*)	C

`>r` pushes *n* onto the return stack. `r>` pops the top element of the return stack and returns it as *n*; `r@` returns a copy of the top of the return stack, without popping it.

`rp@`	(*— addr*)
`rp!`	(*addr —*)
`rp0`	(*— addr*)

`rp@` returns the value of a system variable that points to the top of the return stack. `rp!` replaces with *addr* the value of this pointer (this word may only be used inside of words which do not exit, such as `quit`; in `quit`, it is used to clear the return stack with the phrase `rp0 @ rp!`). `rp0` is a user variable which contains the address of the bottom of the return stack.

7.8 Strings

7.8.1 String size

`count`	(*pstr* — *addr* +*n*)	
`+str`	(*pstr* — *addr*)	
`-trailing`	(*addr len* — *addr len'*)	
`comp`	(*addr1 addr2 len* — *n*)	
`csearch`	(*c addr0 addr1* — *loc T* | *F*)	
`substring?`	(*pstr1 pstr2* — *flag*)	

`count` returns the address *addr* of the first character and the number +*n* of characters of the packed string at *pstr*.

`+str` returns the address of the first available dictionary location after the packed string at *pstr*. *addr* is aligned on a 32-bit word boundary.

`-trailing` adjusts the character count *len* of the string beginning at *addr* to exclude trailing spaces, returning the original address *addr* and the new count `len'`. If the entire string consists of spaces, the latter is 0.

`comp` compares the byte arrays starting at *addr1* and *addr2* and continuing for *len* bytes. Returns 0 if the arrays are the same; 1 if the first array is lexicografically "greater" than the second, and −1 in the opposite case.

`csearch` searches memory between *addr0* and *addr1* is searched for a byte matching `c`. If one is found, it returns the address *loc* of the first matching byte and a TRUE flag. Otherwise, it just returns a FALSE flag.

`substring?` returns TRUE if *pstr1* is identical to the leading portion of *pstr2*.

7.8.2 String input

`"`	(— *addr len*) *text*"	†
`(")`	(— *addr len*)	
`["]`	(— *pstr*) *text*"	†
`("s`	(— *pstr*)	
`"temp`	(— *addr*)	

`"` reads text from the input stream and converts it to a string. It is state-smart: When interpreting, it stores the string at address *"temp*, leaving *addr* and *len* on the stack. When compiling, it saves *text* as a string-literal governed by `(")`; when this is later executed, it leaves *addr* and *len* on the stack.

`["]` is like `"`, but saves *text* under a string-literal governed by `("s)` (moreover, it is not state-smart, and is normally used only only during compilation). When the latter is executed, it will leave *pstr* on the stack.

`""`	(— *pstr*) *name*	
`[""]`	(— *pstr*) *name*	†

`""` reads *name* from the input stream and converts it to a packed string, storing it at address `"temp` and leaving this address as *pstr* on the stack. It is an ordinary Forth word: when it occurs inside a colon definition, it is compiled; when later executed, it will accept *name* from the input stream and leave *pstr* on the stack.

`[""]` is an immediate counterpart to `""`; it is not state-smart and should be used only inside a colon definition. It saves *name* as a string-literal governed by `("s)`; when the latter is later executed, it will leave *pstr* on the stack.

These words are convenient to use (in preference to words of the `"` family) when the text to be assimilated is in token form, i.e., has no spaces in it—for example, a filename.

7.8.3 String movement

`pack`	(*addr len pstr* — *pstr*)
`place`	(*addr len pstr* —)
`"cat`	(*pstr0 pstr1* —)
`"copy`	(*pstr0 pstr1* —)
`save-string`	(*pstr* — *pstr'*)
`nullstring`	(— *pstr*)

`pack` copies the string specified by *addr* and length *len* to a packed string at *pstr*, returning *pstr*. `place` does the same, but does not return *pstr*. [Eliminate `place`?] ♠

`"copy` copies the packed string at *pstr0* to *pstr1*.

`"cat` appends the packed string at *pstr0* to the end of the packed string at *pstr1*. For example, after

```
create sbuf 100 allot
"" hello, sbuf "copy
"" world sbuf "cat
```

`sbuf` will contain the packed string '`hello,world`'. The buffer at *pstr0* is assumed to be long enough to hold the concatenation of the two strings.

`save-string` copies the packed string at *pstr* to a different place, and returns the address of that place as *pstr'*. This may be used to save a string returned by `bl word` or `""` so that another string may be immediately collected. Only one level of temporary string storage is provided.

`nullstring` returns the address a packed string of length 0.

7.8.4 String output

`."`	*text*"	†C
`.(`	*text*)	†
`(.")`		
`skipstr`	(— *addr len*)	

`."` is used inside a colon definitions in the form

`."` *text*"

(note that *text* may contain spaces). It saves *text* as a string literal governed by `(.")`. When the latter is executed, the saved text is displayed.

`.(` is like `."`, but displays the text immediately rather than save it. It is useful for issuing messages at compile time (rather than run time).

`skipstr` is used to implement run-time words which govern string literals. The address and length of the following in-line string left on the stack, and the string is skipped. For example, `(.")` is implemented as `skipstr type`.

7.8.5 String parsing

`word`	(*char* — *pstr*) *text*	*
`(word`	(*char* — *pstr*)	
`'word`	(— *pstr*)	
`blword`	(— *pstr*) *text*	
`delimiter`	(— *addr*)	
`>in`	(— *addr*)	
`getcword`	(*pstr delim fd* — *pstr*)	
`getword`	(*pstr fd* — *pstr*)	
`skipcword`	(*char fd* —)	

The present words are concerned with segmenting the input stream by scanning it while looking for particular delimiters. The most commonly sought segments are word tokens—delimited by white space. For the purpose of the present discussion, a *white-space* or *white* character is any character whose ASCII code is less than or equal to `bl` (the code for space), including all control characters. The end of a line is treated like white space; scanning will continue into the the next line if necessary. Given a character *delim* as a delimiter all other characters are *non-delimiter* characters.

`word` converts *text* from the input stream (up to the first occurrence of the delimiting character *char*) into a packed string *pstr*. It defaults to `(word`. `blword` is the same as `bl word`.

If *char* is `bl`, `(word` skips leading white-space characters, then collects non-white characters until another white-space character is encountered. If *char* is not `bl`, `(word` collects characters from the input stream until it encounters an occurence of *char*, without skipping leading occurences of it.

The actual delimiter encountered is left in the user variable `delimiter`. If the end of the input stream was encountered before a delimiter, `delimiter` contains the constant `eof`.

The packed string returned by `(word` resides in the "free" dictionary area a few bytes above `here`. Note that the text interpreter also uses this area.

The variable `'word` stores the address returned by the previous execution of `word`. It allows one to find the name of the last word that was processed by the text interpreter.

`delimiter` is a user variable containing the actual delimiter character which terminated the last call to `word`, `getword`, or `getcword`.

`>in` is a variable which contains the present character offset within the input stream.

`getcword` accepts the next sequence of non-delimiter characters from the file *fd*, placing them as a packed string into memory starting at *pstr*. The pointer *pstr* is left on the stack unchanged. Leading delimiter characters are *not* skipped. If there were no more acceptable characters in the file, the byte at *pstr* will contain zero, representing a string containing no characters.

`getword` works in a similar way, but it skips leading white characters and accepts the next sequence of non-white characters. The white character which terminates the scanned word is consumed and stored in `delimiter`, where it may be inspected if necessary. If there were no more non-white characters in the file, the byte at *pstr* will contain zero and `delimiter` will contain `eof`.

The behavior of `getword` and `getcword` with respect to leading delimiters is consistent with the way that they are actually used in practice. Specifically, `bl word` is used to scan for the next Forth word, and it is desirable to ignore leading spaces. Tabs, newlines, and carriage returns may appear, and the system must be able to deal with them. When `word` is used with some delimiter other than `bl`, the intention is really that leading delimiters should *not* be ignored! For example, when `word` is used to scan for the ')' which terminates a comment, it is desireable to find one immediately in case of the null comment '( )'. Similarly, `."` and `abort"` should work with null strings, as in `." "` and `abort" "`. The usual work-around is to explicitly check the input stream to see if a delimiter is there before calling `word`. This just points to the fact that there should be two different versions of `word`, one which gets non-white characters, ignoring leading white-space, and one which takes an explicit delimiter off the stack and terminates as soon as it encounters a delimiter.

`skipcword` skips characters up to and including the next occurrence of *char* in the file whose descriptor is *fd*. This is used to implement words such as '(', which skips past the next ')'. This approach is superior to the conventional approach of using `ascii ) word drop`, because `skipcword` has no problem with strings longer than 255 bytes, nor does it waste time by storing the skipped characters in a string that will be ignored.

7.9 Vocabularies

7.9.1 Vocabulary creation

`vocabulary`	*name*
`vocs`	(— *n*)
`assembler`	(— *n*)
`disassembler`	(— *n*)
`forth`	(— *n*)
`hidden`	(— *n*)
`keys-forth`	(— *n*)
`system`	(— *n*)
`[assembler]`	(— *n*) †

`vocabulary` defines *name* as a new vocabulary. Subsequent execution of *name* makes it the first vocabulary in the search order. When *name* is made the CURRENT vocabulary, new definitions will be appended to it.

`vocs` displays the names of all the vocabularies in the system (not just the ones in the search order).

The following vocabularies are predefined

`forth`	The primary vocabulary. Initially, it is the CURRENT vocabulary and the first vocabulary in CONTEXT (see `order`).
`assembler`	Contains words for assembling machine code.
`[assembler]`	Like the previous, but immediate.
`disassembler`	Contains helper word for the disassembler.
`hidden`	Contains words which are used to implement other words but are not intended to be used by user programs and may not be documented. They are "hidden" so as not to clutter the dictionary.
`keys-forth`	Contains keystroke definitions for the commandline editor. The definitions in this vocabulary name control keys and escape sequences, and associate those keys with editing commands. The keys used to invoke line editing commands may be changed by changing the definitions in this vocabulary.
`system`	Contains interface words which directly call UNIX system routines.

`over-vocabulary` (*action-acf voc-acf* —)

Scans the vocabulary `voc-acf`. For each word in that vocabulary, the word at `action-acf` is executed, with the link-field address of the vocabulary word on the stack. This word must consume the name-field address. ♠

The following example displays the names of all the words in the `assembler` vocabulary

```
: l.id l>name .id ;
' l.id ' assembler over-vocabulary
```

The same result could be achieved more easily by `assembler words`. `follow` and `another?` provide an alternate way of doing the same thing.

7.9.2 Vocabulary links

The following words are provided in order to make dictionary manipulation utilities independent of the implementation of the dictionary structure. Also see `link>`.

`voc-link`	(— *addr*)
`voc>`	(*alf* — *acf*)
`link@`	(*alf* — *link*)
`link!`	(*link addr* —)
`link,`	(*link* —)

`voc-link` is a user variable containing the address of the last vocabulary created. All the vocabularies are linked together in a linked list whose head is contained in `voc-link`. Each link (which is stored in the body of the vocabulary data structure) points to the next link. The compilation address *acf* of the vocabulary associated with a link `alf` is given by `voc>`.

`link@` returns the address of the word which is linked to the word whose link field address is `alf`.

`link!` stores the address link at *addr* in the correct format for a dictionary link.

`link,` encloses the address *link* into the dictionary in the correct format for a dictionary link.

`#threads` (— *n*)

`#threads` is the number of threads used by the dictionary hashing scheme. Currently `#threads`=4, that is, each vocabulary is represented by 4 linked lists of words.

7.9.3 Vocabulary search order

`order`
`also`
`only`
`previous`
`seal`
`root`
`definitions`
`#vocs` (— *n*)
`context` (— *addr*)
`current` (— *addr*)

The search-order context consists of a *base* vocabulary (usually ROOT) and a *search-order stack* of additional vocabularies. The number of elements on this stack may grow from 0 to a maximum of `#vocs`; after this maximum, if more elements are pushed on the stack the oldest ones fall off. On a word search, the search-order stack is scanned from the top; the base vocabulary is consulted last. A user variable, `context`, points to the top vocabulary of the search stack. A separate user variable, `current`, points to the CURRENT vocabulary—the one to which new definitions are to be added.

The search-order stack is made to grows by `also`, which duplicates the top stack element (if any); it is made to shrink by `previous`, which drops this element. Executing a vocabulary word such as FORTH or ROOT makes the corresponding vocabulary the top of the stack, superceding the previous one (if any).

`order` displays the names of the vocabularies forming the current context. It also displays the CURRENT vocabulary.

ROOT is the "bootstrap" vocabulary, containing nothing but words necessary to extend the search order. Executing `root` makes ROOT the top of the stack; executing `only` makes ROOT the *only* vocabulary in the search-order stack, as well as the base vocabulary.

`seal` deletes all occurrences of ROOT from the search order, thus removing the means to change the search order (except for changes of the CONTEXT vocabulary, which are done by just mentioning a new vocabulary). This is useful in making turnkey systems in which "everything that is not explicitly allowed is forbidden."

`#vocs` is the number of vocabularies that can be at one time in the search order.

`definitions` changes the value of `current` to point to the CONTEXT vocabulary.

7.10 Character I/O

7.10.1 Ascii codes

`ascii` (— *char*) *text* †
`control` (— *char*) *text* †
`bl` (— *char*)
`bs` (— *char*)
`carret` (— *char*)
`eof` (— *char*)
`linefeed` (— *char*)
`newline` (— *char*)
`printable?` (*char* — *flag*)

`ascii` is used in the form

`ascii` *char*

where *text* typically consists of a single character. The ASCII code for the first (or only) character of *text* is returned on the stack (when interpreting) or compiled as a literal (when compiling).

`control` works in analogous way, but produces the ASCII code for the control character corresponding to the typed character. This is but the ASCII code for the typed character taken modulo hex `20` (decimal 32). For example, since `ascii E` returns hex `45`, `control E` returns `5`.

A few words return the ASCII code (or a special code) for selected nonprinting characters, as follows

word	hex	decimal	function
`bl`	20	32	space
`bs`	8	8	backspace
`carret`	0d	13	carriage return
`linefeed`	0a	10	line feed
`newline`	0a	10	"new line"
`eof`	ffff	-1	end-of-file

`printable` returns TRUE if the character *char* is printable. Printable characters are those between hex `40` and `7e` inclusive.

7.10.2 Character input

`key` (— *char*) *
`(key` (— *char*) *
`sys-key` (— *char*)
`key?` (— *flag*) *
`sys-(key?` (— *flag*)

`key` reads the next ASCII character from the keyboard. If no character has been typed since `key` was last executed, `key` waits until a new character is typed. Characters received by `key` are not displayed. `key` is deferred for convenience in reconfiguring the system; it defaults to `(key`, which is itself deferred and defaults to `sys-key`. The latter accepts all ASCII characters; control characters are not processed by the system for any editing purpose. [Are some control character trapped by the system?] ♠

`key?` returns `true` if a character has been typed on the keyboard since the last time that `key` was executed. The keyboard character is not consumed. It is a deferred word, defaulting to `sys-(key?`.

Also see **interactive**.

7.10.3 Character output

`emit` (*char* —) *
`emit1` (*char* —)
`(emit` (*char* —) *
`sys-emit` (*char* —)

The deferred word `emit` transfers the character *char* to the output stream. It defaults to `emit1`, which updates `#out` to keep track of the cursor position and then calls `(emit` to output the character. The latter is also deferred, and defaults to `sys-emit`.

See Display control (§??) for individual characters and character sequences. Also see ASCII codes (§??).

7.11 Line I/O

7.11.1 Line input

`expect` (*addr +n —*) *
`history-expect` (*addr +n —*)
`sys-expect` (*addr +n —*)
`(newexpect` (*addr +n —*)
`span` (*— addr*)

`expect` receives a line of characters from the keyboard, performing line editing as the characters are typed, and stores them into memory. The transfer begins at *addr* proceeding towards higher addresses one byte at a time until either a RETURN is received or *+n* characters have been transferred. The RETURN is not stored into memory. No characters are received or transferred if *+n* is zero. All characters actually received and stored into memory are displayed. Though the Forth 83 Standard specifies that the RETURN character be displayed as a space, here it is displayed as a RETURN.

`span` is a user variable containing the count of characters actually received and stored by the last execution of `expect`.

The default implementation of `expect` is `history-expect`; this calls `sys-expect` when reading a file, and `(newexpect` (in the `hidden` vocabulary) in interactive mode. The latter provides extensive EMACS-like line editing, with the ability to add, delete, or change characters anywhere in the line being typed. It has a history mechanism which allows the user to retrieve and edit a number of previously-typed lines, and a command completion feature which can be used to reduce the amount of typing necessary.

`tib` (*— addr*)
`#tib` (*— addr*)
`'tib` (*— addr*)

`tib` is the address of the text input buffer, used to hold characters when the input stream is coming from the current input device. The capacity of text input buffer is 80 characters.

`#tib` is the address of a user variable containing the length of the text currently in the text input buffer. `#tib` is accessed by `word` when input is coming from the keyboard.

`'tib` is a user variable containing the address of the text input buffer. One can temporarily change the text input buffer by modifying `'tib`.

7.11.2 Line output

`type` (*addr +n —*) *
`type1` (*addr +n —*)
`(type` (*addr +n —*) *
`sys-type` (*addr +n —*)
`>type` (*addr +n —*)
`".` (*pstr —*)

`type` transfers to the current output stream *+n* consecutive characters from memory, starting at *addr*. No action is taken if *+n*=0. It is a deferred word. It defaults to `type1`, which increments `#out` and calls `(type`. The latter defaults to `sys-type` (cf. `emit`).

`>type` transfers *+n* consecutive characters to the current output stream starting from memory address *addr*. The characters are transferred to the PAD before they are sent to the output stream.

`".` sends the packed string at *pstr* to the output stream; it is equivalent to `count type`.

7.12 Number I/O

7.12.1 Number base

`base` (*— addr*)
`binary`
`octal`
`decimal`
`hex`
`.d` (*n —*)
`.h` (*n —*)
`b#` (*— n*) *number* †
`d#` (*— n*) *number* †
`h#` (*— n*) *number* †

`base` is a user variable containing the current radix for interpretation and conversion conversion of numeric input/output. The range for this radix is [2..40]. The corresponding digits are, besides `0`, the range `1` through `9` followed by the range `a` through `~`.

`binary` sets `base` to two; `octal` sets it to eight; `decimal`, to ten; `hex`, to sixteen.

`.d` displays *n* in decimal without affecting the value of `base`; similarly, `.h` displays *n* in hex.

The words `b#`, `d#`, and `h#` interpret the next word in the input stream with the numeric base temporarily 2, 10, or 16; this word may either be a number or an executable word. The value of `base` is not permanently altered. For example, while working in base ten, `h# 100 10 +` will be read as 266.

If a long segment of code needs to use, say, hexadecimal numerals, the recommended sequence is

```
base @      \ save current base on stack
hex         \ set base to 16
...         \ code segment
base !      \ restore original base
```

For isolated numerals, or when it is desired to explicitly indicate the desired base, use `b#`, `d#`, or `h#`.

7.12.2 Number, input conversion

Numeric input conversion (from a text string to a number's internal representation).

`number` (*pstr — n*) *
`(number` (*pstr — n*)
`number?` (*pstr — n flag*)
`convert` (*+n0 addr0 — +n1 addr1*)
`digit` (*char base — +n T | char F*)
`dpl` (*— addr*)

`number` converts the string *pstr* to a signed 32-bit number *n*, using the value of `base`; it aborts if the string contains unconvertible characters. It is deferred, defaulting to `(number`.

`number?` acts like `number`, but returns TRUE if the string contained only convertible characters (digits in the current base, minus, and period). If an unconvertible characters is found, *flag* is FALSE, and *n* is the number that was collected up to that point. For example, if the string was `123xyzn`, *n*=123 and *flag*=FALSE.

`convert` performs numerical conversion of a text string. Assumes that *+n0* is the partial result of a conversion ending with the character at *addr0*. Continues the conversion from *addr0*+1. For each digit the partial result of the conversion is multiplied by `base` and incremented by the value of the digit. Conversion continues until an unconvertible character is encountered. *+n1* is the result of the conversion; *addr1* the location of the first unconvertible character.

`digit` returns the numeric value of *char* in the specified base, followed by a TRUE flag; if *char* is not a digit, it returns the character itself and FALSE. Appropriate characters are 0–9, a–z, depending on `base`.

`dpl` is a variable containing information about the last number scanned by the text interpreter. If `dpl` contains `-1`, the number did not contain a decimal point. Otherwise, `dpl` represents the number to digits to the right of the decimal point in the last number scanned.

7.12.3 Number literals

`literal`	(*n* —)	†C
`(lit)`	(— *n*)	
`literal?`	(*pstr* — *pstr F* | *n T*)	*
`(literal?`	(*pstr* — *pstr F* | *n T*)	

`literal` is typically used in the form

`[` *number* `] literal`

It saves the value *n* of the expression *number* as a literal governed by `(lit)`. When the latter is executed, *n* will be left on the stack.

`literal?` is executed from `"compile` if the word under consideration was not found in the dictionary. If the string *pstr* is a recognizable number *n*, it returns *n* and TRUE, otherwise returns the original string and FALSE. This is a deferred word, defaulting to `(literal?`.

See `do-literal`.

7.12.4 Number, output conversion

Numeric output conversion (from internal representation of a number to a text string).

`#`	(*+n* — *+n'*)
`<#`	
`#s`	(*+n* — *0*)
`#>`	(*n* — *addr len*)
`hold`	(*char* —)
`sign`	(*n* —)

`<#` initializes the conversion, starting from the least significant digit and storing characters in right-to-left order in a temporary buffer (`#-buf`). The number base may change during the conversion; thus, each digit may refer to a different base.

`#` produces one more digit. It divides *+n* divided by *base*, converts the resulting number to an ASCII character, and appends it to the buffer. *+n'* is the quotient and is maintained for further processing.

`#s` is like `#`, but converts *all* of *+n*, appending digits into the buffer until the quotient is zero. A single zero is added to the buffer if the number was initially zero.

`#>` terminates the numeric output conversion, dropping *n*. The result of the conversion is a character string at *addr*, of length *len*, suitable for `type`.

During the conversion, `hold` may be called to append a character *char* in the buffer. `sign` appends a - (minus sign) if *n* is negative.

7.12.5 Number printout

`.`	(*n* —)
`s.`	(*n* —)
`(.)`	(*n* — *addr len*)
`.r`	(*n +n* —)
`u.`	(*u* —)
`(u.)`	(*u* — *addr len*)
`u.r`	(*u +n* —)
`?`	(*addr* —)

`.` displays *n* in *free field format*, i.e., the absolute value of *n* in the current base, preceded by a minus sign if *n* is negative and followed by a space.

Note: If the base is hex, `.` displays the number in *unsigned* format (since signed hex display is hardly ever desired). A more basic version of this word, namely `s.`, *always* displays *n* as a signed number.

`(.)` gives *addr* and *len* of the string (but without a space at the end) that `s.` would display.

`u.` displays *u* as an unsigned number in a free-field format. `(u.)` gives *addr* and *len* of the string (but without a space at the end) that `u.` would display.

With `.r` and `u.r`, *n* is displayed right aligned in a field *+n* characters wide, respectively in signed and unsigned form. In this implementation, if the number of characters required to display *u*is greater than *+n*, all the characters will be displayed, making the resulting field larger than *+n*.

`?` fetches the value at *addr* (see `@`) and then displays it using '`.`'.

7.13 Display formatting

7.13.1 Display formatting

`cr`		*
`(cr`		
`lf`		
`(lf`		
`??cr`		
`?cr`		
`space`		
`spaces`	(*+n* —)	
`backspaces`	(*+n* —)	
`?line`	(*+n* —)	
`#columns`	(— *n*)	
`#line`	(— *addr*)	
`#lines`	(— *n*)	*
`(#lines`	(— *n*)	
`#out`	(— *addr*)	
`.tab`	(*len* —)	
`tabstops`	(— *addr*)	
`to-column`	(*column* —)	
`?to-column`	(*len col* —)	
`at`	(*line column* —)	
`insert-char`	(*char* —)	
`insert-line`		
`delete-char`		
`delete-line`		
`kill-line`		
`kill-screen`		
`erase-screen`		
`dark`		
`light`		
`beep`		
`up`		
`down`		
`left`		
`right`		
`lmargin`	(— *addr*)	
`rmargin`	(— *addr*)	

These words are concerned with display control.

`cr` is used to terminate the line on the display and go to the next line. It defaults to `crlf`, which sends a carriage return and line feed to the output stream, clears `#out`, and adds 1 to `#line`.

`(cr` sends just a carriage return character (hex `0d`).

`lf` sends a linefeed character on the output stream and adds 1 to the variable `#line` to indicate that one more line has been printed on the terminal.

`(lf` just sends a linefeed character (hex `0a`).

`??cr` starts a new line if the current line is not empty. This is used to ensure that output begins on a new line, without causing an unnecessary blank line.

`?cr` starts a new line, spacing over to the left margin, if the length of the current line is past the right margin. Equivalent to `0 ?line`.

`?line` starts a new line if adding n more characters to the current output line would cause the current line length to go past the right margin.

`#line` is a user variable which is incremented each time that `cr` is executed. It may be used for control of pagination.

`#columns` returns the number of columns of text that are displayable on the screen. [It probably has a bug. Returns 128.] ♠

`#lines` returns the number *n* of lines of text that are displayable on the screen. It defaults to `(#lines`, which determines the number of lines based on the current font parameters.

`#out` is a user variable containing the current column position on the output device. This is updated by `emit1` and some other words which modify the cursor position. It is used for display formatting.

`.tab` advances to the column on the output device that is the next multiple of the value contained in the variable `tabstops`. If there is not enough room on the current line for *len* characters, a new line is started. This is used to produce listings where successive items begin in neat columns.

`to-columns` transmits enough spaces to the output stream to move to the indicated column. If the current line is already at or past that column, only 1 space is transmitted. `?to-columns` is similar, but if printing *len* characters would cause the line length to exceed the right margin, a new line is started at the left margin.

`at` moves the cursor to the indicated line in the range [0..`#lines`) and column in the range [0..`#columns`) on the screen.

`space` **Warning:** The original Forth meaning of this word (send a space character to the output stream) is pre-empted by a StepForth word.

`spaces` sends *+n* spaces to the current output device. `backspaces` sends *+n* backspace characters (hex 8).

`inser-char` insert *char* at the cursor position, moving the rest of the line to the right to make room.

`insert-line` inserts a new line on the screen at the cursor position. The line that was underneath the cursor, and all lines below, are moved down to make room. The bottom line falls off the bottom of the screen and is lost.

`delete-char` deletes the character under the cursor from the screen, moving all the characters to the right leftward to fill in the gap.

`delete-line` deletes the current line from the screen, moving all the lines below it up to fill in the gap.

`kill-line` clears the current line from the cursor position to the right edge of the screen.

`kill-screen` clears the screen from the cursor position to the bottom of the screen.

`erase-screen` clears the entire screen and the cursor is left at the upper-left corner.

After `dark` is executed, ensuing text is displayed in *inverse* video mode; after `light`, in *normal* video mode. With the default color tables, inverse gives white text on black background, and normal gives black on white.

(In F83, `dark` is used to erase the screen; in this system this is done by `erase-screen`.)

`beep` transmits a bell character (hex 7) to the output stream, which causes the system to make a sound.

`up` moves the cursor up one line, unless it is already on the top line. `down` moves the cursor down one line, unless it is already on the bottom line.

`left` moves the cursor left one column, unless it is already in the leftmost column. `right` moves it right one column, unless it is already in the rightmost column.

`lmargin` is a variable containing the column number of the left margin. When the output formatting commands cause the automatic generation of a new line, the new line begins in the column number contained herein. `rmargin` is a variable containing the column number of the right margin. The output formatting commands cause the automatic generation of a new line when the current line length exceeds the column number contained herein.

7.14 Files

See §?? for loading Forth code from a file.

7.14.1 File, opening and closing

`open`	(*pstr mode — fd*)
`#fds`	(— *n*)
`new-file`	(*pstr* —)
`write-open`	(*pstr* —)
`read-open`	(*pstr* —)
`append-open`	(*pstr* —)
`writing`	*name*
`reading`	*name*
`appending`	*name*
`to-file`	*name*
`append-to-file`	*name*
`read`	(— *0*)
`write`	(— *1*)
`modify`	(— *2*)
`close`	(*fd* —)
`close-files`	
`make`	(*pstr — flag*)
`file-exists?`	(*pstr — flag*)
`file-protection`	(— *addr*)
`ifd`	(— *addr*)
`ofd`	(— *addr*)
`file-output`	
`undo-file-output`	

`open` attempts to open the filed named by *pstr*, with an access mode (READ, WRITE, or MODIFY) indicated by the access-mode code *mode*. If the open operation is successful, it returns a file-descriptor *fd* different from 0, otherwise it returns 0. A succesful open sets the file pointer to 0, so that the next sequential access will occur at the beginning of the file.

The constant `#fds` returns the number of file descriptors available, currently 8. A file should be closed when it is no longer being used, so that the file descriptor may be reused.

The three access-mode codes are given by the constants `read`=0, `write`=1, and `modify`=2.[`read` is pre-empted by a homonymous word] ♠

`reading` open for reading the file *name*. If the file is successfully opened, its file descriptor is stored in `ifd`; otherwise, a message is displayed and the operation aborts. `read-open` is like `reading`, but takes the file name from the packed string *pstr*.

`writing` creates and opens for writing a file called *name*. If a file with that name already exists, the old file is destroyed. If the file is successfully opened, its file descriptor is stored in `ofd`; otherwise, a message is displayed and the operation aborts. `new-file` is like `writing`, but takes the file name from the packed string *pstr*. `write-open` does the same, but aborts if the file does not already exist.

`appending` opens a file so that bytes subsequently written will be added to the end; if the file doesn't exist, it is created. For the rest, it works like `writing`. `append-open` takes the file name from *pstr*.

`to-file` creates a new file with the given name, and redirects the output stream to that file. With `append-to-file`, the output stream is redirected and appended to the end of the named file (if the file does not exist it is created). In both case, the output stream is restored to the terminal just before the interpreter prompts for a new line of input. For example, to decompile the words `words` and `order` and put the output in a file called `words.lst`, one would write

```
to-file words.lst  see words
append-to-file words.lst  see order
```

`close` closes the file described by *fd* ar releases that file descriptor. `close-files` closes all the currently open files.

`make` is the basic file-creating word. It creates a new file created with the name given by *pstr*; it returns TRUE if the file is successfully created. If a file with that name already exists, the old file is destroyed.

`file-exists?` returns TRUE if the file named by *pstr* already exists.

`file-protection` is a variable containing the attributes for newly-created files. When Forth creates a file with `make` or any other word that uses `make`, the contents of `file-protection` are the attributes that are assigned to the file. The standard octal UNIX values (used by `chmod`, etc.) are used. Note that the UNIX `umask` value may alter the protection level.

The variables `ifd` and `ofd` record the file descriptors of the file last opened for input (by `reading` or `read-open`) or output (by `writing`, `appending`, `new-file`, or `write-open`. They exists purely for convenience: none of the primitive file system operations use them. In many applications involving files, no more than one input and one output file are needed. In these cases, `ifd` and `ofd` provide convenient predefined variables in which to store the file descriptors.

♠ See `string-fopen`.

[Describe `file-output`, `undo-file-output`. Currently they are in the `hidden` vocabulary.]

7.14.2 File, read and write

`fgetc` (*fd* — *byte*)
`fgets` (*addr len fd* — *len'*)
`fflush` (*fd* —)
`filec@` (*l.addr fd* — *byte*)
`filec!` (*byte addr fd* —)
`fseek` (*addr fd* —)
`+fseek` (*n fd* —)
`fputc` (*byte fd* —)
`fputs` (*addr len fd* —)
`fseek-from-end` (*addr fd* —)
`ftell` (*fd* — *addr*)
`fsize` (*fd* — *size*)

The present words deal with things that can be done with an open file *fd* (more verbosely, a file identified by the file descriptor *fd*). See `open` for words concerned with opening and closing files.

`fflush` writes to disk (or whatever output device) any characters of the file *fd* that are still left in the file buffer.

`fgetc` returns a byte from the file *fd*, from the position specified by the file pointer. The file pointer is then incremented. If there are no more bytes in the file, it returns `eof`.

`fgets` attempts to transfer *len* bytes from the file *fd* to memory, starting at *addr*. Fewer than *len* bytes may be transfered if not enough bytes are left in the file. `len'` is the number actually transfered; if 0, it means that the file pointer was already at the end of the file.

`filec!` stores `byte` into the file *fd*, at the position indicated by *addr*; the file pointer is moved to *addr*+1. `filec@` gets `byte` from the file, and is otherwise analogous.

`+fseek` adds *n* to the file pointer of the file *fd*. This is a good way to move around in a file without actually reading the bytes.

`fputc` writes `byte` (only the 8 low bits) to the file *fd*, at the position indicated by its file pointer, which is then incremented.

`fputs` writes *len* bytes from memory starting at *addr* to the file *fd*, starting from the current file pointer position. The file pointer is left pointing to the byte just past the last byte put in the file.

`fseek` moves the file pointer for file *fd* to the position indicated by *addr*.

`seek-from-end` moves the file pointer for file *fd* to a position *addr* bytes from the end of the file.

`ftell` returns the current position of the file pointer for file *fd*.

`fsize` returns the size in bytes of file *fd*

7.15 Input stream

7.15.1 Interactive

Forth code can be loaded from a file, or can be entered from a terminal and executed interactively. A number of words are provided for functions that are meaningful only in the interactive context.

`interactive?` (— *flag*) *
`(interactive?` (— *flag*)
`exit?` (— *flag*)

`interactive` returns TRUE if the input stream is coming from the keyboard intead of from a disk file. This is used, for example, to control the type of message that is printed in case of an error, or to turning off prompting if the input comes form a file. It is a deferred word, defaulting to `(interactive?`, which returns TRUE if the input stream is coming from the keyboard through `expect`.

`exit?` checks the keyboard to see if anything has been typed. If no character has been typed, it returns FALSE. If the character `q` has been typed, it returns TRUE, and the calling word should stop any output in progress. If any other character has been typed, the message "`More?`" is displayed and `exit?` waits for another key. If this is `q`, `exit?` returns TRUE and the caller should stop; otherwise, it returns FALSE.

This allows convenient interruption and resumption of listings.

`prompt` *
`(prompt`
`status`

The deferred word `prompt` is executed before input is read from the input stream. It is conventionally used to display a prompt. It defaults to `(prompt`, which prints the prompt `ok` if the system is in interpret state, and `]` if in compile state.

`status` is called whenever the system prompts for more input; it defaults to `noop`. The system can be made to do convenient thing before every prompt by installing an appropriate word in `status`. For example,

```
' .s is status
```

causes the stack contents to be printed before every prompt.

`ok`

Does nothing. Defining `ok` to do nothing means that a previous Forth command line may be picked up with an editor or a mouse and fed back to the Forth interpreter without having to worry about stripping off an `ok` prompt that may appear in it.

`error-output` *
`restore-output` *

`error-output` is called before an error message is printed. `restore-output` is called after the message is printed. The intention is to select a separate error stream for the message.

7.15.2 Loading from files

`fload` *filename*
`"load` (*pstr* —)
`(load` (*fd* —)
`needs` *name filename*
♠ `fexit`

[Why not simply `load`?]

`fload` saves the file descriptor for the current input stream, and starts taking the interpreter's input from the file *filename*. When the end of that file is reached, the previous interpreter input source is restored.

With `"load`, the filename of the file to be loaded is taken from memory (typically from the stack) as a packed string at *pstr*, instead of from the input stream.

With `(load`, the file to be loaded is identified by the file descriptor *fd*; after loading, the file is closed. This may be used to load a file starting somewhere in the middle, or to load from some type of file other than a disk file, as is done with the input stream which uses `expect` as a kind of file.

`needs` loads the file *filename* only if the word *name* is not already defined. Typically, *filename* is a file containing the source code for *name*.

`fexit` makes the interpreter ignore the rest of the input file in which it appears. This is achieved by seeking to the end of the input file, so that the next attempt to read it will return end-of-file.

7.15.3 Comments

`(` *text*`)` †
`(s` *text*`)` †
`\` *rest-of-line*↩ †

After the word `(`, everything in the input stream up to and including the terminating right parenthesis is treated as a comment and ignored. There are no limits on the length of *text*.

The word `(s` has an identical effect; it may be used to distinguish stack comments from other comments.

After the word `\`, the remainder of the current input line is ignored.

7.16 Error recovery

Also see `error-output`, `restore-output`.

7.16.1 Abort

`abort` *C
`(abort`
`abort"` *text*" †C
`(abort")` ((*abort*"")) `(abort"")`
flag —
`where` *
`(where`
`?missing` (*flag* —)
`?stack`

`abort` is called to stop the execution of a word, usually as a result of an error condition. It defaults to `(abort`. This clears the stack and performs the function of `quit`; no message is printed. A user-defined error handler may be installed instead of `(abort`, giving the application better control over errors.

Used during compilation, `abort"` saves the error message *text* in a string literal governed by `(abort")`. When the latter is executed, if *flag* is TRUE it displays *text* and then performs an error recovery sequence, including the function of `abort`; if *flag* is FALSE, execution continues.

`where` is a deferred word which displays a message giving the current location in the input stream. Called from `abort` and `abort"`.

`(where` is the default for `where`. If the input stream is interactive (****see) no message is given. If the input comes from a file, it displays the name of the word being compiled (if interpreting) or the most recently defined word (if interpreting).

If *flag* is TRUE, `?missing` aborts and prints the name of the last word read from the input stream, followed by a question mark.

`?stack` aborts if the stack pointer is outside of the space allocated to the stack, printing either "`Stack Underflow`" or "`Stack Overflow`".

7.17 Operating system interface

7.17.1 Shell commands

`sh` *rest-of-line*↩
`cd` *rest-of-line*↩
`cp` *rest-of-line*↩
`df` *rest-of-line*↩
`ls` *rest-of-line*↩
`more` *rest-of-line*↩
`mv` *rest-of-line*↩
`pwd` *rest-of-line*↩
`rm` *rest-of-line*↩
`command:` *name0 name1*
`(cd` (*pstr* —)
`(delete` (*pstr* —)
`(files` (*pstr* —)
`(md` (*pstr* —)
`(rd` (*pstr* —)
`(exec` (*pstr0 pstr1* —)
`(rename` (*pstr0 pstr1* —)
`(size` (*pstr* —)
`child-status` (— *addr*) ♠

[`cd` and `df` happen to be hex numbers!]

`sh` is used in the form

`sh` *rest-of-line*↩

The text *rest-of-line* is executed as a UNIX shell command.

Individual words are provided for the most common shell commands; with these words, any text in *rest-of-line* further qualifies a command.

`cd`	change directory
`cp`	copy files to files or directories
`df`	shows free space on drive
`ls`	display a directory listing
`more`	display the contents of the file, one page at a time
`mv`	move or rename files
`rm`	remove files from disk
`pwd`	display current path

New command words can be added to this list by means of the defining word `command:`. *name0* is the name of the new word, *name1* the name of the desired shell command. When executed, *name0* will issue a shell line consisting of *name1* followed by the remainder of the input-stream line.

Forth system words for some of the above functions are

`(cd`	Change the current directory to that named by `pstr`.
`(delete`	Delete the file named by `pstr`.
`(exec`	Execute the file named by `pstr`, using the string *pstr'* as the program's command line. When the program exits, control returns to Forth.
`(files`	Display all filenames matching the pattern `pstr`.
`(md`	Create a new directory named `pstr`.
`(rd`	Remove the directory named `pstr`.
`(rename`	Rename a file from `pstr` to `pstr'`.
`(size`	Display name and size of all files whose names match `pstr`.

`child-status` is a user variable containing the exit status returned by the UNIX command that was most recently executed from Forth.

7.17.2 Memory allocation

`alloc-mem` (*+n — addr*)
`free-mem` (*addr +n —*)
`init-malloc`
`next-free-mem` (*— addr*)

`alloc-mem` allocates *+n* bytes of memory in a space outside of the dictionary, returning the starting address *addr* of the allocated area. The contents of allocated area is not saved when a Forth system image is made on disk.

`free-mem` releases *+n* bytes of previously allocated memeory starting at *addr*. In the current implementation, only the most-recently-allocated buffer may be freed.

`init-malloc` initializes the simple memory allocator invoked by `alloc-mem` and `free-mem`. It is automatically executed when Forth is first started, if executed again subsequently, the system may crash.

`next-free-mem` is a variable containing the address of the end of free memory, used by `alloc-mem` to allocate more memory. The dictionary grows upward toward the address contained in `next-free-mem`, and memory is allocated by `alloc-mem` by subracting the allocated amount from the address contained in `next-free-mem`.

7.18 System startup

7.18.1 System startup

`origin` (*— addr*)

Returns the address of the start of the Forth system. This address may change from run to run.

`title`
`.version`

`title` displays the name and version of the Forth system. Prints the version number of this Forth system. `.version` displays the version.

`warm`
`warm-hook` *
`(warm-hook`

`warm` is called when the Forth system is restarted after having been interrupted. It executes `warm-hook` and then calls `quit` to return control to the text interpreter.

A warm start differs from a cold start in that a cold start happens only once, when the Forth system is first executed. A cold start brings up the Forth system from scratch, reallocating data structures, buffers, etc. Warm starts can happen many times after the system is started, and generally do not greatly change the state of the Forth system. For example, definitions that have recently been added to the dictionary and not saved to disk are not destroyed by a warm start. Hardware errors such as Bus Error, Illegal Instruction, and Address Error will cause a warm start.

The deferred word `warm-hook` does any cleanup necessary after a hardware exception. It defaults to `(warm-hook`, which among other things clears the stack. It can be reassigned to an application-specific exception handler.

`cold`
`cold-hook` *
`(cold-hook`

`cold` is called once when the Forth system is first starting up, to initialize various aspects of the system.

`cold-hook` is the last thing called by `cold` before starting to interpret commands from the keyboard.

`(cold-hook` is the word installed in the deferred word `cold-hook`. It is redefined in the system several times; each redefinition calls the previous version of the same word. This allows various subsystems to be initialized without requiring them to be compiled together. Each version of `(cold-hook`, after calling the previous version, performs whatever initialization is appropriate for the system with which it is associated. This mechanism may also be used to make an application automatically start itself.

Packages loaded on top of the kernel can arrange to have their own initialization code performed when the Forth system is started by redefining `(cold-hook` and installing the new version in `cold-hook` as follows

```
: (cold-hook (cold-hook new-init ;
' (cold-hook is cold-hook
```

`save-forth` (*pstr —*)

Used in the form

```
"" filename.exe save-forth
```

it saves into a file an executable image of the current Forth system configuration. From UNIX, one can later re-enter that same Forth image with

```
forth filename.exe
```

`bye` *
`(bye`

`bye` exits from Forth back to whatever program called it. It defaults to `(bye`, which performs some system-dependent cleanup before leaving Forth.

If an application needs to execute some further cleanup just before the system exits, it may define a *cleanup* word and attach it to `bye` as follows

: `(bye` *cleanup* `(bye ; ' (bye is bye`

7.19 Implementation-dependent parameters

7.19.1 Storage size

`/n` (*— value*)
`/l` (*— value*)
`/w` (*— value*)
`/token` (*— value*)
`/link` (*— value*)
`/fbuf` (*— value*)
`/fd` (*— value*)
`/n*` (*n — n'*)
`/l*` (*n — n'*)
`/w*` (*n — n'*)
`ca+` (*addr n — addr'*)
`wa+` (*addr n — addr'*)

`la+` (*addr n — addr'*)
`na+` (*addr n — addr'*)
`ca1+` (*addr — addr'*)
`wa1+` (*addr — addr'*)
`la1+` (*addr — addr'*)
`na1+` (*addr — addr'*)
`ta1+` (*addr — addr'*)
`#align` (— *n*)
`align`
`aligned` (*addr — addr'*)

The present words return or make use of the values of certain system implementation constants (such as word size, address size, etc.).

Words that return the value of a constant:

word	value	meaning
`/n`	4	Bytes in a normal word.
`/l`	4	Bytes in a long word.
`/w`	2	Bytes in a half word.
`/token`	4	Bytes in an address token.
`/link`	4	Bytes in a dictionary link.
`/fbuf`	512	Bytes in a file buffer.
`/fd`	60	Bytes in a file descriptor.

Words that multiply *n* by a constant:

word	action
`/n*`	×`/n`
`/l*`	×`/l`
`/w*`	×`/w`

Words that increment a given address *addr* by a certain constant or by *n* times that constant:

word	increment
`/na1+`	`/n`
`/ca1+`	`1`
`/wa1+`	`/w`
`/la1+`	`/l`
`/ta1+`	`/token`
`/na+`	`n`×`/n`
`/ca+`	`n`×`/1`
`/wa+`	`n`×`/w`
`/la+`	`n`×`/l`

`#align` returns the address granularity for the <CPU, which is 4 for a SPARCstation. `align` adjusts the dictionary pointer to the next machine-word boundary; any bytes skipped are filled with 0. `aligned` returns the address *addr'* that would be obtained by adjusting *addr* as in `align`.

7.19.2 Tokenized addresses

`token@` (*addr1 — addr2*)
`token!` (*addr2 addr1 —*)
`token,` (*addr —*)

A *token* is an implementation-dependent representation of an address when it is stored in memory. In this implementation, a token is a 32-bit absolute address.

`token@` fetches a tokenized address from *addr1* and returns it in ordinary form as *addr2*.

`token!` stores the address *addr2* at address *addr1* as a token.

`token,` encloses the tokenized version of *addr* in the dictionary.

7.20 Interpretation

7.20.1 Interpreter

`interpret` *
`(interpret`

`interpret` begins text interpretation at the current postion within the input stream, continuing until this is exhausted. It is deferred, and defaults to `(interpret`.

`"compile` (*pstr —*)

`"compile` will treat the packed string at *pstr* as the name of a Forth word. According to the value of `state` and the immediate attribute of the word, it will either interpret or compile this word. ♠ (A better name for this would be `"interpret`.)

`eval` (*addr len —*)

Interpret Forth source code from a memory buffer beginning at *addr* and continuing for *len* bytes. This works by setting up a file descriptor that describes the memory buffer, and then using `(load` to interpret that fake file. Same as `string-load`.

`execute` (*acf —*)
`perform` (*addr-of-acf —*)

`execute` executes the word definition at *acf*. An error condition exists if *acf* is not a compilation address.

`perform` is equivalent to `token@ execute`.

`string-load` (*addr len —*)
`string-fopen` (*addr len — fd*)

Given the address and length of a memory string, `string-fopen` returns a file descriptor *fd* for this string treated as a file.

`string-load` allows one to interpret a text string in memory. The string is specified by *addr* and *len*; a file descriptor for this string is obtained, and then the file is loaded using `(load`. Same as `eval`.

`do-defined` (*acf n — ??*) *
`do-literal` (*n — ??*) *
`do-undefined` (*pstr —*) *
`interpret-do-defined` (*acf n — ??*) I
`interpret-do-literal` (*n — n*) I
`interpret-do-undefined` (*pstr —*) I
`compile-do-defined` (*acf n —*) C
`compile-do-literal` (*l —*) C
`compile-do-undefined` (*pstr —*) C

When the text interpreter finds a new token in the input stream, it will give it to `"compile` for execution. `"compile` will first look the token up in the dictionary. If it is defined, it will execute `do-defined` giving to it as parameters the *acf* of the word and n=1 if the word is immediate or n=−1 if it is not. If it not defined but can be read as a number, it will execute `do-literal`, passing the number itself as a parameter *n*. Otherwise it will execute `do-undefined`, passing the address *pstr* of the packed string representing the word itself. These three `do-...` words are deferred; in interpreting mode they are implemented respectively as `interpret-do-defined`, `interpret-do-literal`, and `interpret-do-undefined`; in compiling mode, as `compile-do-defined`, `compile-do-literal`, and `compile-do-undefined`.

Depending on the mode (interpret or compile), `do-compile` will either directly execute the word or compile it in the dictionary; `do-literal` will either push the number on the stack or compile it as a literal; `do-undefined` will print the word followed by'`?`' and then either abort or compile the word `lose` and continue with the compilation.

The `lose` provision (instead of aborting) allows one to catch many errors with a single compilation pass. If an error was just due to a mistyped word, one can use `patch` to replace the compiled `lose` with the correct word, without recompiling. At execute time, any occurrence of `lose` still left in the compiled code will print the message "`Undefined word encountered`" and abort.

`quit` *
`(quit`

`quit` is a deferred word which interprets commands from the keyboard. It defaults to `(quit`, which clears the return stack, sets interpret state, accepts new input from the current input device, and begins text interpretation. No message is displayed.

`quit` is called by `abort` to resume text interpretation.

7.20.2 Does constructs

The DOES construct allow the Forth user to specify the execution-time action of the words that will be constructed by a given defining word. In the `does>` form of this construct, this action is specified in Forth; in the `code;` form (see *code*), the action is specified in machine language.

`does>` (— *addr*) †C
`dodoesaddr` (— *addr*)

`does>` defines the execution-time action of a word created by a high-level defining word. It is used in the form:

`:` *namex* `...` `create ...` `does> ...` `;`

where *namex* is the defining word, which uses `create` or an equivalent word to make a new dictionary entry. At execution time, a word defined by *namex* will execute the code following `does>`; execution starts with the address left by the `create`'d word on the stack.

`dodoesaddr` is a variable containing the address of the `dodoes` routine. This is used by `does>` to assemble a JSR instruction to `dodoes`.

7.20.3 Interpretive iteration

`times` (*n* —)
`many`
`#times` (— *addr*)

`times` re-interprets the text of the current command line (up to the `times` command itself) *n* times. More precisely (since *n* may be a word that refers to a variable quantity—which may be changed during the execution of `times`) the command line line will continue to execute until the number contained in `#times` is greater than or equal to the current value of *n*. `times` only works from the keyboard, and cannot be used within a file; it is slower than a compiled loop, since the command line is re-interpreted each time.

`many` works like `times`, but does not have a target number of times. The command line is continuously re-executed until a key is pressed.

`#times` is a variable containing the number of times that the current command line has been executed.

7.21 Compilation

7.21.1 Compiler

`:` *name*
`;` †C
`lose`

`:` (colon) is Forth's chief defining word. The phrase

`:` *name* `...`

creates a word definition for *name* in the CURRENT vocabulary and sets COMPILE mode. The search order is changed so that the first vocabulary in the search order is replaced by the CURRENT vocabulary; the latter is left unchanged. The text from the input stream is subsequently compiled. The newly created word definition for *name* cannot be found in the dictionary until the corresponding `;` or `;code` is successfully processed.

If during compilation of *name* a word is not found and cannot be converted to a number, the word `lose` is compiled. This prints aborts with the message "`Undefined word encountered`". This is in contrast to some other Forth systems which abort right away when the compiler sees an undefined word. Using the `lose` scheme, the compilation does not terminate upon encountering the undefined word, but instead proceeds and finds any other undefined words that may be in the file. This is a good scheme if compilation is very fast. See `compile-do-undefined`.

An error arises if, during compilation from mass storage, the input stream is exhausted before encountering `;` or `;code`.

`;` terminates compilation of a colon word, allows the name of this word to be found in the dictionary, set INTERPRET mode and compiles `unnest`.

See `immediate`.

7.21.2 Compilation control

`compile` C
`(compile)` (*acf* —)
`[compile]` †

`compile` is typically used in the form

`:` *new-word* `...` `compile` *name* `...` `;`

When *new-word* is compiled, the compilation address of word *name* is saved in the dictionary as a literal. When *new-word* is later executed, the address contained in the literal is compiled in the dictionary. *new-word* is typically immediate and *name* is typically not immediate.

`(compile)` takes a word address *acf* from the stack and compiles it in the dictionary.

`[compile]` forces the compilation of the immediate word *name*, which otherwise would be executed.

`immediate`
`immediate?` (*acf* — *flag*)

`immediate` marks the most recently created dictionary entry as a word to be executed rather than compiled when encountered during compilation.

`immediate?` returns TRUE if the word at *acf* is immediate.

`state` (— *addr*)
`?comp`
`?exec`
`]`
`[` †

`state` is a user variable containing the mode (INTERPRET or COMPILE) of the text interpreter: FALSE for INTERPRET, TRUE for COMPILE.

`?comp` aborts if executed while not in COMPILE mode, printing the message "`Compilation Only`". `?exec` aborts if executed while not in INTERPRET mode, printing the message `Execution Only`; only useful inside of immediate words.

The word `]` sets INTERPRET mode: subsequent text from the input stream is executed. The word `[` reverts the interpreter to COMPILE mode: Subsequent text from the input stream is compiled. For typical usage see `literal`.

`hide`
`reveal`
`recursive` †C

`hide` makes the most recently created dictionary entry invisible, so that it will not be found in the dictionary, regardless of whether the entry was previously visible or invisible. Used to implement the behavior of `:`, which ordinarily is not supposed to find a newly created word definition in the dictionary until the matching `;` or `;code` is successfully processed. The reason for making a word invisible while compiling it is so that one can redefine a word in terms of its old definition.

`reveal` makes the most recently created dictionary entry visible, so that it will be found in the dictionary, regarless of whether the entry was previously visible or invisible. Used to implement the behavior of `;` and `;code` which allow the newly created word to be found.

`recursive` makes the name of the colon definition being compiled visible, so that it will be found in the dictionary. This allows the word to be used recursively by compiling the name of the word inside of its own definition. For example, here is a recursive definition of factorial

```
                        : fact ( n -- n! ) recursive
              dup if
    dup 1- fact * else
             1+ then ;
```

7.21.3 Deferred definition

`defer` *name*
`crash`

The construct

`defer` *name*

creates a dictionary entry for *name* and makes it an "execution variable". *name* is similar to a variable which contains the compilation address of another word; that other word is executed when *name* is executed. The other word to be executed is stored into the data area of *name* by `is` or `(is`. If *name* is executed before it has been thus initialized, an error message will be printed. In this implementation, the address of the other word is stored in the user area instead of in the parameter field.

`crash` is the word which is installed in a deferred word when it is first defined. If the deferred word which contains `crash` is executed, `crash` will abort with the message:

name `<--deferred word not initialized`

Bug: There is a case in which `crash` will print the wrong *name*. If the deferred word was executed from the command line by an explicit `execute`, `crash` does not have enough information to find the name of the deferred word; instead `crash` prints the name of the last word interpreted from the input stream. Example:

```
  defer xyz   ' xyz execute
execute <--deferred word not initialized
```

7.21.4 Assembler

`code` *name*
`label` *name*
`c;`
`;code` †
`(;code)`
`end-code`

These words deal with *assembling* machine code rather than *compiling* Forth code.

`code` is a defining word used in the form

`code` *name* ... `end-code`

or in the variant

`code` *name* ... `c;`

It creates a dictionary entry for *name* to be defined by a following sequence of assembly language words. This newly created word definition for *name* cannot be found in the dictionary until a matching `end-code` or `c;` is successfully processed. It executes `assembler`, thus bringing in all of the assembler's specialized words. It does *not* switch to COMPILE mode!

`end-code` terminates a code definition and allows the *name* of the corresponding word to be found in the dictionary. It sets the CONTEXT vocabulary to be the same as the CURRENT vocabulary (this removes the `assembler` vocabulary from the search order, unless you have explicitly done something funny to the search order while assembling the code).

`c;` is a variant of `end-code`; before leaving assembly mode it assembles the NEXT routine, which will return control to the Forth inner interpreter. Both `c;` and `end-code` are defined in the `assembler` vocabulary.

`;code` is used in the form:

`:` *namex* ... `create` ... `;code` ... `end-code`

It is analogous to `does>`, except that the behavior of the word created by *namex* is specified in assembly language instead of high-level Forth.

`(;code)` is the run-time word compiled by `;code` and `does>`. When `(;code)` executes, it changes the code field of the most recently defined word so that that word will execute the code immediately following `(;code)`.

`label` is a defining word used in the form

`label` *name* ... `end-code`

It creates a dictionary entry for *name* under which it assembles the following code until `end-code`. When *name* is later executed, it will return the starting address of this code fragment.

7.22 Decompilation and debugging aids

7.22.1 Decompiler

`see` *name*
`(see)` (*acf* —)
`cdis` (*acf* —)
`dis` *name*
`+dis` *name*
`sifting` *name*
`sift` (*pstr* —)
`.id` (*anf* —)
`.name` (*acf*)
`.calls` (*acf* —)
`.current-word` (*addr* —)

`see` decompiles the word *name*—or disassembles it if it is a code definition. `(see)` does the same to the word at *acf*.

`cdis` disassembles the code word at *acf*. `dis` disassembles the machine instruction starting at *addr*, stopping when a `jump`, `branch`, or `return` instruction is encountered. `+dis` continues disassembling from the last stopping point.

`sifting` displays the names of all dictionary entries whose names contain the string *name*. With `sift`, *name* is taken from memory (typically from the stack) as a packed string at *pstr*, instead of from the input stream.

`name` displays the name of a word given its compilation address *acf*; `.id` Prints the name of the word at the given name-field address *anf*.

`.calls` list all the words which call the word whose compilation address is *acf*.

id—current-word— displays the name of the colon definition within the address range of whose body the given address *addr* lies.

`definer` (*acf* — *definer-acf*)

`definer` returns the compilation address of the defining word which defined the word at *acf*.

7.22.2 Patching

`noop`

Don't do anything. This can be used to waste time, or as a place-holder for something that will be patched in later (cf. **patch**).

`patch` *name' name target*
`npatch` (n' n —) *target*
`(patch` (n' n *acf* —)

`patch` is used in the form

  `patch` *name' name target*

It replaces the first occurence of the word *name* with the word *name'* in the definition of the word *target* as compiled in the dictionary.

`npatch` replaces the first occurrence of the number n with n' in the word *target*.

Warning. If the number n contained in the definition of *target* is expressed by a constant rather than a literal, `npatch` won't work on it since the value of this number does not appear within the body of the definition itself. In this case, use `patch` with an appropriate word for *name'*.

`(patch` is like `npatch`, but the target word is specified by its *acf* rather than by name.

7.22.3 Debugger

8 STEP thesaurus

8.1 Space level

8.2 Event counting

9 Thesaurus index

Index