APPENDIX A: PATTERNFORTH GLOSSARY

This glossary describes all of the Forth words which comprise the patternForth system. These are the extension words which are added onto a "normal" Forth system to provide the functionality of patternForth.

The extension words are divided into two categories.

"Public" words are the "official" glossary of patternForth; those words which are available to the application programmer. These words represent the "appearance" of patternForth, and are defined so as to remain constant across different implementations.

Private words are those words used internally by patternForth, which are not available to the programmer. They are highly implementation-dependent and should remain "hidden." If the Forth system supports a "module" structure, these should be contained within the "private" part of each module.

Stack nomenclature follows common Forth usage. Each word may accept arguments on the stack, and leave arguments on the stack. These are given as

```
( inputs - outputs )
```

in the definition of each word. When multiple inputs or outputs exist, the "topmost" stack item is on the right. Some common notation for stack arguments:

`a` a single cell address

`n` a single cell integer value

`f` a flag; zero = false, nonzero = true

Some Forth words also accept in-line text arguments; that is, text arguments which follow the word in the source code. The nomenclature used for these is

```
( "<text>" - ...
```

where the " marks may be replaced by whatever delimiters are actually required. (This notation follows that used in the ANSI Forth standard.)

Forth IMMEDIATE words which are compiler directives are indicated by the notation C,I .

MEMORY ALLOCATION WORDS

"Public" words

```
\ALLOC ( - )
```

Initializes the memory allocator, and empties the string space.

```
ALLOC ( n - a 0 )
or ( n - 1 )
```

Allocates n bytes of memory from the string space. If successful, returns the address of the data area in the allocated region, and a false flag. If failed, returns a true flag. ON THE 8086: the address returned is in the string segment.

```
RELEASE ( a - )
```

Given the address of the data area in an allocated region of memory (as returned by ALLOC), returns that region to the available space.

"Private" words

```
$SEG ( - n )
```

A constant, returning the starting 8086 "paragraph" -- i.e., segment pointer -- used for string space.

`ROVER ( - a )`

A variable, holding the starting position in string space for allocation searches.

STRING CREATION AND ACCESS WORDS

"Public" words

`\HASH ( - )`

Empties the hash space, clears all hash lists, and initializes the default hash table.

`>$ ( a n - s )`

Given the address and length of a string, finds or creates that string in the string space, and returns its string descriptor. Aborts if string cannot be created. ON THE 8086: the string is presumed to be in the Forth segment.

`" ( "<text>" - s )`

Usage: `" <text>" ...`

Finds or creates the string <text> in the string space, and returns its string descriptor. This word is "state-smart" and may be used interpretively from the command line, or within a Forth colon definition. In the latter case, the word is not found/ created until the Forth word executes.

`\$ ( a n - )`

Given the address and length of a string, deletes the string from string space and the hash table. ON THE 8086: the string is presumed to be in the Forth segment. ("wipe-string")

`$TEXT ( s - a seg n )`

Given a string descriptor, returns an address and length where its string text may be found. This address is not guaranteed to remain valid after other string access functions; it should be used immediately and should not be saved. ON THE 8086: the address is returned as address, segment. N.B.: future versions may copy this string text to a "safe" buffer in Forth space.

`$! ( n s - )`

Given a string descriptor and an arbitrary cell of data 'n', stores the data in the "associated data" field of the string; i.e., stores the data at that string- identified location. The data may be any single-cell data, including an integer, an address, or a string descriptor. ("string-store")

`$@ ( s - n )`

Given a string descriptor, fetches the data 'n' associated with ("stored at") that string. ("string-fetch")

`TABLE ( s - )`

Given a string descriptor, creates and initializes a new hash table, and associates that hash table with the string (makes the given string the identifier for the new table).

`{ ( s - )`

Sets the string context to the hash table associated (by TABLE) with the string s. This context will remain in effect until changed by } or another { . Note: by definition, all newly-created strings reference the "default" hash table, so if this operator is used on a string which has not had a TABLE defined, the default table is selected.

`} ( - )`

Restores the string context to the "default" hash table.

"Private" words

`?HASH ( f )`

If entered with a true flag, displays the message "hash space full" and aborts.

`HALLOC ( - a f )`

Allocates one hash list element from the free pool. Returns the address of the element in hash space, and a flag: false if successful, true if failed (for ?HASH).

`HASH ( a n - a n u c )`

Given the address and length of a string in Forth space, calculates the 16-bit ascension value u and the 8-bit hash code c. (Only 6 bits of the hash code are presently used.) The address and length are left on the stack.

`HFIND ( a n u c - a n u a' f )`

Given the address a, length n, ascension value u, and hash code c of a string (as returned by HASH), searches the current hash table for that string. Returns a, n, and u unchanged, plus the address a' of the link to the hash list element for the string, and a flag f, false if found, true if not found. If the string was not found, the address a' is the link which points to the string's position in the sorted list; i.e., the link which has to be changed to insert the new string into the hash list.

`HINSERT ( u a' A n - A f )`

Given the ascension value u, address a' of the preceding link (as returned by HFIND), an address A of allocated memory in string space, and the length n, inserts a new element into the hash list and initializes the element. This allocates an available element from the free pool, links it into the given hash list, and initializes its pointers and other data values. The associated data is set to zero, as is the associated table (default table).

`HDELETE ( a' - )`

Given the address of the link to a hash list element, removes that element from the singly-linked list. The given link is relinked "around" the deleted element. The deleted element is added to the free list.

`\TABLE ( a - a )`

Given the address a of the "root" element, allocates 8 more elements to make up a hash table, and initializes the table to empty. The address a remains on the stack.

`HFREE ( - a )`

A variable which points to the head of the linked list of "available" hash elements.

`$CONTEXT ( - a )`

A variable which contains the address (in hash space) of the hash table which is currently being used for string searches.

`HSIZE ( - n )`

A constant, the size in bytes of a hash list element.

`H0 ( - n )`

A constant, the address of the first hash list element in the initialized list. Used by \HASH.

`#H ( - n )`

A constant, the total number of hash list elements available in the hash space.

PATTERN MATCHING WORDS

"Public" words

`<< ( - 20 20 ) C,I`

Begins a series of alternatives; equivalent to the Snobol4 '(' . The series of alternatives is terminated with >> . Sets of << >> may be nested. In the current patternForth, it is also necessary to bracket single alternatives with << >> .

`| ( - a 21 ) C,I`

Separates alternatives in a series; equivalent to the Snobol4 '|'. Typical usage:
`<< alt1 | alt2 | alt3 >>`

`>> ( 20 20 ... a 21 - ) C,I`

Terminates a series of alternatives, or a single "alternative." Equivalent to the Snobol4 ')'.

`PAT: ( "<name>" - ) C,I`

Usage: `PAT: <name> ... ;PAT`

Begins a pattern definition called <name>. <name> is a high-level Forth word which can be used in other Forth definitions. It will follow the rules for a pattern element (e.g. returning a success flag), so it can be used to construct other patterns.

`PAT; ( - ) C,I`

Ends a pattern definition.

`EVALUATE ( s a - f )`

Execute the pattern definition at address a, using the string s as the subject string. Returns a success flag f, true if a match was achieved, false if not. (This is the 'top level' of pattern evaluation, beyond which there is no backtracking.) Typical usage: " string" ' <name> EVALUATE

"Private" words

`ENTER< ( - a oip ) (R: ip - ip a )`

Sets up the stack context for pattern matching. The current scan position a is pushed onto the Return stack, to be used by (|). A "fallout" backtracking record is pushed onto the History stack. ENTER< is compiled after the "ordinary" colon definition code by PAT: . The sequence `PAT: name ...` is equivalent to `: name ENTER< ...`

```
>EXIT ( a oip - 1 ) (R: ip a - ip )
or ( xx - xx a iip 1 ) (R: ip a - ip )
```

Cleans up the stack context at the end of a subpattern. The scan position is dropped from the Return stack. If no backtracking records were pushed during the subpattern -- as indicated by the 'oip' pushed by ENTER< being on the top of the History stack --then the "fallout" record is discarded; otherwise, a "fallin" record is pushed onto the History stack. A "true" flag is pushed onto the stack to indicate subpattern success. >EXIT is compiled before the "ordinary" colon exit by ;PAT. The sequence `... ;PAT` is equivalent to `... >EXIT ;`

`(<<) ( - ) (R: ip a - ip a )`

The execution-time code for << . Saves the current scan cursor in the Return stack context, in the topmost Return stack item. This establishes the position where alternatives will be tried.

```
(|) ( =t - a ip ) (R: ip a - ip a )
or ( =f - ) (R: ip a - ip a )
```

The execution-time code for | . Tests the result of a pattern element or subpattern, and either proceeds to the next alternative (if failure), or skips the remaining alternatives (if success). If the former, the scan position is restored non- destructively from the top of the Return stack. If following alternative(s) are skipped, its instruction address ip and the "restore" position of the scan pointer, a (obtained from the Return stack), are pushed on the History stack.

```
(>>) ( =t - )
or ( a ip =f - )
```

The execution-time code for >> . Tests the result of a pattern element or subpattern, and invokes backtracking if required. If the flag is true, execution proceeds normally. If false, a "backtracking record" is popped off of the stack, restoring the subject scan pointer 'a' and the Forth interpretation pointer 'ip' to a previous state.

`MARK ( - ) (R: ra - ra sa )`

Marks the position of the History (data) stack, by saving its current stack pointer on the Return stack. This will be used to discard unneeded History information from the stack at the end of a pattern match.

`RESTORE ( xx ... f - f ) (R: ra sa - ra )`

Restores the position of the History stack, from the pointer which was saved by MARK. The topmost History stack item is presumed to be a flag; it is preserved on the stack (removed, and replaced after the stack position is restored).

`(FALLIN) ( a ip ra - ) (R: - ra a')`

A high-level code fragment, which, when executed by a backtrack, restores one level of Return context. This Return context is composed of the subpattern return address ra (from the History stack), and the scan pointer a' (from CURSOR, which was just restored by the backtrack). After this is built, the backtracking record a ip on the History stack is invoked.

`(FALLOUT) ( - 0 ) (R: ip a - )`

A high-level code fragment, which, when executed by a backtrack, returns to the caller of this subpattern with a failure flag. The Return stack context is removed and a failure flag is pushed.

`FAILURE ( - 0 ) (R: ip a - )`

A high-level code fragment, executed when all backtracking options have been exhausted. The backtrack record which executes FAILURE is the "deepest" record on the Return stack. When executed, it removes the return stack context, pushes a failure flag, and returns to the caller. Note: at present, this is identical to the action for (FALLOUT).

`SUBJECT ( s - )`

Given a string s, sets the subject string pointers used by the pattern match logic. Note: the current implementation causes the pattern matcher to directly use the string text stored in string space. This will become invalid if compaction is used; in such a case the string should be copied to a "safe" buffer in Forth space, and the pointers set there.

`CURSOR ( - a )`

A variable holding the current scan position in the subject string. This is an offset from CURADR; the starting CURSOR position is zero (the left end of the string).

`CURADR ( - a )`

A variable holding the starting address of the subject string in memory. CURSOR and EOS are offsets from this address.

`CURSEG ( - a )`

A variable holding the 8086 segment where the subject string is stored. The full 8086 address of the subject string is CURSEG:CURADR .

`EOS ( - a )`

A variable holding the offset of the end of the string; i.e., the CURSOR value for the "right end" of the text. This is equal to the length of the string text.

PATTERN OPERATORS

These words may be used as pattern elements in a pattern expression.

"Public" words

`MATCH ( s - f )`

Compares the text at the current scan position with with the string s . Returns true if matched, false if not. Typical usage: " text" MATCH

`ANY ( s - f )`

Matches a single character at the current scan position with the string s . Returns true if the character is contained anywhere within s, false if it is not. Typical usage: " characters" ANY

`LEN ( n - f )`

Returns true if there are n characters remaining in the subject text (to the right of the current scan position). Typical usage: n LEFT

`LEFT ( - n )`

Returns the number of characters remaining in the subject text. Used within LEN; also available to the programmer. (Since any non-zero value is considered "true", this can also be used to test "is there any text left?")

`POS ( n - f )`

Returns true if the scan pointer is at position n. (0 is the left end of the pattern; EOS @ will return the right end of the pattern.)

`ARB ( - f )`

Matches an arbitrary number of arbitrary characters. Always returns true; ARB can match zero characters. ARB always attempts to match the largest possible string, it "backs" down to zero as it is backtracked.

"Private" words

`(MATCH) ( adr seg n - f )`

Compares the text at the current scan position with the string given by adr seg n . Returns true if matched, false if not. Used within MATCH.

`(ANY)`

Matches a single character at the current scan position with the string given by adr seg n . Returns true if the character is contained anywhere within s, false if it is not. Used within ANY.

STRING OPERATORS

`$TYPE ( s - )`

Displays the string text on the terminal, in the manner of Forth's TYPE. Handling of non-ASCII characters is machine-dependent.

`SIZE ( s - n )`

Returns the length of a string n. (This length does not include the null terminator that is automatically appended by the string creation routine.)

`CONCAT ( s1 s2 - s3 )`

Concatenates strings s1 and s2, returning the new string s3. s1 and s2 are unaffected.

`SUBST ( s1 ofs n - s2 )`

Returns the substring of s1, starting at ofs for n characters, as a new string s2. s1 is unaffected. The left end of the string is offset zero. If there are insufficient characters in the string, the result is truncated; if ofs is beyond the right end of the string, a null string is returned.

`FILLED ( n c - s )`

Creates a string of length n filled with the character (byte value) c.

`$>N ( s - n )`

Converts a text string s to a binary number n, using the current conversion BASE. The Forth rules for numeric conversion are followed: conversion proceeds up to the first non-convertible character. Minus signs and decimal points are considered non-convertible. Thus " 123.56XY" $>N will return 123, while " -12" $>N will return zero. ("string-to-number")

`N>$ ( n - s )`

Converts a binary number n to a string s, using the current conversion BASE. This is a signed conversion with no leading or trailing spaces.