

# The Strictly False Programming Language

M. Randall Holmes

September 12, 2005

## 1 Introduction

### 1.1 Version

This is dated September 12, 2005, correcting some errors (notably, `m` and `M` were swapped) and adding some features (notably `Z`) introduced in that version.

### 1.2 Introduction

This document is the reference for the Strictly False programming language, which is an extension of the elegant False language proposed by Wouter van Oortmerssen (see <http://wouter.fov120.com/false/> for an account of this language), which is itself a variant of Forth.

Van Oortmerssen says “I designed this language with two particular objectives: confusing everyone with an obfuscated syntax, and designing an as powerful language as possible with a tiny implementation: in this case a compiler executable of just 1024 bytes (!), written in pure 68000 assembler.”

In our opinion the syntax, while certainly compact, is hardly obfuscated (though certainly confusing to the uninitiated): it is simply reverse Polish notation, as found in Forth or on an HP calculator, and compactness is enforced by the fact that every command is a single character (with the caveat that there are really some two-character commands, at least in our opinion). In False there are also some brackets. The language is extremely easy to parse (for a computer).

This said, it must be admitted that the language is rather hard to read for the uninitiated. We think that the extreme compactness has at least one

sensible application: we plan to implement this language on our handheld, where keystrokes are annoying.

Forth (and False, and Strictly False) are based on the use of a stack. The + command of any of these languages, for example, has the specification “take two numbers off the stack, add them, and put the result on the stack”. A data item like 2 also has execution behavior: “put 2 on the stack”. The program 2 3 + then reads “put 2 on the stack (stack is now [2, ...]); put 3 on the stack (stack is now [3, ...]); take the top two items off the stack and add them, then put the result back on the stack (stack is now [5, ...]).

The program itself can be thought of as a stack: execution consists of popping commands one by one off the program stack and making the appropriate changes to the data stack. False takes this viewpoint and introduces a new twist. Consider the state of the machine with [3,+] in the program stack: this is a function with one argument, which may briefly be summarized as “add 3”. The refinement which produces False (on top of a set of Forth-style stack commands with one-character opcodes) is the ability to introduce objects like [3+] (program stack states) as data objects which can be put on the stack! A new opcode ! allows these functions to be executed: for example, the program 2[3+]! works as follows: “put 2 on the stack (stack is now [2, ...]); put [3+] on the stack (stack is now [2, [3, +], ...]); apply (3+ is added to the beginning of the program stack; stack is now [2, ...]); put 3 on the stack (stack is now [3, 2, ...]); add (stack is now [5, ...])”.

Of course what is actually popped onto the stack when [3+] is put there is the address of a list data structure.

Our reaction to the False language was of course first to marvel at its obfuscated nature just like anyone else. But the compact representation of higher-order objects (functions of any number of arguments) allowed by False is of considerable philosophical interest to us. One point about it is that it allows a representation of abstractions which does not necessarily involve variables, in the same way that combinatory logic does this.

At this point, we originally wrote an extensive discussion of why one cannot implement combinatory logic directly with False functions. On examining this text, I decided that if I included it at this point no one would finish reading the document, so I deferred it to the last section. Briefly, one cannot abstract into False functions because there is no way to operate on the insides of them; but False functions are lists of commands (some of which are confused with data), so if one adds basic operations on lists one gets the ability to implement synthetic combinatory logic – and to do lots

more interesting programming :-)

Strictly False adds list processing to the capabilities of False and we see that we gain the full (theoretical) power of synthetic combinatory logic, which is a complete functional programming language (False is certainly a complete programming language in some senses, but it is probably not a complete functional programming language).

Another way in which Strictly False differs from False is that it is (dynamically) typed. Each object is either an integer, a character, a truth value, or a list/function, and operators expect to see arguments of particular types. In the original False, various cool hacks could be obtained by type-casting: these are forbidden here.

Strictly False has file I/O: it can read and write characters from files and also can execute a file as a function and save a function to a file. It has the same function definition capability as False (using the `:` and `;` opcodes) but these are explicitly restricted to functions; the capability is added of converting such defined functions to new opcodes. It also has the ability to address a large supply of numerically addressed data cells (the current interpreter has a very generous memory model, probably because I haven't yet considered implementing it with real memory allocation).

Strictly False supports continuation passing (!!).

The current implementation is an interpreter running under Standard ML; we hope to implement it in C (for efficiency) and then in some language implement it for our handheld device.

## 2 The Implementation in ML

We assume familiarity with ML. If you have compiled, loaded, and opened `purefalse.sml`, then to execute a Strictly False program `<program>` you should type

```
execute "<program>"
at the ML prompt.
It is useful to be aware that
execute "T"
toggles the trace feature and that
execute "nd"
clears the stack, which persists.
executelines();
```

allows you to write many lines of literal text to be fed to the interpreter, stopping when you enter an empty line. This allows you to avoid ML escape sequences for `\`, `"`, and perhaps other characters, and it also allows one to enter carriage returns inside messages, a peculiar feature of `False` which is inherited by `Strictly False`.

All these commands need to be typed at the ML prompt: if the trace feature is on, you need to hit return many times before you get back to the ML prompt! The system may also stop for input (and `SF` does not have an input prompt: you are encouraged to use a message for this purpose!)

## 3 Command Reference

### 3.1 Parsing the Language

Most commands in `Strictly False` are single characters. There are two special two-character forms:

**Characters:** `'x` represents the literal character following the single quote. The effect of executing this is to push the character onto the stack.

**Atomic Programs:** `'x` stands for the atomic program `[x]` (where `x` is a single character). It's really only present so that the bracket notation is eliminable in principle.

There are four multi-character notations.

**Comments** `{...}` is a comment. The only restriction on the text represented by the ellipsis is that curly brackets are balanced: comments can be nested. Comments are completely ignored by the interpreter.

Curly brackets have another function: if you see an expression in curly brackets in the display of the data stack it indicates that this is a literal piece of program text embedded in the term. Be aware that program text dropped onto the data stack is executed immediately. Functions dropped onto the data stack have most program text eliminated and replaced with the corresponding data (there are some exceptions). A single character opcode is always a piece of program text, and this is not indicated in the display. Messages in displayed functions are always enclosed in curly braces: do not do this in your programs! The P

command (see below) loads the current continuation as an unanalyzed chunk of program text.

**Messages** "... " is a message. The effect of executing this is to send the text represented by the ellipsis to standard output. There is no effect on the stack. The only restriction on the text is that it may not contain double quotes (it may contain carriage returns, though this is difficult to exploit in this interpreter). Messages embedded in functions on the data stack are always program text, and so are enclosed in braces as well.

**Functions/Lists** [...] is a function or list. The text must be a valid Strictly False program (the interpreter does not check for the validity of opcodes, just for the syntax of multi-character forms). The effect of executing the bracketed form is simply to put it on the stack (like any other data). If the ! opcode (application) is run, the function on top of the stack is executed as a Strictly False program. The language also supports list operations; one must be careful, though, as one may pop out a “data item” with execution behavior!

**Numerals:** In some theoretical sense numerals can be interpreted as made of single character opcodes and the execution of the interpreter reflects this (it actually proceeds single character by single character, with one-character lookahead where digits are involved), but the internal parser views them as single chunks. A string of one or more digits followed by an underscore \_ is a numeral representing a negative integer. The underscore is also a general unary additive inverse operator. The only place where whitespace has a function is where it is used to mark breaks between numerals.

## 3.2 Opcodes (in functional categories)

### 3.2.1 Pure Stack Operations

The “pick” operation of False, which was represented by a nonstandard character, is not provided but can be implemented.

**drop:** % When this command is executed, the top element of the data stack is dropped. (False).

**swap:** \ When this command is executed, the top two elements of the stack are interchanged. (False).

**duplicate:** \$ When this command is executed, the top element of the stack is duplicated (an extra copy is put on). (False).

**rotate:** @ When this command is executed, the third element of the stack is brought to the top: if the original order were 123 (from the top down) the new order would be 312. (False).

### 3.2.2 Simple Data

There are three types of simple data in Strictly False, booleans, integers, and characters. There is one other type of data: programs = lists. The data typing is a difference from False.

**true:** t When this command is executed, a copy of the value “true” is placed on the stack. (new in Strictly False).

**false:** f When this command is executed, a copy of the value “false” is placed on the stack. (new in Strictly False).

**digits:** 0–9 These execute in two different ways, depending on whether they are followed by another digit or not. In any case, the result is standard decimal notation for integers. (False).

**numerals:** A block of digits is interpreted as an integer as usual (in base 10). A block of digits followed (not preceded) by `_` is interpreted as a negative integer. (False).

**Characters:** As noted above, `'x` is read as the character `x` (for each possible choice of character.) The effect of executing this is to put a copy of the character on the stack. (False).

**Atomic Programs:** For any character `x`, the notation `'x` is an alternative notation for the program `[x]`. (new in Strictly False; `'` means something quite different in False).

### 3.2.3 Simple Input and Output

**Messages:** " . . . " `q` `r` As noted above, any string of characters enclosed in double quotes (excluding double quotes but allowing carriage returns) has the effect when executed of sending the enclosed character string to standard output. `q` sends a double quote to standard output. `r` sends a carriage return to standard output. (quoted strings as in False; `q` and `r` are new in Strictly False).

Messages embedded in functions on the data stack are program text, and so are displayed in braces.

**Character input:** `^` When executed, this command accepts character input. Input is buffered, so you can enter many characters and the program will take them as it needs them. No prompt is provided: you should use a message for this purpose. (False).

**Character output:** `comma(,)` When executed, a character is removed from the top of the stack and printed. Type errors stop the program. (False).

**Integer output:** `period(.)` When executed, an integer is removed from the top of the stack and printed. Type errors stop the program. (False).

**Flush buffers:** The close quote `)` flushes the interpreter's input buffer and ML's output buffer (in this implementation). (This is different from False in using a standard character.)

### 3.2.4 Operations on Simple Data

**Arithmetic Operations:** `+` `-` `*` `/` Each of these takes two integers off the stack and puts the result of applying the operation (with the second item on the stack being the first argument) back on the stack. All calculations are mod a large number (200000000 in this implementation) with the larger half construed as negative numbers (subtract 200000000 from numbers greater than 100000000); there are no overflow crashes, though results may be unexpected. Division by zero stops the program. Type errors stop the program. (False: details of our arithmetic that avoid overflows are different, I assume).

**Comparison:** `=` `<` `>` Each of these takes two items from the stack (both integers or both characters) and compares them in the indicated way,

with the second item on the stack being the first argument. A truth value is placed on the stack. Type errors stop the program. = is also supported for lists/programs, but it behaves differently; see the entry below. (False: I don't know that equality of lists is supported).

**Type Casting:** *c C* The *c* command takes a character off the stack and replaces it with an integer or vice versa. An integer will be processed mod 256 (no crashes). Type errors crash the program. (new in Strictly False: not needed in False).

The *C* command similarly converts characters to atomic programs ('*x*' to '*x*') and vice versa. (new in Strictly False).

**Propositional Logic:** *~ & |* These commands take one or two truth values off the stack and replace them with the result of the appropriate operation: negation, and, (inclusive) or respectively. (False)

### 3.2.5 Operations on Lists/Programs

All commands except ! (function application) are new in Strictly False. I do not know whether False supports equality of lists.

**Brackets:** *[...]* Entering a bracketed Strictly False program simply places a copy of this program/list on the stack. *[]* has the same effect as *n*.

**Empty list/program:** *n x* The *n* opcode puts an empty list *[]* on the stack. The *x* opcode puts *t* on the stack if the list on top of the stack is empty and *f* if it does not; it does **not** remove the tested list! Type errors stop the program.

**Cons:** *p* When executed, removes the first item (which must be of any type) and the second item (which must be a list) from the stack and replaces them with the list with the first item as head and the second as tail.

**Composition:** *o* When executed, removes two lists from the stack and appends the second item to the first item (this is list concatenation).

**Head:** *i* When executed, removes a list from the top of the stack, pushes the head of that list onto the stack (or executes the head of the list if it

is an opcode or program text) then pushes the tail of the list back onto the stack. Repeated execution of *i* traces the execution of a function on top of the stack step by step.

**Inert Head:** *j* When executed, removes a list from the top of the stack, puts the one-item list containing its head on the stack then puts its tail on the stack. This averts possible execution behavior!

**Function application:** *!* When executed, causes the list on top of the stack to be executed as a program.

**List equality:** *=* This is supported with some misgivings. It does *not* remove the two lists compared from the stack; it just adds an appropriate truth value to the top of the stack.

### 3.2.6 Control Structures

Both of these are as in False.

The *!* operator above and the *P* and *D* operators below are also control operators.

**Conditional:** *?* Pops two items off the stack, a function (top) and a truth value. Executes the function just in case the truth value is *t*.

**Loop:** *#* Pops two functions off the stack: the top item (second argument) is the body of the loop and the second item (first argument) is the test. Executes the test: there must be a truth value on top of the stack at the end of each execution of the test (or the program stops with a type error). As long as the truth value is true, the body continues to be applied.

### 3.2.7 Function Definitions

**Function definition:** *:* pops a character (top) and a function (second item, first argument) off the stack and binds that function to that argument (only functions may be defined). Use *;* to exploit this binding. Note that typically this will look like *'x:* not *x:* as in False. (False, except only functions may be bound to a character and the argument really is a character).

**Use of function definition:** `;`  pops a character off the stack and looks up the function bound to that character: this function is immediately executed (not as in `False`, where any kind of data can be bound using `:`). Note that typically this will look like `'x;.` (`False`, except that the argument is a character and it incorporates the effect of `!`).

**Viewing a defined function:** `E`  pops a character off the stack and puts the function bound to it on top of the stack (as `;`  except inert). This is actually the `False` version of `;` , I believe.

**Defining Opcodes:** `B`  (not really intended to be used inside a program): takes a character off the stack, and defines the character (if there is no conflict) as a new opcode with the effect of the defined function. `'x;`  will be replaced with `x`  as appropriate inside the text of the function to enable recursion to work. The evil function `x`  which makes essential use of `'x:`  inside its text will not be satisfactorily simulated. There is no user command to remove such a binding! (New in `Strictly False`).

A hacking opportunity: at the moment conflicts with built-in opcodes are not recognized, so one could define a function to extend the behavior of an existing function (into cases where the original function stops with a type error). I may fix it so this can't be done.

### 3.2.8 Machine Access

These are dangerous commands which can produce dazzling special effects. They are all new in `Strictly False`.

**Data Stack Access:** `s S d`  The relatively harmless `s`  tests whether the data stack is empty. The more exciting `S`  loads the data stack onto the data stack as the top element. The aggressive `d`  replaces the data stack altogether with the list which is the top item of the stack (so `nd`  erases the data stack completely). Creation of a data stack with executable program text or opcodes appearing after data causes special effects: when data is popped off, executable text revealed will be executed. Although I haven't tried it yet, it ought to be possible to create a data stack which is effectively an infinite lazy list.

**Program Stack Access:** `P D`  The `P`  command loads the remaining portion of the current program yet to be executed (the continuation) onto the

top of the data stack. This is loaded as a chunk of unanalyzed program text, and so it is displayed in braces. The `D` command replaces the program being executed with the function on top of the data stack. So `nD` is **stop**, and the combined use of `P` and `D` should allow continuation passing!

### 3.2.9 The Memory Array

These commands are new in Strictly False.

There is a memory array indexed by all available integers. Cells must be initialized before being used, and in fact local declarations are supported (after a local declaration is undone, earlier values become available). Each memory cell is actually a stack, whose top element is the current value in that cell.

**Initialize:** `I` Takes two items off the data stack, the second of which is an integer index of a cell. Pop the top item from the data stack onto the indexed memory cell stack: this is declare and initialize a variable.

**Deallocate:** `e` Takes one item off the data stack, the integer index of a memory cell. Pop off the top item in the stack associated with that memory cell. The effect is to free a variable (and perhaps to make a variable with the same index declared earlier visible again). If the memory cell stack is empty, the interpreter stops with an error.

**Assignment:** `A` Takes two items off the data stack, the first being a value to be assigned and the second the integer index of a memory cell. Pop off the first item in the memory cell's stack and push the "value to be assigned" taken from the data stack onto the memory cell stack. The effect is to assign the value to the indexed variable.

**Reference:** `a` Takes one item off the data stack, an integer indexing a memory cell, and puts the top item in the stack in the indexed memory cell on top of the data stack. The effect is to get the current value of the indexed variable onto the stack.

### 3.2.10 File Input and Output

These commands are new in Strictly False. Note that in all file commands the `fileID` (a character bound to the file as an identifier) is always the argument

next to the opcode.

**Open a file:** `O` removes all characters from the stack and reads the first one as a file designator and the rest as a filename, with the second character on the data stack being the last character in the file name. This file is opened for input and output and bound to the top character as an identifier (fileID).

**Open a file for read only:** `Z` is as `O`, except that the file is opened to read only. The interpreter will prevent writing operations to files opened with `Z` (by fatal error).

**Close a file:** `F` Takes a single character off the stack and closes the file for which it is fileID for both input and output. If no file is found, fatal error.

**Read a character from a file:** `R` Takes a character (fileID) off the stack and attempts to read a character from that file. It puts a truth value (true if it read the character) on the stack then puts the character read on the stack if there is one. If the fileID does not refer, the interpreter stops with an error.

**Write a character to a file:** `W` Takes two characters off the stack: writes the second item (first argument) to the file indicated by the first item (second argument) (error if fileID does not refer).

**Write a function to a file:** `m` takes a character off the stack (fileID) then a function, and writes the function to a file (the contents of the file will not include the brackets around the function).

**Execute a function from a file:** `M` takes a character off the stack (fileID) and executes the function written there.

### 3.2.11 Wish List

I'd like to be able to save file information in a memory cell; this would need to be another data type (file handles). The command to retrieve a file handle would supply the fileID to which to bind it.

I'd like to be able to edit defined functions: I think that all I need to do is be able to bring it to the top of the stack without executing it (now implemented by `E`), and then a Strictly False line editor can be written :-)

In an implementation where memory is really being manipulated, the `I` command should probably have a parameter to indicate the length of a block of memory cells to be allocated; when I write such an implementation, I will probably change the spec of the language (and change this interpreter) to do this: for the moment, cells have to be declared one by one anyway, because the implementation of the array is stupid :-)

### 3.2.12 Debugger Commands

These are provided with a mind to making the interpreter its own development environment.

**Trace feature:** `T` toggles the trace feature on and off.

**Display data stack:** `U` sends an image of the data stack to standard output.

**Display continuation:** `V` sends the current continuation (rest of the program to be executed) to the standard output.

## 4 Implementation of Combinatory Logic

Let a function  $f$  of one variable be considered as a False program which pops a value  $x$  off the stack and replaces it with  $f(x)$ .

In combinatory logic, we construct for any expression  $T$  in variables and application, a function  $(\lambda x.T)$  such that  $(\lambda x.T)(x) = T$  for all  $x$ . If  $T = x$ , we define  $(\lambda x.T)$  as `I`, where  $I(x) = x$  for all  $x$ . If  $T = a \neq x$ , we define  $(\lambda x.T)$  as  $K[a]$ , where  $K[a](x) = a$  for any  $x$ . If  $T = U(V)$ , we define  $(\lambda x.T)$  as  $S[(\lambda x.U), (\lambda x.V)](x)$  (where we may suppose that  $(\lambda x.U)$  and  $(\lambda x.V)$  have already been defined, by structural induction on the language of terms built up from variables by application).

$I$  is implemented by a False command which does nothing (`[$%, “duplicate then drop”, will do the trick)`, so `x[$%]!` will execute as desired, leaving just  $x$  on the stack.

$K[a](x) = a$  tells us that a program  $P$  implementing  $K[a]$  should have the effect of removing the top item  $x$  from the stack and replacing it with  $a$ : the program `[%a]` has this effect: `x[%a]!` puts  $a$  on the stack.

$S[a, b](x) = a(x)(b(x))$  is the behavior required for  $S[a, b]$  to work as advertised. `[$b!\a!!]` is the program with the desired effect, where `$` is

the program which duplicates the top of the stack, `\` is the program which swaps the top two elements of the stack, `(%` (seen above in `K`) is the program which drops the top element of the stack and `!` is the function application program).

This means that any expression written in the language of function application and concrete functions can be abstracted from in `False`. But notice that we cannot abstract into our representations of  $K[a]$  and  $S[a, b]$  themselves, since they involve the further construction of `False` functions (lists of commands).

We observed that  $K[a]$  is represented by `[%a]` and considered that it is obvious how to construct this: the function is a list: add `a` to it at the front, then add `%` to it. In other words, since we have “functions” in `False` which are clearly list data structures, add the operations appropriate to lists to the language.

There is a technical problem which immediately presents itself: we can’t add `%` onto the list at all, because it has execution behavior: we are presumably pushing one data item on the stack into another, and `x` (presumably data) can be there without difficulty, but we can’t put `%` on the data stack except in the packaged form `[%]`. So two different flavors of push are provided: the program `p` adds the data item which is second on the stack to the front of stack (list or function) on top of the stack, while the program `o` composes the two stacks on top of the stack: to prepend `%` to the stack, push the data item `‘%` (an abbreviation for `[%]`; the `‘` notation makes it so that the bracket notation is in principle eliminable in favor of one-character opcodes). To start constructing any list, we need the null list (opcode `n`). So `K` is implementable in our extended dialect as `[n\p‘%o]`: pop the null list onto the stack, then push our single argument `x` into it (`\` puts 3 and the null list in the correct order for `p`) creating `[x]` at the top of the stack, then push `[%]` onto the stack (this will safely sit there rather than do anything) and compose the two (obtaining `[%x]`).

Let’s look at `S`. `[$b!\a!]` is our implementation of  $S[a, b]$ , and we want to be able to read it as  $S(a)(b)$ .

`[n[!!]oap[!\]o\p[$]o]` is  $S(a)$  (this just builds the string of commands representing  $S[a, b]$  if `b` is on top of the stack; it is easier to follow if you realize that it has to be built back to front. And, similarly, `S` is `[n[p[!\]o\p[$]o]o\p[n[!!]o]o]`. Wasn’t that easy? [I really have tested this; it runs (and it definitely required debugging)!]