Using Test Driven Development to build a new Forth interpreter

Peter Knaggs

January 5, 2023

Abstract

.

1 Introduction

We are describe a method for brining up a new Forth interpreter from scratch using a Test Driven development approach and the John Hayes test suite[1].

The minimum requirements are quite basic:

1. A simple Data stack
2. A simple Dictionary
3. A few native definitions of just three standard words
4. A native implementation of the test harness
5. A simple interpret loop
6. The ability to read lines from a file

To demonstrate just how simple this approach is an initial system, written in under 500 lines of C, is provided in the appendix.

2 Data Stack

The core system requires a relatively simple data stack. In annex A we see a simple array of integers and a stack pointer to index into the array.

We define four methods that work on this array:

push	Place a new integer value on the top of the stack and increment the stack pointer. This should check for a stack overflow;
pop	Return the top most integer value from the stack and decrement the stack pointer. This should check for a stack underflow;
popLong	Return a double number from the top of the stack;
nip	Remove the item under the top of stack from the stack.

Note that for simplicity of the example, we use a incrementing stack pointer. This stack implementation is very basic and we would expect it to be changed quite significantly during the process.

[1] ftp://ftp.taygeta.com/pub/Forth/Applications/ANS/core.fr

3 Dictionary

This is a simple linked list of word definitions. Each definition has a simple data structure (`XT_t`), which contains the name of the word, a pointer to a procedure (`ptr_func_t`) that does not take any arguments and does not return any value. All communication to the word is via the data stack.

This dictionary is not optimised in any way, it has no knowledge of immediate words, compilation semantics, word lists, or even colon definitions. We assume this rather simple dictionary will be replaced with a more complex data structure during the development, adding the missing features as they are required by the test suite.

3.1 A method of adding a new word to the dictionary

A function to add new native code word to the dictionary. This function is not to be invoked directly from the interpreter but is only intended to initialise the dictionary. It will simply map a word name to a native code function, which it does by adding a `XT_t` to the linked list that makes up the dictionary. For example:

```
AddWord("TESTING", comment);
```

would associate the word "`TESTING`" with the C function `comment`.

3.2 A method of finding a word in the dictionary

A function to step though the linked list, looking to match a dictionary item with a given name. If the name is found, the corresponding data structure (`XT_t`) is returned otherwise a `NULL` is returned indicating the name was not found in the dictionary.

4 Echo

When debugging the scanning of the input source, it is useful to echo the text as it is scanned. A special variable *echo* is defined to enable this behaviour.

We define two words to allow the test harness to control the echoing of the input.

+ECHO Turn echoing on, white space and words are written to the console as they are processed.

–ECHO Disable the echo display.

5 Scanning

We provide three methods to scan and process the input:

nextChar Read the next character from the input file. If this detects the end of the line, it will automatically read the next line from the input file. It is also responsible for outputting the character if the system in in *echo* mode. It will return the character or the special value `EOF` if there is no more text in the file.

```
⟨next char from input⟩ ≡
    char ← line[position]
    increment position
    if char is end of line then
        line ← read line from file
        position ← 0
        increment line number
        if echo then
            println
```

```
            print line number, ": "
        end if
        char ← line[position]
        increment position
    end if

    if echo then
        if char is not white space or char is space or char is tab then
            print char
        end if
    end if

    return char
```
■

nextWord Read the next word from the input, ignoring any leading white space. A word is considered to be any non-white space text. Returns a pointer to the start of the word or NULL if there is no more text in the file.

```
⟨next word from input⟩ ≡
    (Ignore leading white space)
    char ← space
    while char is white space do
        char ← ⟨next char from input⟩
    end while

    if char is EOF (end of file) then
        return null (end of file)
    end if

    (Read name up to next space)
    name ← empty string
    while char is not white space and char is not EOF (end of file) do
        name ← name + char
        char ← ⟨next char from input⟩
    end while

    return name
```
■

parseNumber
Takes two parameters, the text to parse (as returned by the nextWord) and a pointer to an integer where it will put the resultant number. It will attempt to parse the text as a number (using the base value for the radix). If it can parse the number, it will return *true* and place the number in the integer passed as the second parameter, otherwise it will return *false*.

```
⟨parse text as number⟩ ≡
    value ← 0
    sign ← 1
    position ← 0
    min ← ordinal 'A'
    max ← min + base - 10;

    char ← text[position]
    increment position
    if char is '-' then
        sign ← -1
        char ← text[position]
```

```
    increment position
  end if

  while char is not end of text do
    char ← upper case (char)
    if char is digit do
      char ← ordinal char - ordinal '0'
    else if ordinal char ≥ min and ordinal char < max then
      char ← ordinal char - min + 10
    else
      return invalid value
    end if
    value ← (value × base) + char
    char ← text[position]
    increment position
  end while

  number ← value × sign
  return valid value
```
■

6 Forth Words

The Hayes test suite uses three normal Forth words without testing them first. As we are defining the test harness as native words, we need to provide native definitions of these words:

HEX (6.2.1660) Set the number radix (base) to 16.
Note that all numbers in the test suite are given in hexadecimal;

\ (6.2.2535) End of line comment – Ignore the rest of the line;

((6.1.0080) In-line comment – ignore all text up to the next).

7 Test harness

There are two different test harnesses to be considered depending on suite of test being use. The original John Hayes test harness uses {, -> and }. In the Forth200x document Anton Etrl extended the test harness to allow for floating point values, this version uses T{, -> and }T.

As we are only going to use the core tests provided by the Hayes suite we do not actually need the floating point extension.

{ Start a test case, we clear the data stack at the start of the test, resetting the data stack depth back to zero.

```
⟨start test case⟩ ≡
  test start depth ← 0
```
■

-> Save test case. This must save the current data stack in a test stack, record the depth of the stack and reset the data stack.

```
⟨save test case⟩ ≡
  test stack ← data stack
  test end depth ← data stack depth
  data stack depth ← 0
```
■

} End a test case. This is the most complex definition as it must compare the current data stack with the one saved in the test stack and report any differences.

```
⟨end test case⟩ ≡
    match ← (test end depth is data stack depth)
    n ← test start depth
    while match is true and n < test end depth do
      match ← data stack[n] is test stack[n]
    end while

    if match is false then
      println "Stack Mismatch"
      print "Found: "
      for n ← test start depth upto test end depth do
        print test stack[n]
      end for

      println
      print "Expecting: "
      for n ← test start depth upto data stack depth do
        print data stack[n]
      end for
      println
      abort
    end if

    data stack depth ← test start depth
```

■

TESTING Ignore the rest of the line. The harness uses a variable *verbose* to control weather the line is sent to the console or not. We have the *echo* option witch will do the same. You could of course provide an implementation that will send the line to the console even when the *echo* option is disabled.

8 Temporary Definitions

The test suite defines a number of words, both colon definitions and constants, before it has tested these features. Our system is so simple, we can not currently process these definitions therefore, as a temporary measure, we need to comment out these definitions and provide our own (native code) versions.

Once we are past the initial stages of the test suite, it moves on to the defining words. This will require changing the way the dictionary is store, but is also means we can uncomment the definitions and remove our temporary ones, allowing the test suite to operate in the manner originally intended. See section 10 (Procedure) for details.

8.1 Colon Definitions

Thankfully the test suite only defines two helper words in the early stages:

BITSSET? This will test the value on the top of the stack to see it it has a value other than 0. Returning either one 0 or two 0's on the stack:

```
⟨temporary bitsset definition⟩ ≡
    top ← pop()
    push(0)
    if top is 0 then
```

```
    push(0)
  end if
```
■

BITS Counts the number of bits in the value on the top of the stack:

```
⟨temporary bits definition⟩ ≡
  top ← pop()
  count ← 0
  while top is not 0 do
    increment count
    shift top right by 1 bit
  end if
  push(count)
```
■

8.2 Constant Definitions

Similarly we have to comment out the constant definition, replacing them with our own native code versions. Fortunately most languages provide equivalent constant values so the native code versions are relatively simple:

Constant	Forth Definition	Meaning
`0S`	`0`	All bits are zero
`1S`	`0 INVERT`	All bits are one
`<TRUE>`	`1S`	All bits are one
`<FALSE>`	`0S`	All bits are zero
`MSB`	`1S 1 RSHIFT INVERT`	most significant bit only
`MAX-UINT`	`0 INVERT`	maximum unsigned integer
`MAX-INT`	`1S 1 RSHIFT`	maximum signed integer
`MIN-INT`	`1S 1 RSHIFT INVERT`	minimum signed integer
`MID-UINT`	`1S 1 RSHIFT`	mid-point of unsigned integer
`MID-UINT+1`	`1S 1 RSHIFT INVERT`	mid-point of unsigned integer plus one

Once we have implemented and tested the `CONSTANT` definition we can uncomment these constants and remove the temporary definitions.

8.3 Division

The C language does not define whether division is symmetrical or not. So we need to comment out the definition of `IFFLOORED` and `IFSYM`, replacing them with our own native versions that simply ignore the rest of the line. Unfortunately that does mean we also have to provide our own version of the subsequent helper words:

IFFLOORED Ignore rest of line

IFSYM Ignore rest of line

T/MOD Native implementation of `/MOD` (6.1.0240)

T/ `T/MOD NIP`

TMOD `T/MOD DROP`

T*/MOD Native implementation of `*/MOD` (6.1.0110)

T*/ `T*/ NIP`

Again, once we have tested colon-definitions, we can uncomment the `IFFLOORD` and `IFSYM` definitions and remove our temporary definitions.

9 Interpret Loop

Like the dictionary and the stack, the interpret loop is very simple. It has no knowledge of the more advanced features, such as state. These will need to be added as development progresses.

The loop will simply read one name at a time from the input file ⟨*scan next word from input*⟩. It will look the name up in the dictionary ⟨*find name in dictionary*⟩, if the name is found it will execute the associated definition, otherwise it attempts to process the name as a number ⟨*parse name as number*⟩. If it is a valid number, the number is placed on the stack, otherwise a "name not found in dictionary" error is reported.

```
⟨interpret loop⟩ ≡
  begin
    name ← ⟨next word from input⟩
  while name is not null (end of file)
    word ← ⟨find name in dictionary⟩
    if word is not null (name found in dictionary)
      execute word.function (execute word)
    else (word not in dictionary)
      value ← ⟨parse name as number⟩
      if value is valid number then
        push(value)
      else (name not in dictionary or a valid number)
        println
        println "Word not found: ", name
        abort
      end if
    end if
  repeat
```
■

10 Procedure

We are now ready to process the Hayes test suite[2]. Any time the test reports a missing word, the word should be defined and the test suite again. This will allow you to run the following sections of the test suite:

1. Basic Assumptions
2. Booleans: `INVERT AND OR XOR`
3. Shifts: `2* 2/ LSHIFT RSHIFT`
4. Comparisons: `0= = 0< < > U< MIN MAX`
5. Stack operations: `2DROP 2DUP 2OVER 2SWAP ?DUP DEPTH DROP DUP OVER ROT SWAP`
6. Return stack operations: *this has been moved to later in the suite*
7. Add/Subtract: `+ - 1+ 1- ABS NEGATE`
8. Multiplication: `S>D * M* UM*`
9. Division: `FM/MOD SM/REM UM/MOD */ */MOD / /MOD MOD`

The rest of the suite requires fully working versions of the `:` and `CONSTANT` defining words. At this point it would be useful to copy the first 12 tests from section 15 (Defining Words) of the test suite to the top of the test file, allowing basic testing of both words.

[2] `ftp://ftp.taygeta.com/pub/Forth/Applications/ANS/core.fr`

Once `CONSTANT` is defined and tested, it should be possible to uncomment the constant definitions and remove the corresponding native code definitions (8.2). Allowing the constants to be defined by the test suite.

Similarly, when `:` has been defined and tested, it should be possible to uncomment the `IFFLOORED` and `IFSYM` definitions and remove the dependent native code definitions (8.3). Unfortunately we can not uncomment the `BITSSET?` and `BITS` definitions until after section 13 (Flow control).

10. Memory: `HERE , @ ! CELL+ CELLS C, C@ C! CHARS 2@ 2! ALIGN ALIGNED +! ALLOT`
11. Characters: `CHAR [CHAR] [ ] BL S"`
12. Dictionary: `' ['] FIND EXECUTE IMMEDIATE COUNT LITERAL POSTPONE STATE`
6. Return stack operations: `>R R> R@`
13. Flow control: `IF ELSE THEN BEGIN WHILE REPEAT UNTIL RECURSE`

It should now be possible to remove the two temporary colon-definitions (`BITSSET?` and `BITS`) in section 8.1 from our system and allow the test suite to define them.

It should also be noted that we have moved section 9 of the test suite (return stack operations) to just after section 12 (Dictionary).

14. Loops: `DO LOOP +LOOP I J UNLOOP LEAVE EXIT`
15. Defining Words: `: ; CONSTANT VARIABLE CREATE DOES> >BODY`
16. Evaluate: `EVALUATE`
17. Parser input: `SOURCE >IN WORD`
18. Numbers: `<# # #S #> HOLD SIGN BASE >NUMBER HEX DECIMAL`
19. Memory movement: `FILL MOVE`
20. Output: `. ." CR EMIT SPACE SPACES TYPE U.`
21. Input: `ACCEPT`
22. Dictionary Search Rules

Having completed the Hayes test suite, most of the CORE word set from the ANS Forth standard have been implemented and tested. We are now ready to move on to using the more advanced testing as presented in the Gerry Jackson test suite[3] and/or the Forth200x standard[4].

11 Experience

The Test Driven Development approach to developing a new interpreter outlined here has been used to to successfully develop two compilers, one in Java and one in C#. An example of the base code necessary to start this process is given in the appendix. This demonstrates a small initial code size of just under 500 lines of C[5] (ignore comments).

A Code

```
1 #include <stdlib.h>     /* Standard Library: malloc, free, exit */
2 #include <stdarg.h>     /* Variable argument processing: va_list, va_start, va_end */
3 #include <stdio.h>      /* Standard Input/Output: fprintf, vfprintf, stderr, puts, fopen, fclose, fgets, EOF */
4 #include <string.h>     /* String Library: strdup, strchr, strrchr, strcmp, strcpy, strlen, strcat, memset, memcpy */
```

[3] https://github.com/gerryjackson/forth2012-test-suite
[4] https://forth-standard.org/standard/testsuite
[5] https://www.rigwit.co.uk/forth/baseforth.c

```c
#include <ctype.h>      /* Character Library: isgraph, isspace, toupper, isdigit */
#include <limits.h>     /* Constants: INT_MAX, UINT_MAX */

/* Maximum line buffer length */
#define MAXLINE 1024

/* input file name */
static char* filename = NULL;

/* line number within input file */
static int lineNo;

/* file pointer for current input file */
static FILE* fin;

/* input line buffer */
static char* line = NULL;

/* current scanning position within the input line buffer */
static char* pos;

/* current radix (base) for number conversion */
static int base = 10;

/* ========== Error Handling ========== */

/* Forward reference to the freeDict function to free the memory used by the dictionary. */
void freeDict();

/**
 * @brief Report an error message to the standard error and exit the program.
 * The error message may contain parameter place holders with the additional parameters being provide after the
 * message.  This will display the message on the standard error stream, free any allocated memory and exit the
 * program with the exit code.
 * @param code    the exit code.
 * @param format  the error message to be displayed (may contain parameter descriptions).
 * @param ...     any additional parameters required by the format.
 */
void Error(int code, char* format, ...) {
    if (format) {
        fprintf(stderr, "\n%s(%d): ", filename, lineNo);
        va_list vaargs;
        va_start(vaargs, format);
        vfprintf(stderr, format, vaargs);
        va_end(vaargs);
    }

    freeDict();
    free(filename);
    if (line) { free(line); }
    if (fin) { fclose(fin); }
    exit(code);
}

```

```
59  /**
60   * @brief Remove the directory name from a file path.
61   * This will return a pointer to the first character of the last part of the path (or the start of the path, if the path does
62   * not have any directories).
63   * @param filename pointer to the start of the file path.
64   * @return a pointer to the start of the filename within the path.
65   */
66  char* rmDir(char* filename) {
67      char* temp = strrchr(filename, '/');    /* Unix directory separator */
68      if (!temp) {
69          temp = strrchr(filename, '\\');     /* Dos directory separator */
70      }
71      return temp == NULL ? filename : ++temp;
72  }
73
74  /**
75   * @brief Report the program usage, with an error message and a filename that will be displayed after the error
76   * message.  Note this does not free memory so may only be used in the initializations, before the dictionary memory
77   * has been allocated.
78   * @param progname  the program name, may contain a full path name.
79   * @param message   the message to be displayed.
80   * @param filename  the filename causing the error.
81   */
82  void usage(char* progname, char* message, char* filename) {
83      progname = rmDir(progname);
84      char* temp = strchr(progname, '.');
85      if (temp) {
86          *temp = 0;
87      }
88
89      printf("Usage: %s <filename>\n", progname);
90      if (filename) {
91          printf(message, filename);
92      } else {
93          puts(message);
94      }
95      exit(EXIT_FAILURE);
96  }
97
98  /* ========== Data Stack ========== */
99
100 #define MAXSTACK 10
101 int stack[MAXSTACK];    /* Data Stack */
102 int dsp = 0;            /* Data Stack pointer/depth */
103
104 /**
105  * @brief Push a single cell item on to the data stack.
106  * This will report an error if the stack is full.
107  * @param data the item to be placed on the stack.
108  * @return the data item placed on the stack.
109  */
110 int push(int data) {
111     if ( dsp >= MAXSTACK ) {
112         Error(EXIT_FAILURE, "Stack Overflow");
113     }
114     stack[dsp++] = (int) data;
115     return data;
116 }
117
118 /**
119  * @brief Remove the item at the top of the stack and return it.
120  * This will report an error if the stack is empty.
121  * @return the data item at the top of the stack.
122  */
123 int pop() {
124     if ( dsp == 0 ) {
125         Error(EXIT_FAILURE, "Stack Underflow");
126     }
127     return (int) stack[--dsp];
128 }
```

```
129
130 /**
131  * @brief Remove a double cell item from the top of the stack and return it.
132  * @return a double cell item.
133  */
134 long long popLong() {
135     long long top = (long long) pop() << (sizeof(long) * 8);
136     return top | pop();
137 }
138
139 /**
140  * @brief Remove the second item on the data stack.
141  * @return the data item removed from the stack.
142  */
143 int nip() {
144     int data = stack[dsp];
145     stack[--dsp] = data;
146     return data;
147 }
148
149 /* ========== Dictionary ========== */
150
151 /**
152  * @brief A pointer to a function that takes no arguments are does not return a value, i.e., void func().
153  */
154 typedef void (*ptr_func_t)();
155
156 /**
157  * @brief The Execution Token data structure.
158  */
159 struct XT_s {
160     char*           /name;
161     ptr_func_t      /func;
162     struct XT_s*    /next;
163 };
164
165 /**
166  * @brief A pointer to an Execution Token data structure.
167  */
168 typedef struct XT_s* XT_t;
169
170 /**
171  * @brief The head of the dictionary linked list.
172  * A pointer to the most recent XT in the dictionary. Each XT contains a pointer to the next XT in the dictionary
173  * with the last XT in the list holding the NULL for the next value.
174  */
175 static XT_t dict = NULL;
176
177 /**
178  * @brief Free all memory used by the dictionary.
179  * Loop though the dictionary, one entry at a time, and free the memory used by the word name and the XT_s
180  * data structure itself.
181  */
182 void freeDict() {
183     XT_t next = dict;
184     while ( dict != NULL ) {
185         next = dict->next;
186         free(dict->name);
187         free(dict);
188         dict = next;
189     }
190 }
191
192 /**
193  * @brief Add a word into the dictionary.
194  * This will build a new XT data structure which it will place at the head of the dictionary linked list, placing the
195  * current head of the list as the next item in the XT data structure.
196  * @param name  word name to add.
197  * @param func    pointer to c-function to preform the word's action.
198  */
```

```
199 void AddWord(const char* name, ptr_func_t func) {
200     XT_t xt = (XT_t) malloc(sizeof(struct XT_s));
201     xt->func = (ptr_func_t) func;
202     xt->name = strdup(name);
203     xt->next = dict;
204     dict = xt;
205 }
206 
207 /**
208  * @brief Find a word in the dictionary, returning the word's XT data structures or NULL if the word is not found.
209  * This will start at the head of the dictionary and follow the links to each XT in the dictionary until it either finds the
210  * XT with the given name or comes to the end of the linked list.
211  * @param name the word to search for.
212  * @return a pointer to the XT of the word or NULL if not found.
213  */
214 XT_t find(char* name) {
215     XT_t current = dict;
216     while ( current != NULL ) {
217         if ( strcmp(current->name, name) == 0 ) {
218             break;
219         } else {
220             current = current->next;
221         }
222     }
223     return current;
224 }
225 
226 /* ========== Echo ========== */
227 
228 static int echo = 1;
229 
230 void echoOff() { echo = 0; }
231 void echoOn()  { echo = 1; }
232 
233 void initEcho() {
234     AddWord("+ECHO", echoOn);
235     AddWord("-ECHO", echoOff);
236 }
237 
238 /* ========== Scanning ========== */
239 
240 /**
241  * @brief Read the next character from the input file.
242  * If the character is the end of line marker, read the next line from the file and return the first character of the new
243  * line.  If in echo mode write the character to the console, when reading a new line write the line number to the
244  * console.
245  * @return the character or EOF if at the end of the file.
246  */
247 char nextChar() {
248     char c = *pos++;
249     if (c == 0) {
250         if (fgets(line, MAXLINE, fin)) {
251             pos = line;
252             lineNo++;
253             if (strlen(line) + 1 == MAXLINE) {
254                 Error(EXIT_FAILURE, "Line␣too␣long␣for␣buffer␣of␣%d␣characters", MAXLINE);
255             }
256             if (echo) {
257                 printf("\n%4d:␣", lineNo);
258             }
259             c = *pos++;
260         } else {
261             return EOF;
262         }
263     }
264     if (echo && (isgraph(c) || c == '␣' || c == '\t')) {
265         putchar(c);
266     }
267     return c;
268 }
```

```
269
270 /**
271  * @brief Return the next word from the input.
272  * This will ignore any leading white space and return a pointer to the next non white-space character in the input
273  * line.  It will replace the first white-space character after the word name with an end of text marker to convert that
274  * part of the input line into a string.
275  * @return a pointer to the first character of the word.
276  */
277 char* nextWord() {
278     /* Ignore leading white space */
279     char c = ' ';
280     while (isspace(c)) {
281         c = nextChar();
282     }
283
284     if (c == EOF) {
285         return NULL;
286     }
287
288     /* Read name up to next space */
289     char* name = pos - 1;
290     while (!isspace(c) && c != EOF) {
291         c = nextChar();
292     }
293
294     /* Mark end of word */
295     *(pos - 1) = 0;
296
297     return name;
298 }
299
300 /**
301  * @brief Attempt to convert text into a number using the current base.
302  * This will attempt to convert the string in text into a number using the value is base as the radix.  If successful the
303  * number will be placed in the integer pointed to by the number parameter and a true value is returned otherwise a
304  * false value is returned.
305  * @param text      the text to be parsed.
306  * @param number  a pointer to a location where the number can be stored.
307  * @return true if the text is a number or false if not.
308 */
309 int parseNumber(char* text, int* number) {
310     int value = 0;
311     int sign = 1;
312
313     char c = *text++;
314     if (c == '-') {
315         sign = -1;
316         c = *text++;
317     }
318
319     while (c > 0) {
320         c = toupper(c);
321         if (isdigit(c)) {
322             c = c - '0';
323         } else if (c >= 'A' && c < 'A' + base - 10) {
324             c = c - 'A' + 10;
325         } else {
326             return 0;   /* Not a valid number */
327         }
328         value *= base;
329         value += c;
330         c = *text++;
331     }
332
333     *number = (value * sign);
334     return c == 0;
335 }
```

```
336
337 /* ========== Forth Words ========== */
338
339 /**
340  * @brief Set BASE to 16
341  */
342 void hex() { base = 16; }
343
344 /**
345  * @brief Ignore all text up until the end of the line.
346  */
347 void comment() {
348     int len = strlen(pos);
349     while (len-- > 0) {
350         nextChar();
351     }
352 }
353
354 /**
355  * @brief In-line comment - ignore all text up until the next ).
356  */
357 void parn() {
358     char c;
359     do {
360         c = nextChar();
361     } while (c != EOF && c != ')');
362 }
363
364 /**
365  * @brief Add the Forth words HEX, \ and ( to the dictionary.
366  */
367 void initForth() {
368     AddWord("HEX",   hex);
369     AddWord("\\",    comment);
370     AddWord("(",     parn);
371 }
372
373 /* ========== Test Harness ========== */
374
375 static int testStack[MAXSTACK]; /* Test stack */
376 static int tend;
377
378 /**
379  * @brief Start a test, start with a clean data stack.
380  */
381 void testStart() {
382     dsp = 0;
383 }
384
385 /**
386  * @brief Save the test stack.
387  * Copy the current data stack to the test stack.
388  */
389 void testSave() {
390     memset(testStack, 0, sizeof(int) * MAXSTACK);
391     memcpy(testStack, stack, sizeof(int) * dsp);
392     tend = dsp;
393     dsp = 0;
394 }
395
396 /**
397  * @brief End a test.
398  * Compare the current data stack with the test data stack.
399  * Report an error if the two stacks to not match exactly.
400  */
```

```
401 void testEnd() {
402     int match = (tend == dsp);
403     for (int n = 0; (match && n < tend); n++) {
404         match = (stack[n] == testStack[n]);
405     }
406 
407     if ( !match ) {
408         fprintf(stderr, "\n%s(%d): Stack Mismatch\n", filename, lineNo);
409         fprintf(stderr, "Found:     ");
410         for (int n = 0; n < tend; n++) {
411             fprintf(stderr, "%d ", testStack[n]);
412         }
413 
414         fprintf(stderr, "\nExpecting: ");
415         for (int n = 0; n < dsp; n++) {
416             fprintf(stderr, "%d ", stack[n]);
417         }
418 
419         fprintf(stderr, "\n");
420         Error(EXIT_FAILURE, NULL);
421     }
422     dsp = 0;
423 }
424 
425 /* ==== Temporary colon definitions to be removed once : is defined and tested */
426 /* { : BITSSET? IF 0 0 ELSE 0 THEN ; -> } */
427 void bittest() {
428     int top = pop();
429     push(0);
430     if (top) {
431         push(0);
432     }
433 }
434 
435 /* : BITS ( x -- u ) 0 SWAP BEGIN DUP WHILE MSB AND IF >R 1+ R> THEN 2* REPEAT DROP ; */
436 void bits() {
437     unsigned int top = pop();
438     int count = 0;
439     while (top) {
440         count++;
441         top >>= 1;
442     }
443     push(count);
444 }
445 
446 /* ==== Temporary CONSTANT definitions to be removed once CONSTANT is defined and tested */
447 void zeros()        { push(0); }                    /* 0                          */
448 void ones()         { push(~0); }                   /* 0 INVERT                   */
449 void cfalse()       { push(0); }                    /* 0S                         */
450 void ctrue()        { push(~0); }                   /* 1S                         */
451 void msb()          { push(~INT_MAX); }             /* 1S 1 RSHIFT INVERT         */
452 
453 /* The Comparison operators define a number of constants */
454 /* MSB, MAX-UINT, MAX-INT, MIN-INT, MID-UINT, MID-UINT+1 */
455 void maxUInt()      { push(UINT_MAX); }             /* 0 INVERT                   */
456 void maxInt()       { push(INT_MAX); }              /* 0 INVERT 1 RSHIFT          */
457 void minInt()       { push(INT_MIN); }              /* 0 INVERT 1 RSHIFT INVERT */
458 void midUInt()      { push(UINT_MAX >> 1); }        /* 0 INVERT 1 RSHIFT          */
459 void midUI1()       { push(~(UINT_MAX >> 1)); }     /* 0 INVERT 1 RSHIFT INVERT */
460 
461 /* C can use either Floored or Symmetric division */
462 /* The following temporary colon definitions can be removed once : is defined and tested */
463 int isFloored() {
464     return (-3 / 2) == -1;
465 }
466 
```

```
467 /* IFFLOORED    : T/MOD  >R S>D R> FM/MOD ; */
468 /* IFSYM        : T/MOD  >R S>D R> SM/REM ; */
469 void tdm() {
470     int top = pop();
471     long long nos = popLong();
472 
473     long long div;
474     int rem;
475 
476     if ( isFloored() ) {
477         div = nos / top;
478         rem = (int) (nos - (div * top));
479     } else {
480         div = nos / top;
481         rem = (int) (nos % top);
482     }
483 
484     push(rem);
485     push((int) div);
486 }
487 
488 /* IFFLOORED    : T/   T/MOD SWAP DROP ; */
489 /* IFSYM        : T/   T/MOD SWAP DROP ; */
490 void td() { tdm(); nip(); }
491 
492 /* IFFLOORED    : TMOD   T/MOD DROP ; */
493 /* IFSYM        : TMOD   T/MOD DROP ; */
494 void tm() { tdm(); pop(); }
495 
496 /* IFFLOORED    : T*_/MOD >R M* R> FM/MOD ; */
497 /* IFSYM        : T*_/MOD >R M* R> SM/REM ; */
498 void tsdm() {
499     int top = pop();
500     long long a = pop();
501     long long b = pop();
502     long long mul = a * b;
503 
504     long long div;
505     int rem;
506 
507     if ( isFloored() ) {
508         div = mul / top;
509         rem = (top - (int) div * top);
510     } else {
511         div = mul / top;
512         rem = mul % top;
513     }
514 
515     push(rem);
516     push((int) div);
517 }
518 
519 /* IFFLOORED    : T*_/ T*_/MOD SWAP DROP ; */
520 /* IFSYM        : T*_/ T*_/MOD SWAP DROP ; */
521 void tsd() { tsdm(); nip(); }
522 
523 /**
524  * @brief Initialise the dictionary with the test harness and temporary definitions.
525  */
526 void initTest() {
527     /* Hayes test harness { -> } */
528     AddWord("{",                testStart);
529     AddWord("->",               testSave);
530     AddWord("}",                testEnd);
531     /* Forth200x test harness T{ -> }T */
532     AddWord("T{",               testStart);
533     AddWord("}T",               testEnd);
```

```
534     AddWord("TESTING",          comment);
535
536     /* Temporary colon definitions (Basic assumptions and Memory) */
537     AddWord("BITSSET?",         bittest);
538     AddWord("BITS",             bits);
539
540     /* Temporary constant definitions */
541     AddWord("0S",               zeros);
542     AddWord("1S",               ones);
543     AddWord("<TRUE>",           ctrue);
544     AddWord("<FALSE>",          cfalse);
545
546     /* Comparison operators use the following constants */
547     AddWord("MSB",              msb);
548     AddWord("MAX-UINT",         maxUInt);
549     AddWord("MAX-INT",          maxInt);
550     AddWord("MIN-INT",          minInt);
551     AddWord("MID-UINT",         midUInt);
552     AddWord("MID-UINT+1",       midUI1);
553
554     /* Temporary colon definitions (Division) */
555     AddWord("IFFLOORED",        comment); /* Ignore rest of line */
556     AddWord("IFSYM",            comment); /* Ignore rest of line */
557     AddWord("T/MOD",            tdm);
558     AddWord("T/",               td);
559     AddWord("TMOD",             tm);
560     AddWord("T*/MOD",           tsdm);
561     AddWord("T*/",              tsd);
562 }
563
564 /* ========== Main ========== */
565
566 /**
567  * @brief Open the file given in as the command line argument.
568  * This will process all of the command line arguments, checking that there is only one.  It will attempt to open the
569  * file, if not able to it will add the .forth extension to the file and try again, if the file is still not found it will try the
570  * .fr extension.  If successful the fin and filename global variables configured, otherwise it will report a usage error
571  * and exit.
572  * @param argc  the number of arguments contained in the argv array.
573  * @param argv  an array of strings, one for each command line argument.
574  */
575 void openFile(int argc, char *argv[]) {
576     /* Check we have the right number of arguments */
577     if ( argc == 1 ) {
578         usage(argv[0], "We need a file to process!", NULL);
579     } else if ( argc > 2 ) {
580         usage(argv[0], "Can only process one file at a time", NULL);
581     }
582     filename = argv[1];
583
584     /* Process the file name (allow for ".forth" extension) */
585     int len = strlen(filename) + 10;
586     char* name = (char*) malloc(len * sizeof(char));
587     strcpy(name, filename);
588
589     /* Does the file exist? */
590     fin = fopen(name, "r");
591     if ( !fin ) {
592         /* File not found, try ".forth" extension */
593         strcat(name, ".forth");
594         fin = fopen(name, "r");
595     }
596
597     if ( !fin ) {
598         /* Still not found, try ".fr" extension */
599         strcpy(name, filename);
600         strcat(name, ".fr");
601         fin = fopen(name, "r");
602     }
603
```

```
604     if ( !fin ) {
605         /* Still not found, give in */
606         usage(argv[0], "Can␣not␣open␣input␣file:␣\"%s\"", filename);
607     }
608 
609     filename = strdup(rmDir(name));
610     free(name);
611 }
612 
613 /**
614  * @brief Initialise the system.
615  * First it will attempt to process the command line arguments. It will then initialise the dictionary before initialising
616  * the line buffer so that the first call to nextChar will load the first line of the file into the buffer.
617  * @param argc  the number of command line arguments in the argv array.
618  * @param argv  an array of strings, one for each command line argument.
619  */
620 void init(int argc, char* argv[]) {
621     /* Open input file */
622     openFile(argc, argv);
623 
624     /* Initialise dictionary */
625     initEcho();
626     initTest();
627     initForth();
628 
629     /* Initialise line buffer */
630     line = (char*)malloc(MAXLINE * sizeof(char));
631     pos = line;
632     *pos = 0;
633     lineNo = 0;
634 }
635 
636 /**
637  * @brief The main interpret loop.
638  * This will read the input file, one word at a time, it will look up each word in the dictionary and if found it will
639  * preform the action associated with the word, otherwise it will attempt to convert the word into a number (using the
640  * current base).  If successful it will place the number on the data stack otherwise it will report a word not found
641  * error and abort.
642  * @param argc  the number of command line arguments in the argv array.
643  * @param argv  an array of strings, one for each command line argument.
644  * @return does not return
645  */
646 int main(int argc, char* argv[]) {
647     init(argc, argv);
648 
649     /* Interpret loop */
650     char* name;
651     while (name = nextWord()) {
652         XT_t word = find(name);     /* Lookup name in dictionary */
653         if (word) {                 /* if name found */
654             word->func();           /* Execute XT*/
655         } else {                    /* Name not found */
656             int value;              /* convert to number */
657             if (parseNumber(name, &value)) {
658                 push(value);        /* Push number onto stack */
659             } else {                /* Not a word or a number */
660                 Error(EXIT_FAILURE, "Word␣not␣found:␣%s", name);
661             }
662         }
663     }
664     Error(EXIT_SUCCESS, NULL);
665 }
```