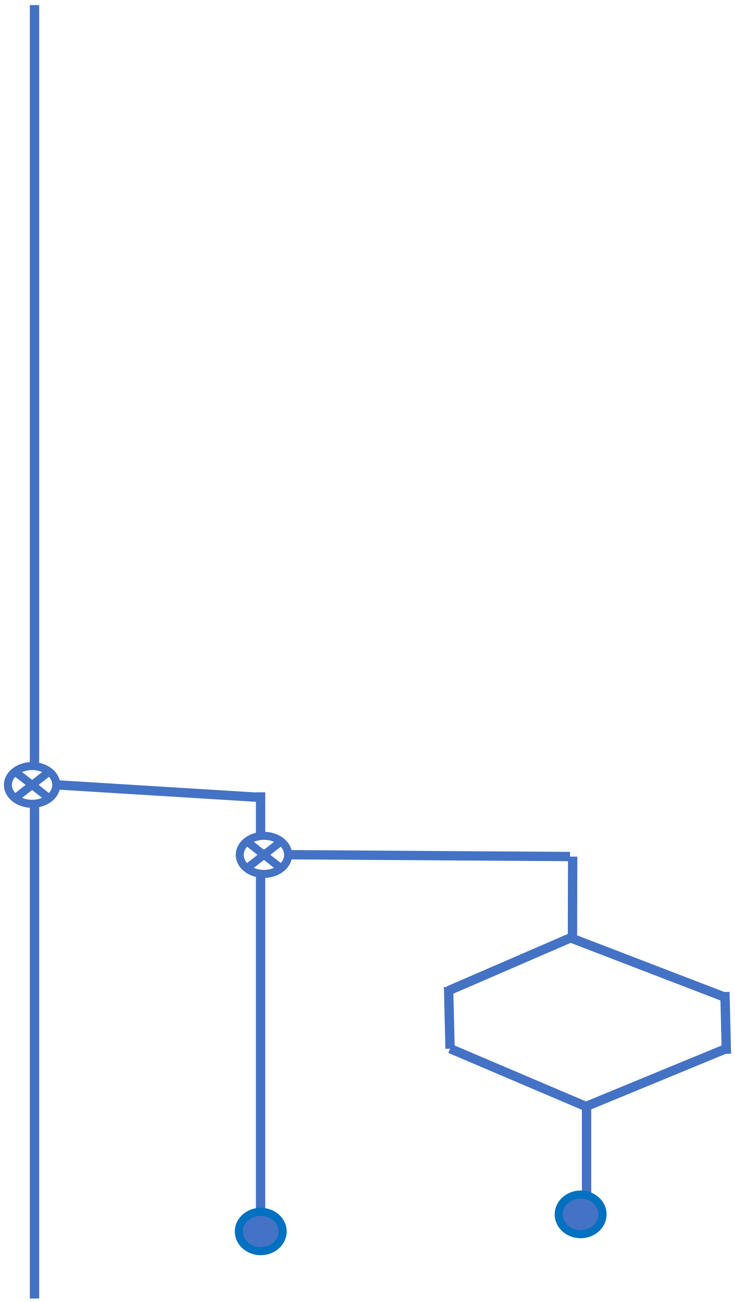


Program Debugging

SVFIG
Aug. 28, 2021
Bill Ragsdale

The Need

Here are fourteen tools to use for testing. I certainly don't use them all every time. Just pick and choose.

Built on Win32Forth.

Draws on '`see`', '`view`' and '`debug`'.

Onward . . . .

Fourteen Tools To Success

1. Use command line input.
2. Write as a definition; test compilation.
3. Test from the command line.
4. Rewrite the code showing parameters.
5. Forth ‘`see`’.
6. Forth ‘`view`’.
7. Create a data test set.
8. Add breakpoints using ‘exit’.
9. Add `.s` internally.
10. Active test reporting `[IF] [ELSE] [THEN].`
11. Add error trapping using ‘abort”’.
12. Integrate testing with a wrapper word.
13. Use ‘`debug'`, directly or internally.
14. Selective compilation: `[IF] [ELSE] [THEN].`

1. Quick Command Line Test

Let us say I have a new Forth system or have made significant low level changes.

```
I want to test:  + - * /
```

Quick Command Line Test

Let us say I have a new Forth system or have made significant low level changes.

```
I want to test:  + - * /
```

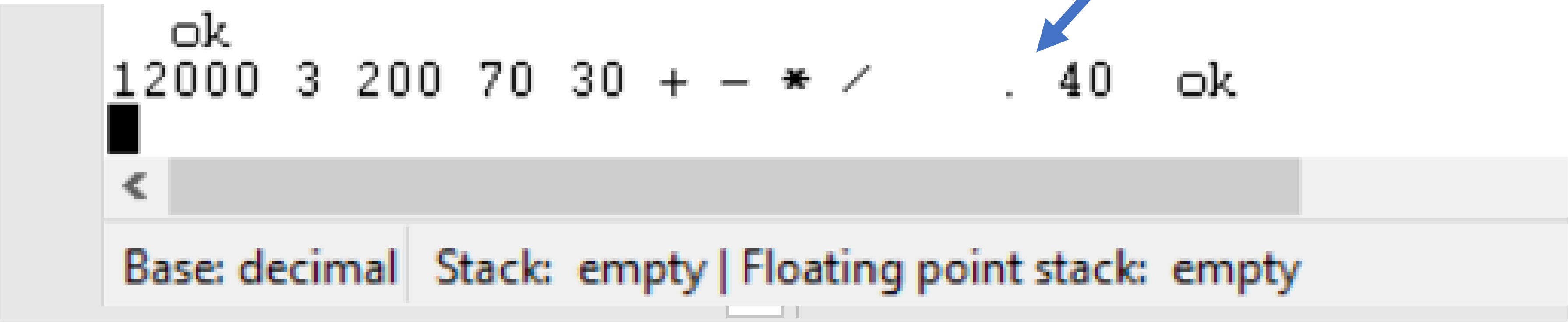
Upon any problems, I’ll have to review my low level code and debug.

Quick Command Line Test

I want to test: + - * /

Use five integer values and expect to see ‘40’.

DOT

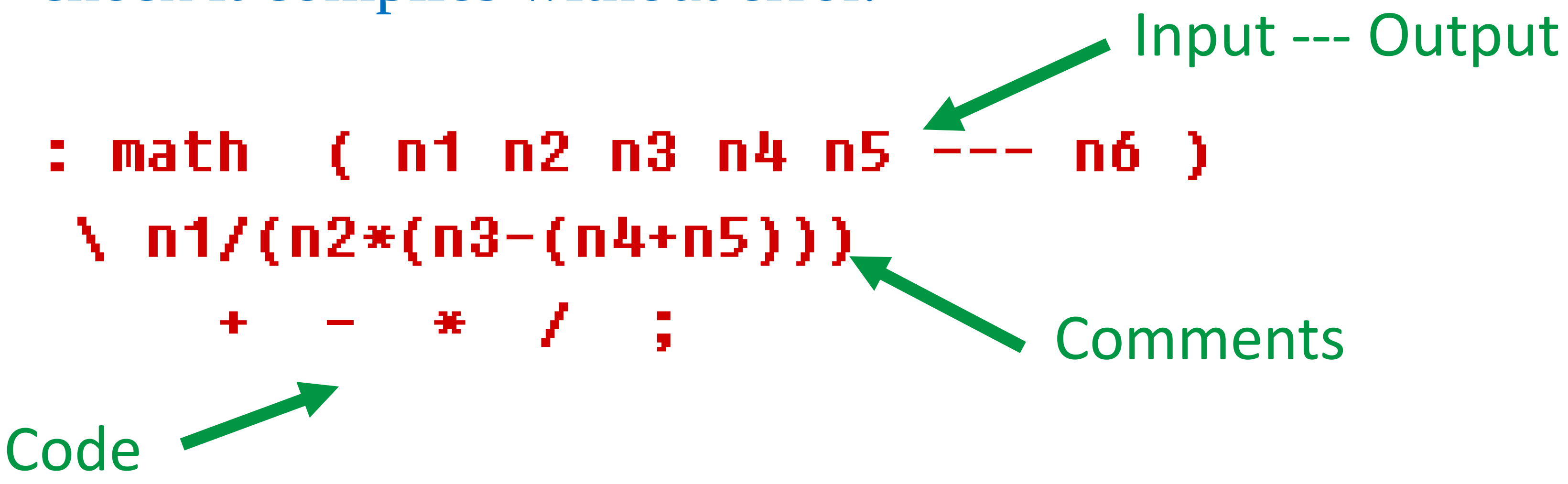


```
  ok
12000 3 200 70 30 + - * /      . 40   ok
```

Base: decimal | Stack: empty | Floating point stack: empty

2. Form A Definition

Write in the accepted source code format and check it compiles without error.



Input --- Output

```
: math  ( n1 n2 n3 n4 n5 --- n6 )
 \ n1/(n2*(n3-(n4+n5)))
     +  -  *  /  ;
```

Comments

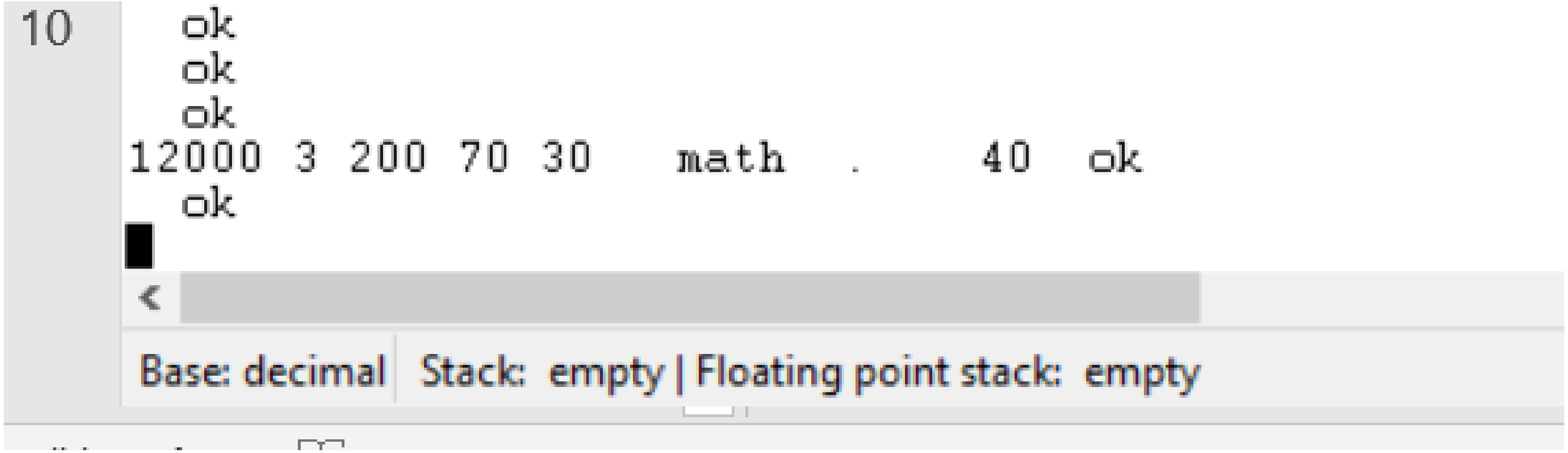
Code

For repeated testing ‘math’ does the testing of the four math operators.

3. Less Typing; Fewer Errors

For repeated testing ‘math’ does the testing of the four math operators.

```
12000 3 200 70 30    math   .    <enter>
```



```
10     ok
       ok
       ok
    12000 3 200 70 30   math   .    40  ok
       ok
```

Base: decimal | Stack: empty | Floating point stack: empty

And expect to see: 40

4. Rewrite Showing Stack Values

Still trouble? Add in the stack actions as comments. Helps when you return much later.

```
: math ( n1 n2 n3 n4 n5 --- n6 )
 \ n1/(n2*(n3-(n4+n5)))
         \ n1 n2 n3 n4 n5
   +     \ n1 n2 n3 n4+n5
   -     \ n1 n2 n3-(n4+n5)
   *     \ n1 n2*(n3-(n4+n5))
   /   ; \ n1/(n2*(n3-(n4+5)))
```

This well may correct for mental errors on the parameter execution order.

5. See The Compiled Code

Enter: 'see math' and see the definition decompiled from its object code in memory.

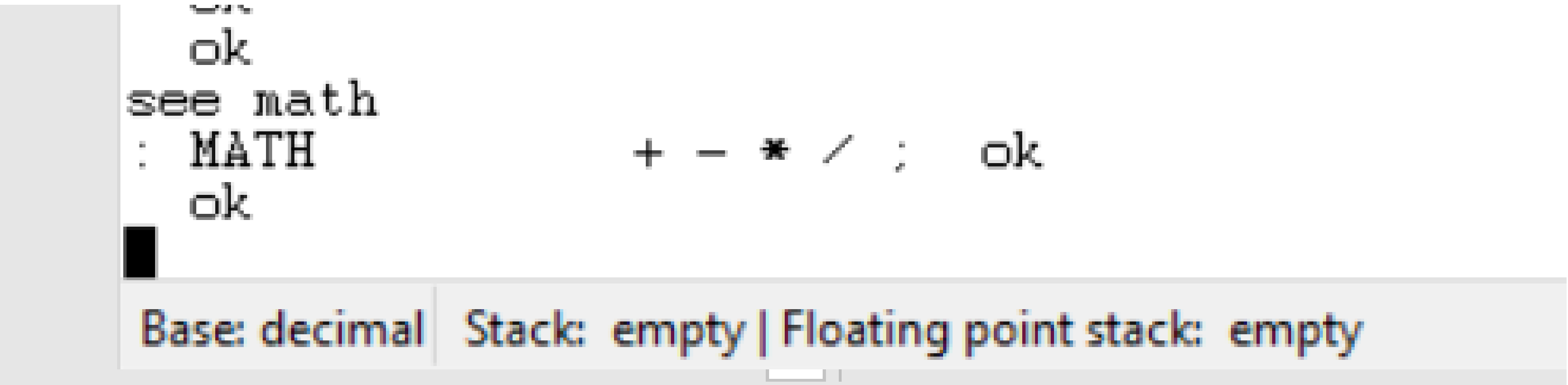
```
see math  <enter>
```

See The Compiled Code

Enter: ‘see math’ and see the definition decompiled from its object code in memory.

```
see math  <enter>
```

Is this what we intended?



```
   ok
see math
: MATH                + - * / ;   ok
   ok
```

Base: decimal | Stack: empty | Floating point stack: empty

6. View The Source Code

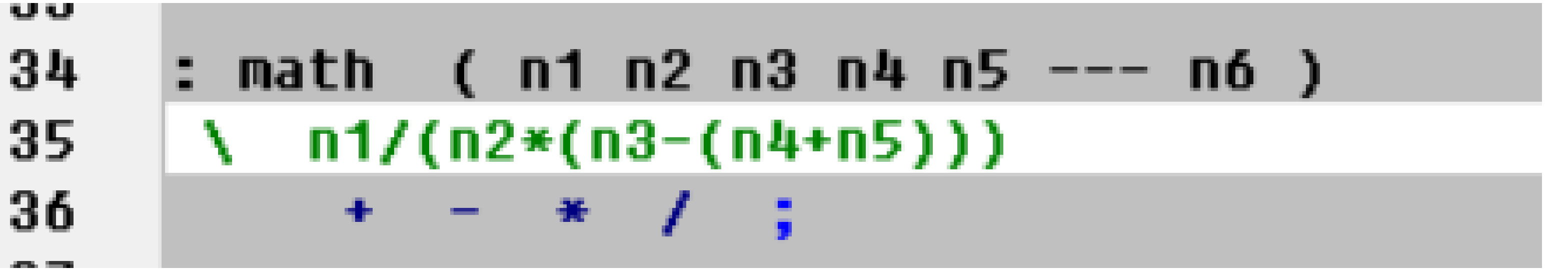
Enter: ‘view math’ and see the source code in its file.

```
view math  <enter>
```

6. View The Source Code

Enter: ‘view math’ and see the source code in its file.

```
view math  <enter>
```



```
: math  ( n1 n2 n3 n4 n5 --- n6 )
 \   n1/(n2*(n3-(n4+n5)))
        +   -   *   /   ;
```

7. Using A Data Test Set

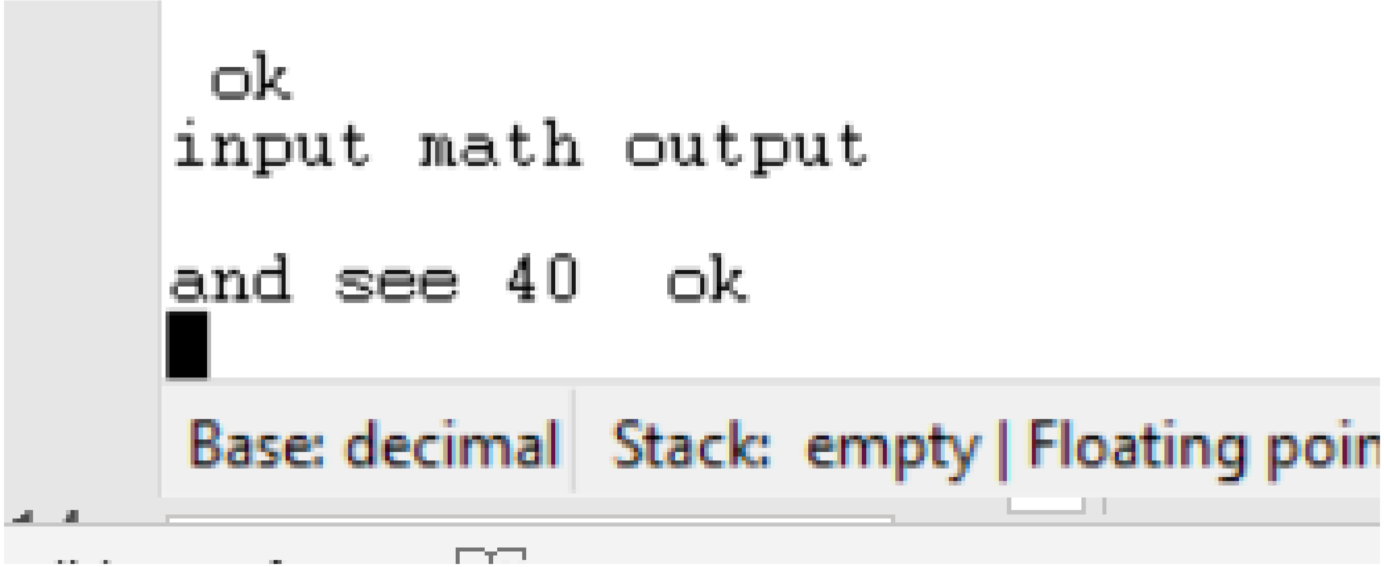
Create words to support testing.

```
: input 12000 3 200 70 30   ;

: output   ." and see "   .   ;
```

7. Using A Data Test Set

Create words to support testing.

```
: input 12000 3 200 70 30   ;

: output   ." and see "   .   ;
```



```
 ok
input math output

and see 40  ok
```

Base: decimal | Stack: empty | Floating poin

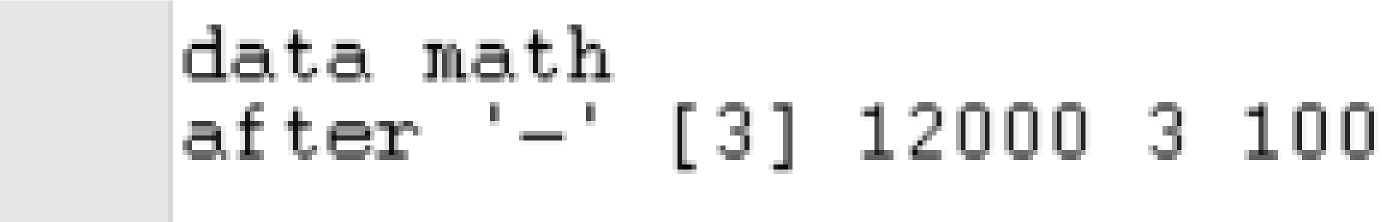
8. A Simple Breakpoint

Use ‘`exit`’ to halt execution and ‘`.s`’ to see the stack contents at the point.

```
: math
   +  -
   cr ." after '-' "  .s exit
   *  /    ;
```

8. A Simple Breakpoint

```
: math
   +   -
   cr ." after '-' "  .s exit
   *  /    ;

data math
```



```
data math
after '-' [3] 12000 3 100
```

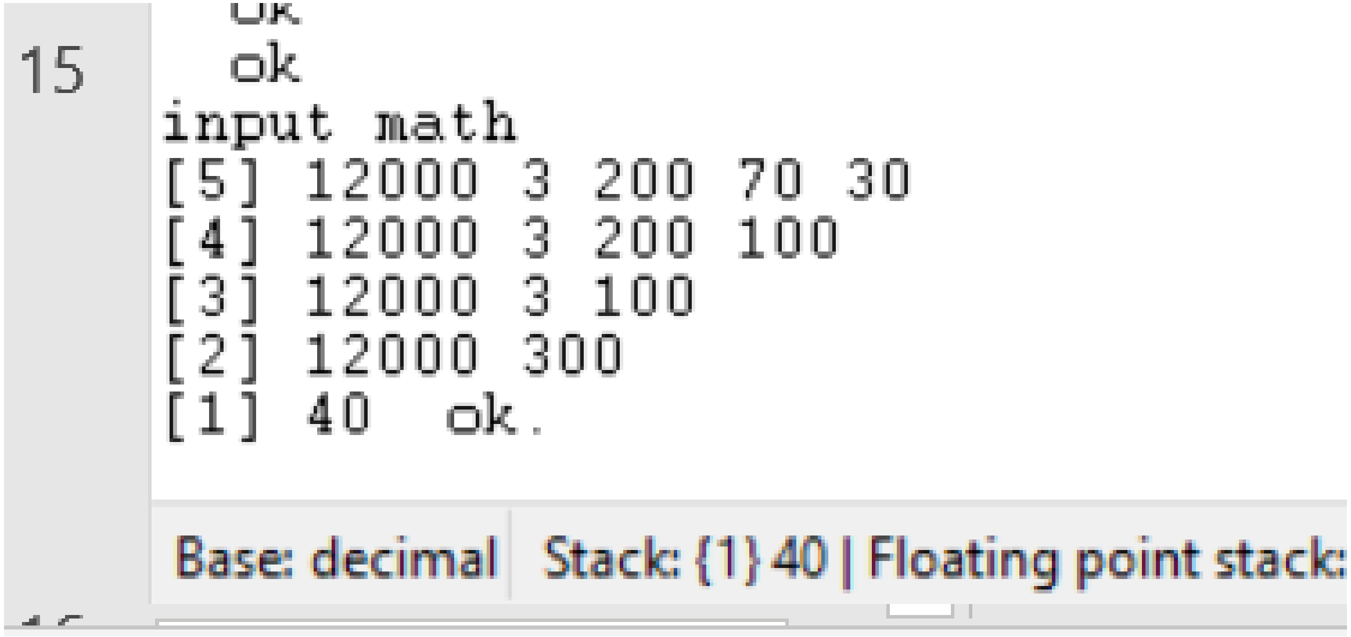
9. Creative Use of .s

Add ‘.s’ to show the stack contents during execution. This is a substitute for a tracing word like ‘debug’.

```
: math      cr  .s
       +    cr  .s
       -    cr  .s
       *    cr  .s
       /    cr  .s   ;
```

Creative Use of .s

```
: math     cr  .s
        +   cr  .s
        -   cr  .s
        *   cr  .s
        /   cr  .s   ;
```

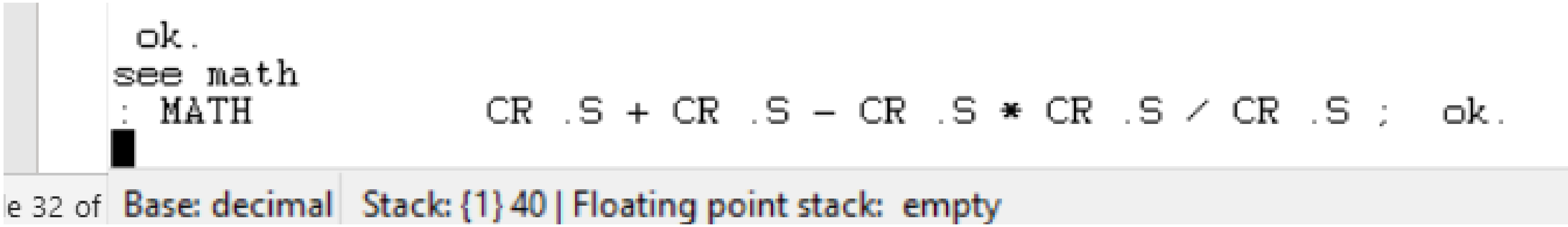


```
   ok
input math
[5] 12000 3 200 70 30
[4] 12000 3 200 100
[3] 12000 3 100
[2] 12000 300
[1] 40  ok.
```

Base: decimal | Stack: {1} 40 | Floating point stack:

Creative Use of .s

```
: math      cr  .s
        +   cr  .s
        -   cr  .s
        *   cr  .s
        /   cr  .s   ;
```



```
 ok.
see math
: MATH          CR .S + CR .S - CR .S * CR .S / CR .S ;  ok.
```

e 32 of | Base: decimal | Stack: {1} 40 | Floating point stack: empty

10. Active use of [IF] [THEN]

Make an error report itself with conditional text.

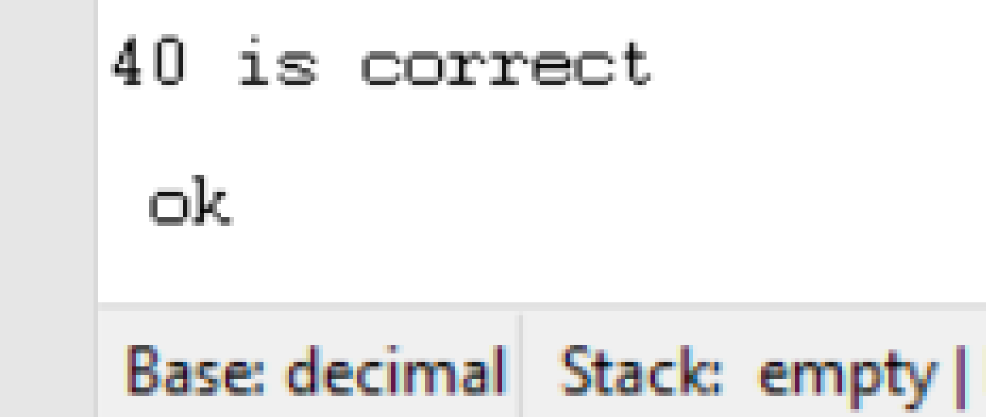
```
input math dup . 40 = [IF] .( is correct)
                    [ELSE] .( is incorrect ) [THEN]
```

```
40 = [IF]   says ‘is correct’
     [ELSE] says ‘is incorrect’
```

10. Active use of [IF] [THEN]

```
input math dup . 40 = [IF] .( is correct)
                     [ELSE] .( is incorrect ) [THEN]
```

```
40 = [IF]   says ‘is correct’
     [ELSE] says ‘is incorrect’
```



```
40 is correct
 ok
```

```
Base: decimal | Stack:  empty |
```

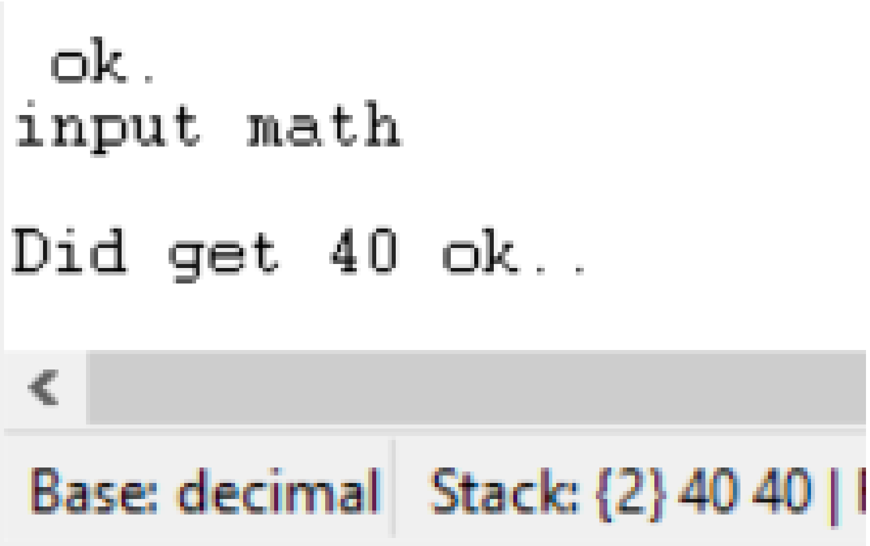
11. Add abort" As Error Test

Insert ‘abort”’ with a preceding test. Another form of breakpoint.

```
: math   +  -  *  / dup 40 <>
   cr abort" Expected 40 "
   cr ." Did get 40" ;
```

11. Add abort" As Error Test

```
: math    +   -   *   / dup 40 <>
    cr abort" Expected 40 "
    cr ." Did get 40" ;
```



```
 ok.
input math

Did get 40 ok..
```

Base: decimal | Stack: {2} 40 40 |

12. Integrate With A Wrapper

Combine ‘`input`’ ‘`math`’ ‘`output`’ into a 'wrapper' word.

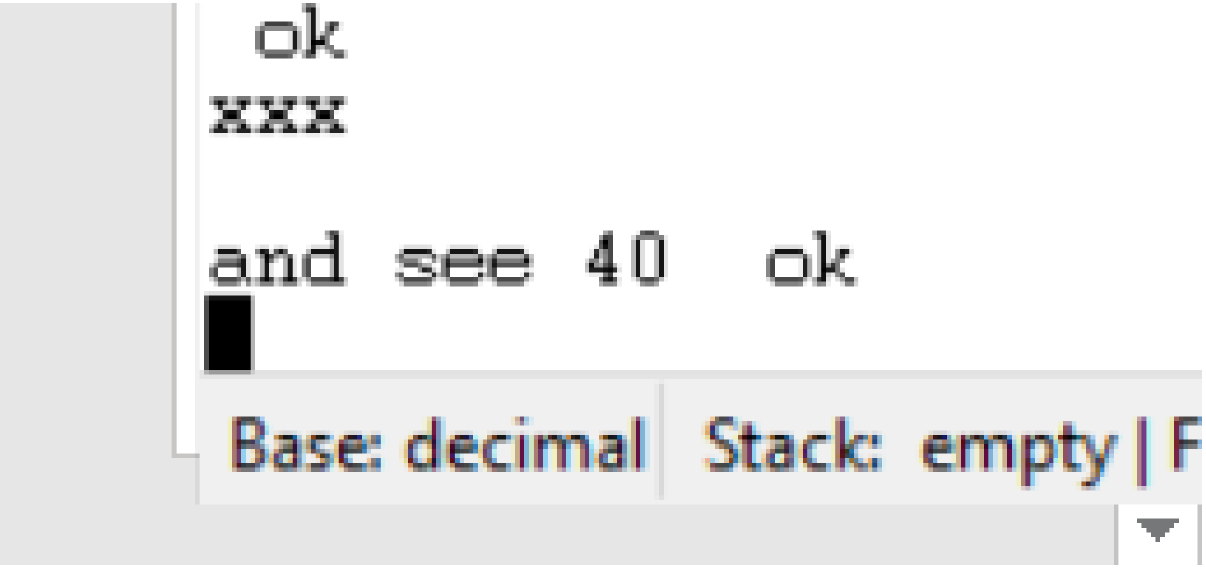
For repeated testing it is easer to type one word.

```
: xxx input math output ;
```

12. Integrate With A Wrapper

Combine ‘**`input`**’ ‘**`math`**’ ‘**`output`**’ into a 'wrapper' word.
For repeated testing it is easer to type one word.

```
: xxx input math output ;
```



```
  ok
xxx

and see 40   ok
```

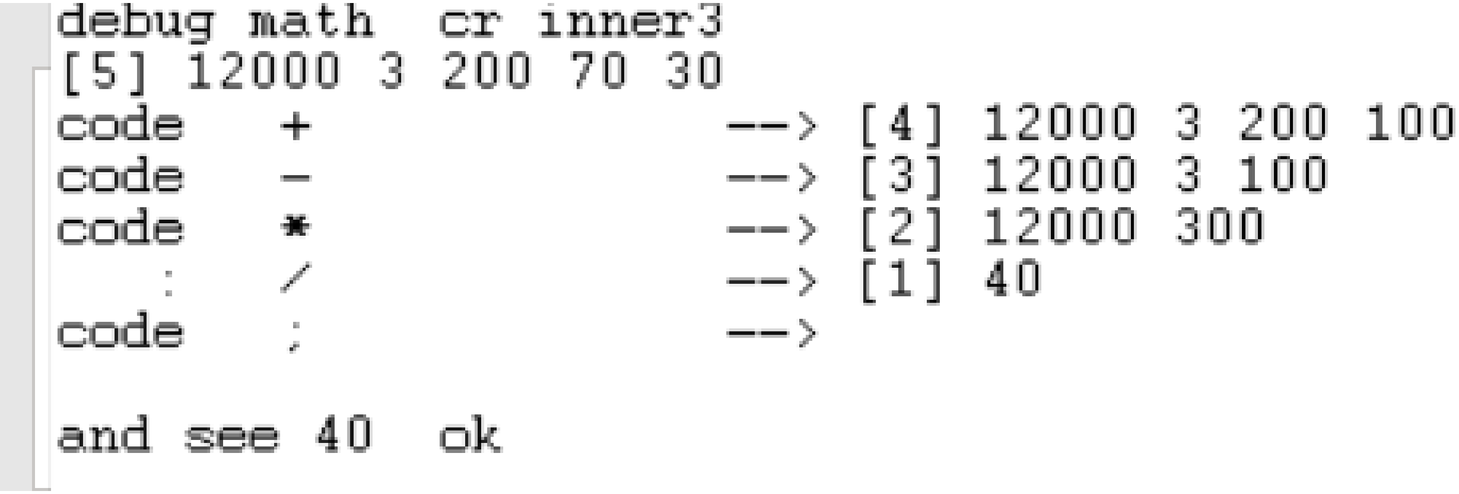
13. Debug Internally

Win32F 'debug' is powerful. It can trace from direct console input or upon a lower level word used within other words.

```
: inner1 input math output ;
: inner2 inner1 ;
: inner3 inner2 ;

debug math   inner3
```

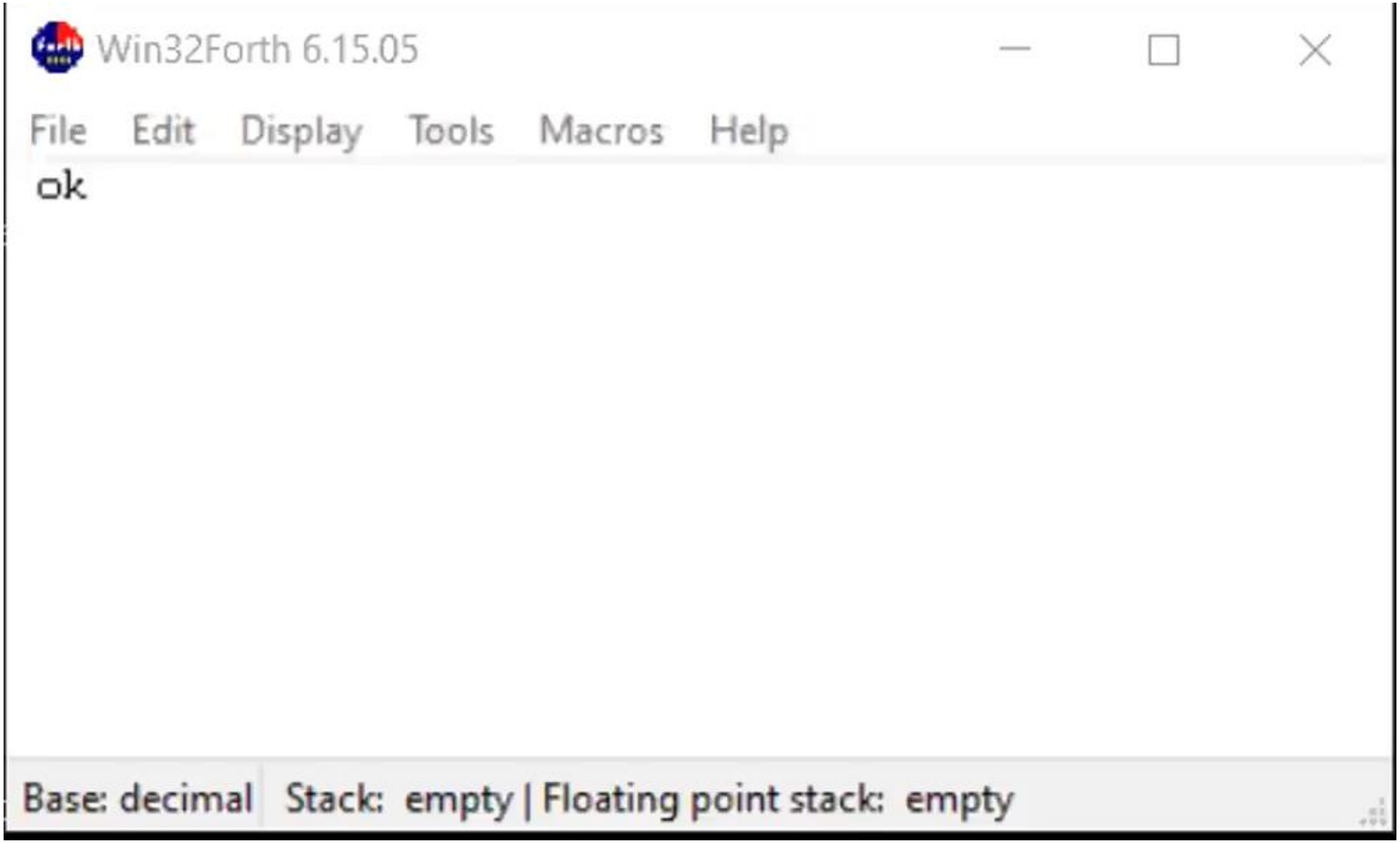
13. Debug Internally

```
: inner1 input math output ;
: inner2 inner1 ;
: inner3 inner2 ;
debug math   inner3
```



```
debug math  cr inner3
[5] 12000 3 200 70 30
code   +             --> [4] 12000 3 200 100
code   -             --> [3] 12000 3 100
code   *             --> [2] 12000 300
   :   /             --> [1] 40
code   ;             -->

and see 40  ok
```

13. Debug Internally



```
ok
```

Base: decimal Stack: empty | Floating point stack: empty

14. [IF] [ELSE] [THEN]

You can use **[IF] [ELSE] [THEN]** to selectively include tests within a compiled word.

test? is an immediate word controlling the following **[IF] . . . [THEN]** to include a **'cr .s'** print stack command in the compiled output.

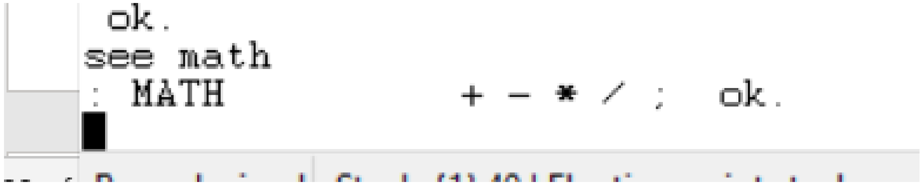
```
: math      test? [if] cr .s [then]
         +  test? [if] cr .s [then]
         -  test? [if] cr .s [then]
         *  test? [if] cr .s [then]
         /  test? [if] cr .s [then] ;
```

14. [IF] [ELSE] [THEN]

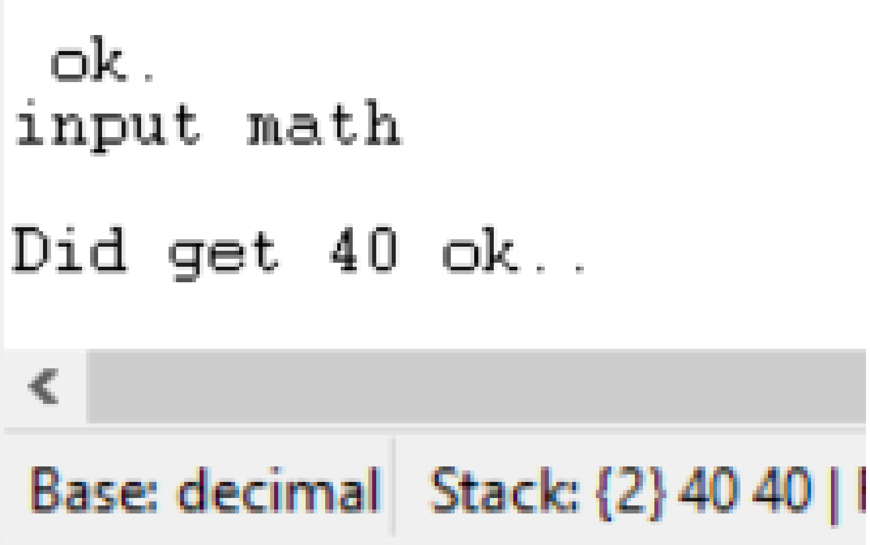
```
true value test? immediate

: do-tests  true  to test?  ;
: no-tests  false to test?  ;

do-tests
: math      test? [if] cr .s [then]
         +  test? [if] cr .s [then]
         -  test? [if] cr .s [then]
         *  test? [if] cr .s [then]
         /  test? [if] cr .s [then] ;
```

14. [IF] [ELSE] [THEN]

With no-tests, math only



```
 ok.
see math
: MATH            + - * / ;  ok.
```

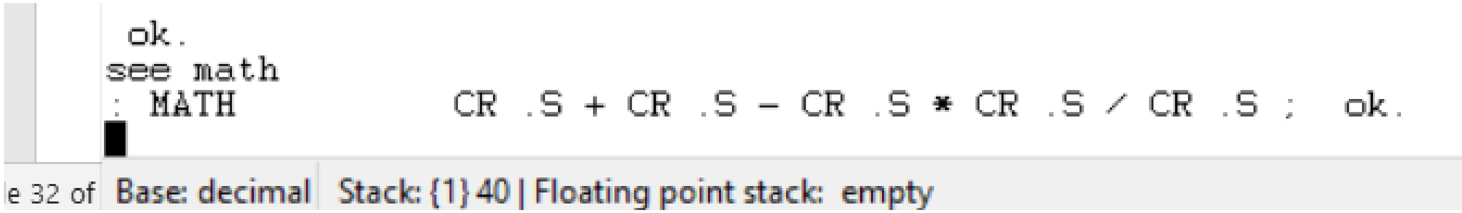


```
 ok.
input math

Did get 40 ok..
```

Base: decimal | Stack: {2} 40 40 |

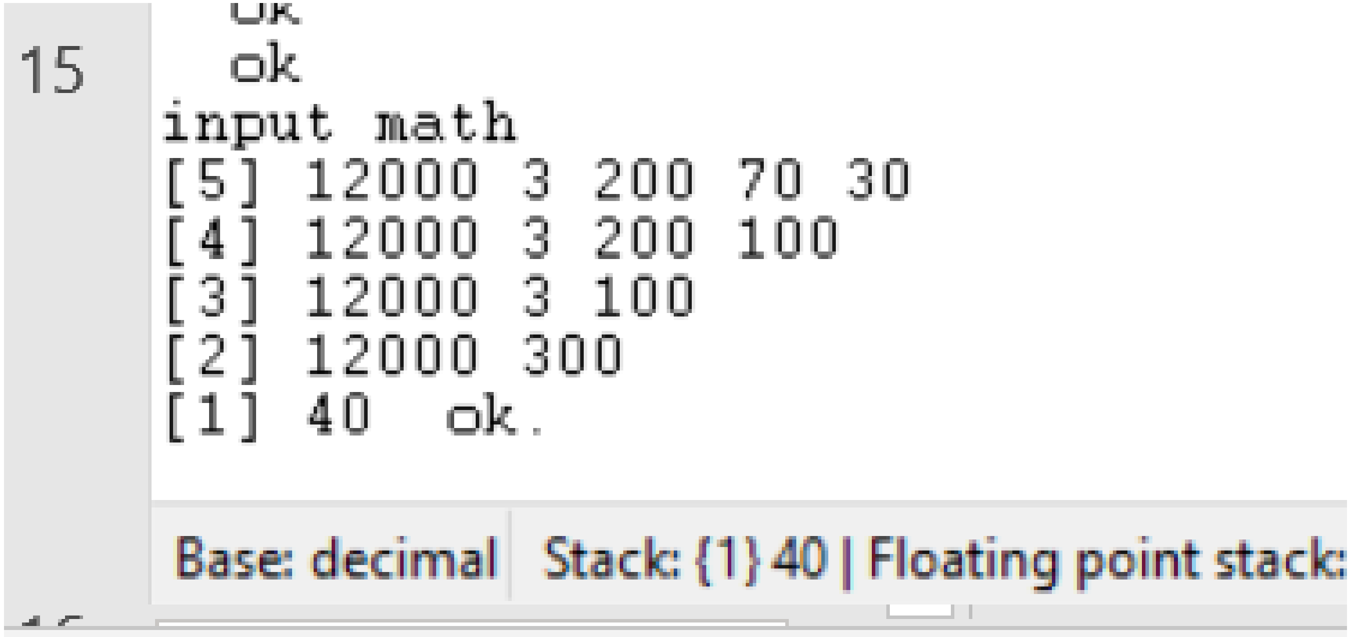
14. [IF] [ELSE] [THEN]

With do-tests, showing 'cr .s' diagnostic.



```
 ok.
see math
: MATH          CR .S + CR .S - CR .S * CR .S / CR .S ;  ok.
```

```
e 32 of   Base: decimal   Stack: {1} 40 | Floating point stack:  empty
```



```
   ok
   ok
input math
[5] 12000 3 200 70 30
[4] 12000 3 200 100
[3] 12000 3 100
[2] 12000 300
[1] 40  ok.
```

```
Base: decimal   Stack: {1} 40 | Floating point stack:
```

Benefits

Keep a variety of testing and debugging methods in your Forth repertoire.

I used to insert stack dumps and exits at suspected problem points. Now, I mostly use ‘`debug`’ for a full word trace.

I took me a couple of years to discover ‘`debug`’ as Win32Forth is huge and has limited documentation.

So, see my Win32Forth Guide on Github.

References

- https://github.com/BillRagsdale/Forth_Projects

- https://github.com/BillRagsdale/WIN32Forth-Guide