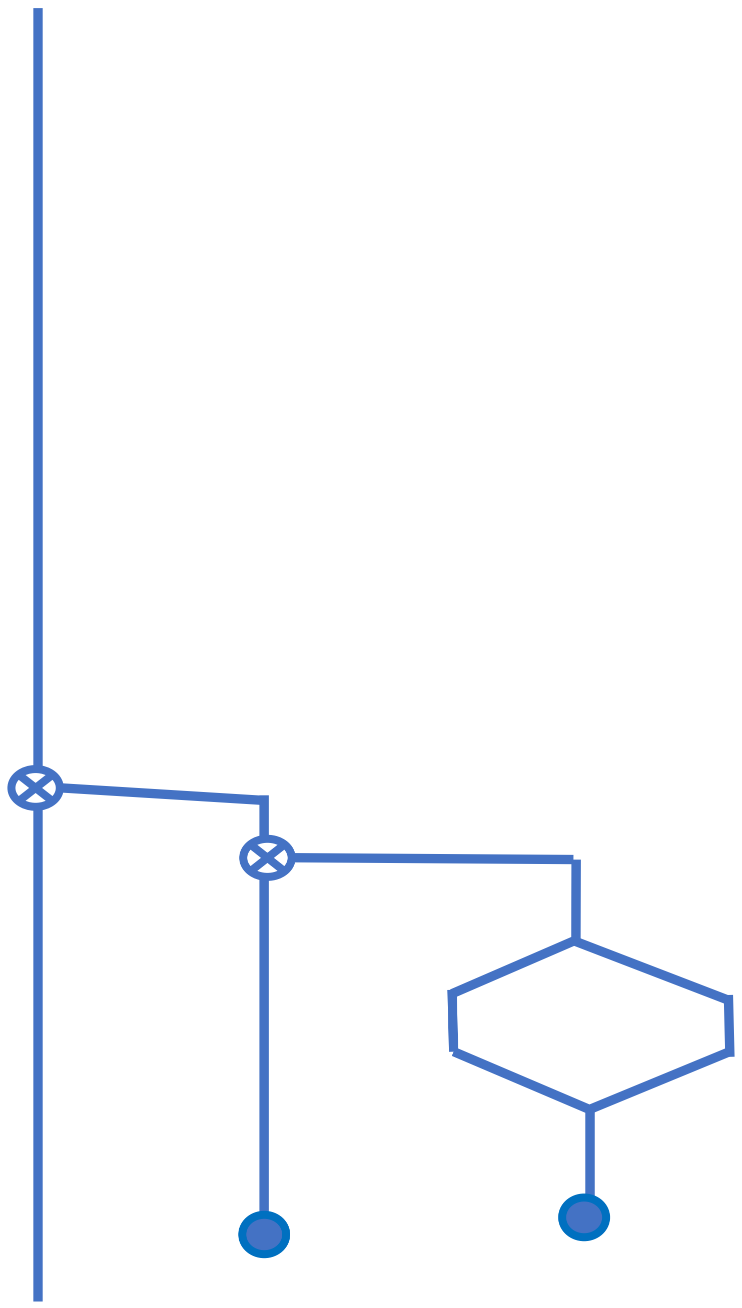


Learning Forth by Programming a Game

SVFIG
Mar. 27, 2021
Bill Ragsdale

Today . . .

We’ll teach Forth programming in a practical situation. A game.

Today . . .

We’ll teach Forth programming in a practical situation. A game.

The style will be just as you would experience in real life.

Today . . .

We’ll teach Forth programming in a practical situation. A game.

The style will be just as you would experience in real life.

Small steps. Lots of testing. Sometimes ripping code apart for a fresh approach.

The Basics

I assume you are familiar with the basic Forth words.

```
constant   variable   value
create     :     ;     exit   allot
dup  drop  swap  over  rot
 +    -    *    /     mod
0=   and   or   not
if else then do loop leave begin again
while until
```

If not, review *Starting Forth* by Leo Brodie, on-line.

And . . .

For the Forth newcomers, or a newcomer to any computer language, reading well written code is a great way to learn syntax and programming style.

And . . .

I’ll formally apply a four level process many already use. But they often do it mentally and may skip some steps.

And . . .

I’ll formally apply a four level process many already use. But they often do it mentally and may skip some steps.

Old timers already follow this process but may find the specific methods interesting.

Style I

These days available memory is huge and computers are blindly fast.

So . . .

Style I

These days available memory is huge and computers are blindly fast.

So . . .

Use long names.   And . . .

Style I

These days available memory is huge and computers are blindly fast.

So . . .

Use long names.   And . . .

Factor into many small words. So . . .

Style I

These days available memory is huge and computers are blindly fast.

So . . .

Use long names.   And . . .

Factor into many small words. So . . .

You gain clarity and testability.

Our Game Words

FULL-GAME	PLAYER-INPUT	PLACE-SYMBOL
EMPTY?	RANGE?	ASCII>##
CURRENT-PLAYER	START	UNPLAYED
3NUMBERS	.SQUARE	3NUMBERS
DASHES	O!	X!
O	X	CLEARGAME
3-CR	.GAME	SQUARE@
SQUARE!	ACTION	

The Process

Discovery: What have we just learned and how does it lead us closer to completion?

The Process

Discovery: What have we just learned and how does it lead us closer to completion?

Design: State the next steps in words . . . "Pseudo Code". (stack diagram)

The Process

Discovery: What have we just learned and how does it lead us closer to completion?

Design: State the next steps in words . . . "Pseudo Code". (stack diagram)

Code: Write Forth code.

The Process

Discovery: What have we just learned and how does it lead us closer to completion?

Design: State the next steps in words . . . "Pseudo Code". (stack diagram)

Code: Write Forth code.

Test: Test the fresh code.

Discovery

Let’s program the game.

Design

On a 9 square board, enter X or O to play the game Tic-Tac-Toe or Naughts and Crosses.

```
 X | O | O          1 | 2 | 3
-----------        ---------
 O | X | X    or    4 | 5 | 6
-----------        ---------
 X | O | X          7 | 8 | 9
```

Discovery

The project is divided into:

Information storage

Data access methods

Formatting and error checks

Playing the game.

And . . .

This game development will appear to be straightforward and logical. Not so in real life.

And . . .

This game development will appear to be straightforward and logical. Not so in real life.

I redesigned and recoded this game three times for the result we will see.

Discovery

We need storage for the game play.

Design

Create storage named ‘**action**’ for a 9 square game.

Discovery

We need storage for the game play.

Design

Create storage named ‘**action**’ for a 9 square game.

Code

```
9 CONSTANT #squares  \ the game size.

CREATE action  #squares cells allot
```

Code

```
CREATE action #squares cells allot
```

Test

```
action #squares cells dump
```

```
4498F4 | 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00
449904 | 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00
449914 | 00 00 00 00
```

Discovery

We can use another creation way to help testing.

Design

Create named storage for a 9 square game.

Preload numbers to assist testing.

Discovery

We can use another creation way to help testing.

Design

Create named storage for a 9 square game.
Preload numbers to assist testing.

Code

```
CREATE action 1 , 2 , 3 , 4 , 5 , 6 ,
              7 , 8 , 9 ,
```

Code

```
CREATE action 1 , 2 , 3 , 4 , 5 , 6 ,
              7 , 8 , 9 ,
```

Test

```
action #squares cells dump
```

```
4498F4 | 01 00 00 00 02 00 00 00  03 00 00 00 04 00 00 00
449904 | 05 00 00 00 06 00 00 00  07 00 00 00 08 00 00 00
449914 | 09 00 00 00
```

Style II

In the interest of brevity I am omitting most stack diagrams and summary comments from each definition.

Style II

In the interest of brevity I am omitting most stack diagrams and summary comments from each definition.

It is essential you always include them. You may come back to your code years later. Here is the proper format:

```
: insert-cr    ( n --- )
\ insert a cr every third value of n
      3 mod    0=  if cr then ;
```

Discovery

We need words to write and read symbols to and from ‘action’.

Design

Access ‘action’ squares 1 . . 9 offsetting to 0 . . 8.

Discovery

We need words to write and read symbols to and from ‘action’.

Design

Access ‘action’ squares 1 . . 9 offsetting to 0 . . 8.

```
: square!     ( square # --- )
\ write a symbol into a square.
   action rot  1- cells+ ! ;

: square@     ( square --- # )
\ read a symbol from a square.
  action swap  1- cells+ @ ;
```

Discovery

We need words to write and read symbols to and from ‘action’.

Design

Access ‘action’ squares 1 . . 9 offsetting to 0 . . 8.

```
: square!  action rot  1- cells+ ! ;

: square@  action swap 1- cells+ @ ;
```

```
4 77 square! 5 88 square! 6 99 square!
4 square@ . 5 square@ . 6 square@.
And see  77  88  99  ok
```

Style III

There are a number of conventions followed in Forth naming.

- Adding @ means fetching from memory.

  ```
  square@
  ```

Style III

There are a number of conventions followed in Forth naming.

- Adding @ means fetching from memory.
  `square@`
- Adding ! means soring in memory.
  `square!`

Style III

There are a number of conventions followed in Forth naming.

- Adding @ means fetching from memory.

  `square@`

- Adding ! means soring in memory.

  `square!`

- Adding . (dot) means outputting.

  `.game`

Style III

There are a number of conventions followed in Forth naming.

- Adding @ means fetching from memory.
  `square@`
- Adding ! means soring in memory.
  `square!`
- Adding . (dot) means outputting.
  `.game`
- Using > and < means to and from. And so on.

Discovery

The array ‘action’ is addressed 0 to 8 while the board squares are numbered 1 to 9. This 1- adjustment in made in square@ and square!.

By limiting access to ‘action’ by only two words this adjustment needs only to be made in one place.

```
: square!  action rot  1- cells+ ! ;

: square@  action swap 1- cells+ @ ;
```

Discovery

We need to see the board contents.

Design

Over the 9 cells of ‘action’ display the contents.

Code

```
: .game #squares 1+ 1
   do i square@ .  loop ;
```

```
Test     .game
    1   2   3   77   88   99   7   8   9   ok
```

Discovery

But this doesn’t look like the real game.

Design

For squares evenly divisible by 3 insert ‘cr’.

Code

```
: 3-cr 3 mod 0= if cr then ;
: .game    #squares 1+ 1 do
    i 3-cr i square@ .  loop ;
```

Discovery

But this doesn’t look like the real game.

Design

For squares evenly divisible by 3 insert ‘cr’.

Code

```
: 3-cr 3 mod 0= if cr then ;
: .game    #squares 1+ 1 do
    i 3-cr i square@ .  loop ;
```

Test

```
.game
 1   2   3
77  88  99
 7   8   9      ok
```

Discovery

Early testing is done. We need to clear the board.

Design

Over squares 1 to #squares place a zero in each square.

Code

```
: ClearGame
 #squares 1+ 1 do i 0 square! loop ;
```

Discovery

Early testing is done. We need to clear the board.

Design

Over squares 1 to #squares place a zero in each square.

Code

```
: ClearGame
 #squares 1+ 1 do i 0 square! loop ;
```

Test

```
ClearGame  .game  and see:
  0 0 0
  0 0 0
  0 0 0   ok
```

Discovery

Notice the loop range is 1 to #squares+1.

Design

LOOP terminates BEFORE the loop limit. So we have to make the limits 1 to 10 to execute over squares numbered 1 to 9.   Just add 1.

Code

```
: ClearGame
 #squares 1+ 1 do i 0 square! loop ;
```

Discovery

We need to place symbols in the squares

Design

Declare the numeric values ‘X’ and ‘O’.

Use >square to write these values into cells.

```
( square ---  )   \ write symbol into square.
```

Discovery

We need to place symbols in the squares

Design

Declare the numeric values ‘X’ and ‘O’.

Use >square to write these values into cells.

(square ---)   \ write symbol into square.

Code

```
1 CONSTANT X    2 CONSTANT O
: X!   X square! ;   \ place X
: O!   O square! ;   \ place O
```

Code

```
1 CONSTANT X    2 CONSTANT O
: X!   X square! ;   \ place X
: O!   O square! ;   \ place O
```

Test

```
ClearGame
1 X!   2 X!   3 X!   7 O!   8 O!   9 O!
.game     and see:

    1  1  1
    0  0  0
    2  2  2    ok
```

Discovery

We now are able put values into the game squares. We need to show them as X’s and O’s to play.

Design

If the stored number is zero print that square number.
If the stored number is one print “X”.
If the stored number is two print “O”.
This calls for a CASE statement.

+

Discovery

We now are able put values into the game squares. We need to show them as X’s and O’s to play.

Design

If the stored number is zero print that square number.
If the stored number is one print “X”.
If the stored number is two print “O”.
This calls for a CASE statement.

Note: the input cell number is duplicated and then incremented at the end as these words will chain together.

Code

```
: .square     dup square@
  case 0 of   dup .      endof
       1 of   ." X "     endof
       2 of   ." O "     endof
    endcase 1+ ;
```

Test

```
 1 .square   4 .square    7 .square
```

and see.

```
X   4   O     as action holds   1   1   1
                                0   0   0
                                2   2   2
```

Discovery

We would like to see a more realistic display.

Design

Between squares show a “|”.

Between rows show “-----------”.

Discovery

We would like to see a more realistic display.

Design

Between squares show a “|”.

Between rows show “-----------”.

```
Code  ( AS A PROTOTYPE, to get spacing.)
: 3numbers cr ."  1 | 2 | 3 " ;
: dashes    cr ."  -----------" ;
: .game   ( --- )
    cr 3numbers dashes
       3numbers dashes 3numbers ;
```

Code

```
: 3numbers cr ."  1 | 2 | 3 " ;
: dashes   cr ."  --------- " ;
: .game
     cr 3numbers dashes
        3numbers dashes 3numbers ;
```

Test

```
.game

    1 | 2 | 3
    ---------
    1 | 2 | 3
    ---------
    1 | 2 | 3   ok
```

Discovery

We are ready do show the ‘live’ game display.

Design

Modify ‘3numbers’ to use ‘.square’ to show contents of the squares.

Discovery

We are ready do show the ‘live’ game display.

Design

Modify ‘3numbers’ to use ‘.square’ to show contents of the squares.

Code

```
: 3numbers ( square --- square+1 )
    cr .square ." | “
       .square ." | “  .square ;
```

Code

```
: 3numbers ( square --- square+1 )
   cr .square ." | "
      .square ." | "  .square ;
```

Test

```
    1  3numbers   and see:
    X | X | X

    7  3numbers   and see:
    O | O | O
```

Code

```
: 3numbers  cr   " .square  ." | "
    square  ." | " .square  ;
: .game   cr
   1  3numbers dashes
      3numbers dashes 3numbers drop ;
```

Test

```
.game

   X | X | X
   ---------
   4 | 5 | 6
   ---------
   0 | 0 | 0   ok
```

Discovery

We are close to playing the game. Let’s manually place the markers.

Design

Use X! and O! to place markers.

Discovery

We are close to playing the game. Let’s manually place the markers.

Design

Use X! and O! to place markers.

```
Test
ClearGame
1 X!  2 O!  3 X!  4 O!  5 X!  6> O!
7 X!  8 O!  9 X!   .game to see
  X | O | X
  ---------
  0 | X | O
  ---------
  X | O | X
```

Discovery

We will setup some of the controls for game play.

Design

We need the number of UNPLAYED games.

UNPLAYED is odd for X and even for O.

Start a game with UNPLAYED   at   #squares (9)

Discovery

We will setup some of the controls for game play.

Design

We need the number of UNPLAYED games.
 UNPLAYED is odd for X and even for O.
Start a game with UNPLAYED   at   #squares (9)

Code

```
0 VALUE unplayed
: current-player  unplayed 1 and   ;
: start
   ClearGame  #squares to unplayed ;
```

Note: As a VALUE, UNPLAYED gives its stored value.

Discovery

Until now, tested words communicate with one another so no error checking has been used. But input from players can be rather complex and uncertain.

Discovery

Until now, tested words communicate with one another so no error checking has been used. But input from players can be rather complex and uncertain.

We'll add range checking, test for input errors, and early exits.

Discovery

Until now, tested words communicate with one another so no error checking has been used. But input from players can be rather complex and uncertain.

We'll add range checking, test for input errors, and early exits.

For clarity and ease of testing, break these functions into words as the usual Forth style.

Discovery

The user inputs ASCII key while we need decimal numbers.

Design

Do the math on a KEY input by subtracting the ASCII value of ‘0’ (48) .
’49 ASCII#>’ would give a decimal 1.

Discovery

The user inputs ASCII key while we need decimal numbers.

Design

Do the math on a KEY input by subtracting the ASCII value of ‘0’ (48) .
’49 ASCII#>’ would give a decimal 1.

Code

```
: ASCII>#  ascii 0 - ; ( char --- n )
```

Test

```
(key) 53  ASCII>#  .   And see 5
5  ok
```

Discovery

More housekeeping. We need a range check for valid user input and a test for an empty square.

Design

Give a true flag for user input within 1 to 9.

Give a true flag if a square is zero.

Code

```
: range?  dup 1 <  swap 9 > or 0= ;

: empty?   square>  0=   ;
```

Discovery

Play alternates between X and O.

Design

Depending on ‘current-player’ place an X or O in the specified square: odd or even. Then decrement the VALUE ‘unplayed’.

Discovery

Play alternates between X and O.

Design

Depending on ‘current-player’ place an X or O in the specified square: odd or even. Then decrement the VALUE ‘unplayed’.

Code

```
: place-symbol ( square --- )
   current-player if X! else O! then
   -1 +TO unplayed   ;
: ps place-symbol ; \ give a short name
```

Code

```
: place-symbol
   current-player if X! else O! then
   -1 +TO unplayed   ;
: ps place-symbol ; \ give a short name
```

Test

```
start 1 ps 3 ps 4 ps 6 ps 7 ps 9 ps
.game

  X | 2 | O
  ---------
  X | 5 | O
  ---------
  X | 8 | O    ok
```

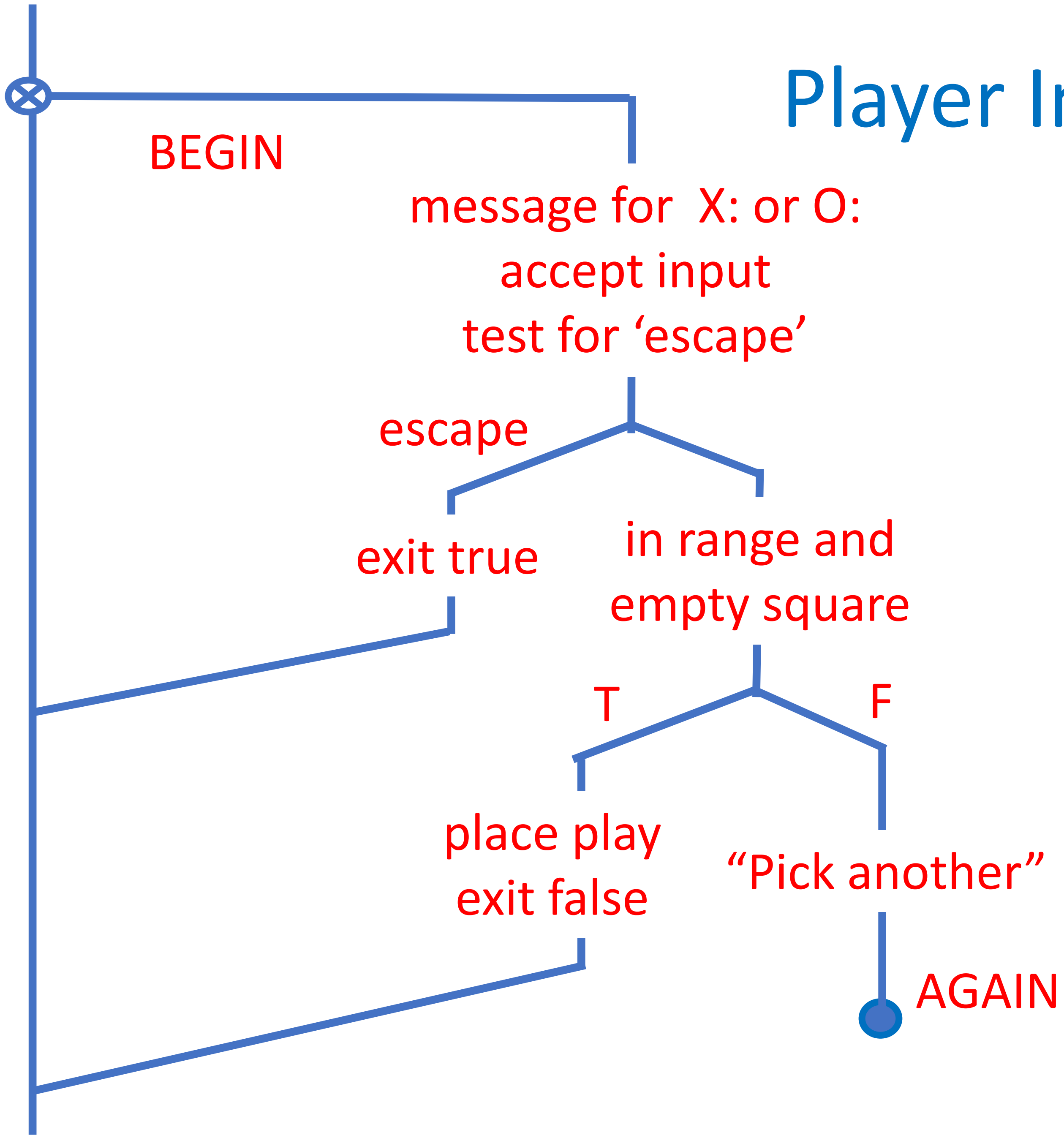
Discovery

How do we process one play?

Design

BEGIN, for one play X or O.
Instruct the player and accept a keystroke.
If it is ‘escape’, notify and exit true.
Otherwise, convert to a square number.
If in range and if the square is empty then place the corresponding marker on the board.
Decrement the ‘unplayed’ value, exit false.
Otherwise, remind the player to input a square number and repeat. AGAIN

Player Input



Player Input

Code

```
: player-input
  BEGIN  cr ." Square number for “
   current-player if ." X:  “
                 else ." O:  " then
   key dup emit   dup  27 ( esc ) =
   if  drop ." Exiting“  true  exit then
   ASCII># dup  range?  over empty? and
   if place-symbol .board  false  exit then
   ( otherwise )
       drop ." Pick another square.  “
   AGAIN  ;
```

Test

```
start   player-input   and see:
Square number for X:  2

  1 | X | 3
 -----------
  4 | 5 | 6
 -----------
  7 | 8 | 9       ok
```

Full Game Play

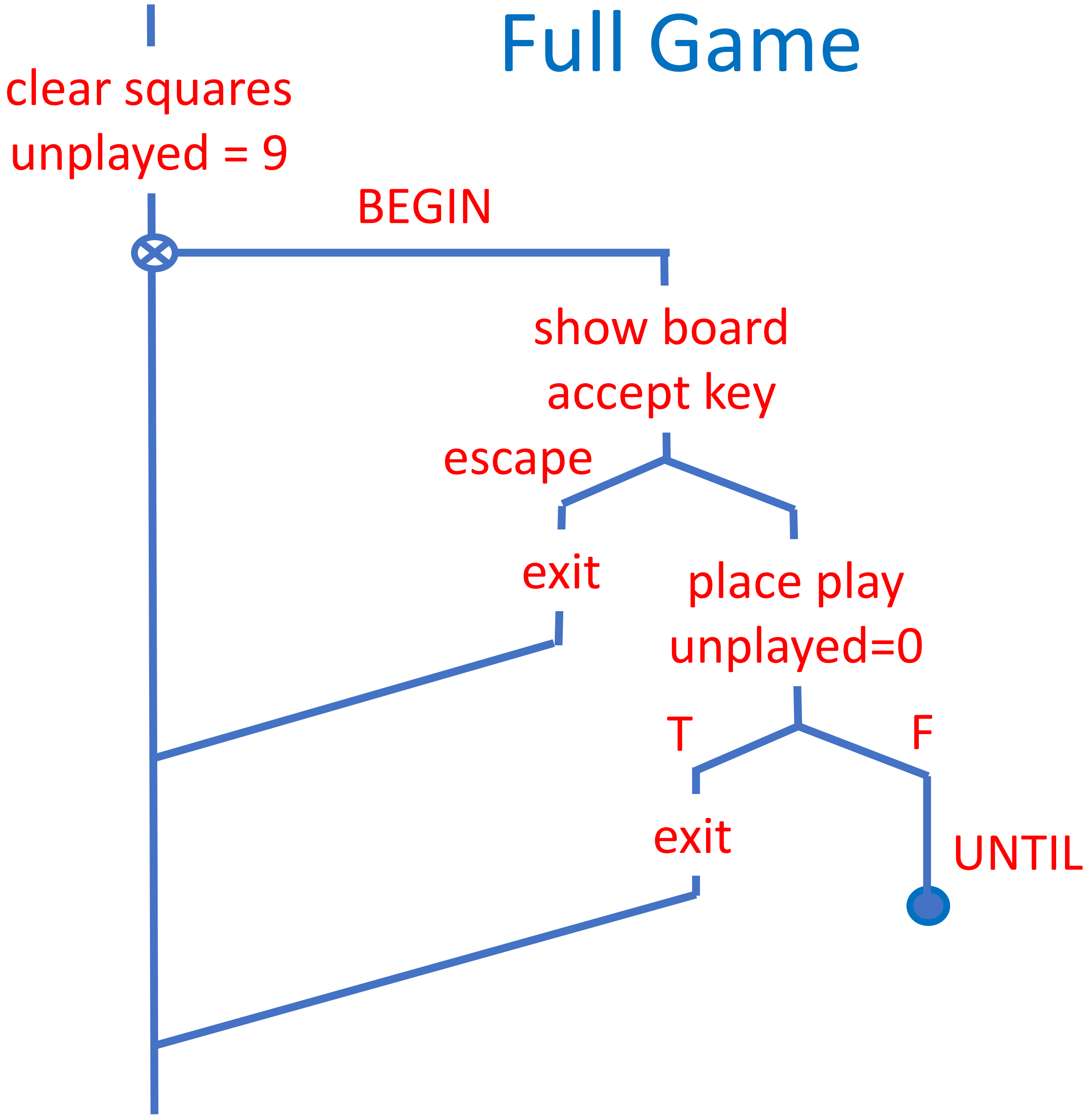
Discovery

We now have the final FULL-GAME without scoring. Scoring will be added in Session Two.

Design

```
Clear ‘action’ and set ‘uplayed’  squares to 9.
Remind the user how to exit early.
BEGIN Display the board.
          Accept the player input and store.
          If input was ‘escape' exit the game early.
          Repeat (for next play).
   UNTIL all nine plays have been made.
```

Full Game


clear squares
unplayed = 9
BEGIN
show board
accept key
escape
exit
place play
unplayed=0
T
F
exit
UNTIL

Code

Full Game Play, unscored

```
: full-game
 start  cr ." Enter 'esc' to exit. “
 BEGIN .game player-input
     if exit then
     unplayed
   0= UNTIL ;
```

Code

Full Game Play, unscored

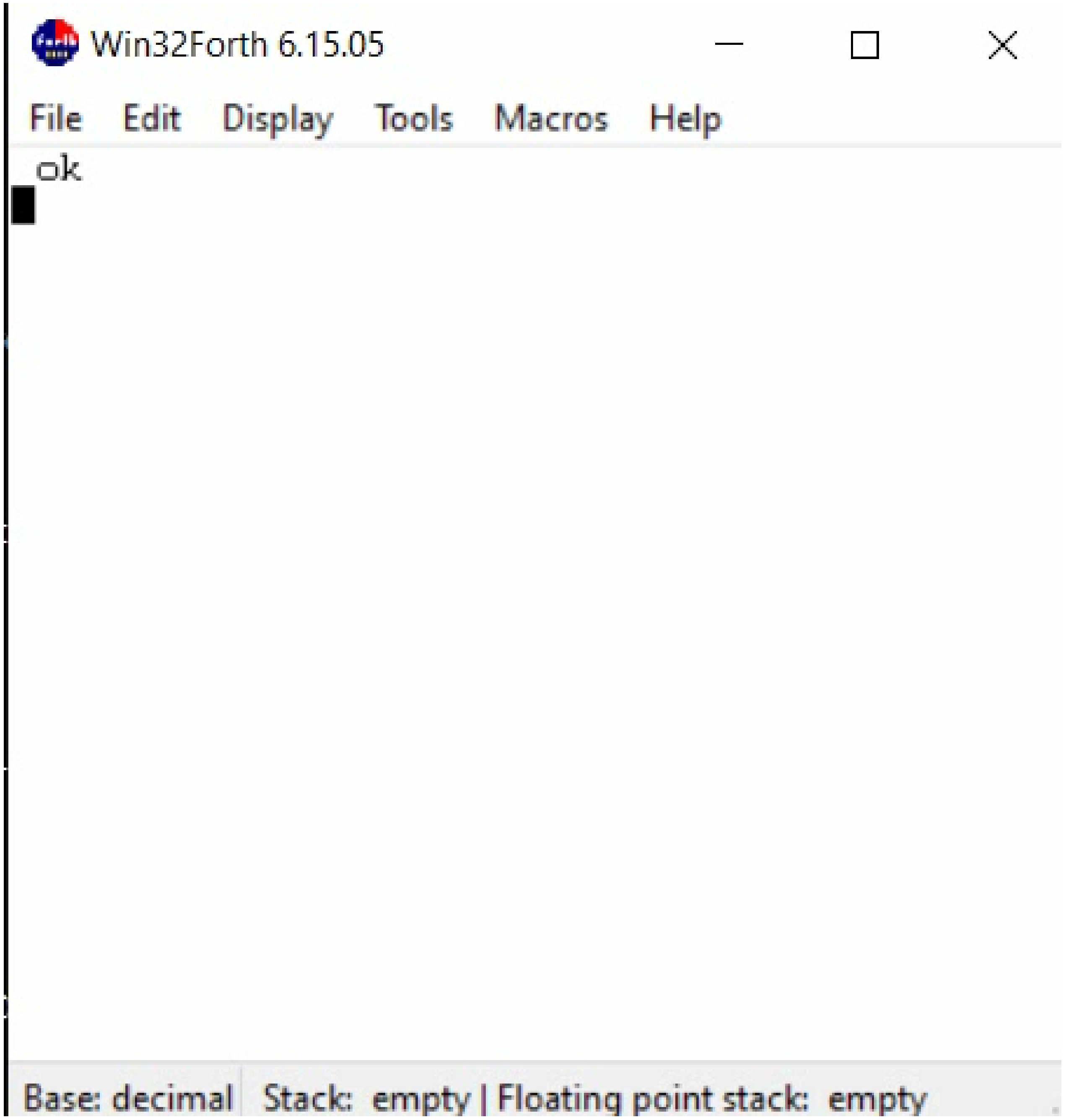
```
: full-game
 start  cr ." Enter 'esc' to exit. “
 BEGIN .game player-input
      if exit then
      unplayed
    0= UNTIL ;
```

Test

```
full-game square number for X:  9

  X | O | 3
 ------------
  4 | X | 6
 ------------
  O | 8 | X
```

Now we see actual play.



```
 ok
```

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

The Future

We have the basis of a language for board games. Change #squares and a bit more for 8x8 checkers and chess.

```
64 CONSTANT #squares  .game

0  0  0  0  0  0  0  0
0  0  0  0  0  0  0  0
0  0  0  0  0  0  0  0
0  0  0  0  0  0  0  0
0  0  0  0  0  0  0  0
0  0  0  0  0  0  0  0
0  0  0  0  0  0  0  0
0  0  0  0  0  0  0  0   ok
```

The Future

Change square@ and square! to use byte storage rather than cell storage.

```
: square!  action rot  1-  c! ;

: square@  action swap 1-  c@ ;
```

This flexibility is one of the features of Forth.

Summary

We could have written ‘full-game’ as one huge, integral program. Often done in other languages.

Summary

We could have written ‘full-game’ as one huge, integral program. Often done in other languages.

By breaking into many small words we have created the beginning of a game language.

And facilitated testing.

Summary

We could have written ‘full-game’ as one huge, integral program. Often done in other languages.

By breaking into many small words we have created the beginning of a game language.

And facilitated testing.

This program was ripped up and re-written three times bouncing between top-level and primitive words. My guidance: Just jump in.

References

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

Questions?

Discovery

IN BLUE

Design

IN GREEN

Code

IN RED

Test

.action

IN BLACK