Tiny PILOT

for tiny 6502 computers
by Nicholas Vrtis

Presented in MICRO
Sept. 1979 - issue 16, page 41

I: L: and P: statement additions by Bob Applegate
MICRO, August, 1980

Co-PILOT Matching Extension by Robert Schultz
MICRO, October, 1980

Source code hand-entered; this program guide;
X: statement, [and ← commands; L: and P: modifications;
Commodore 16 port with color text and sound addition
by Dave Hassler, 2023 & 2025

*** STARTING TINY PILOT ***

1. Load the interpreter from disk with *LOAD "TINYPILOT-C16",8,1* (or *,1,1* for tape).*
2. Once loaded, start the interpreter by typing *SYS 4352* . There is 10.5 KB of user RAM.

* If using the "Co-PILOT" modified version, load *TINYPILOTC16(XM)* instead.
For more information about the Co-PILOT version see the `M:` statement discussion, below.

*** EDITOR ***

Command	Function
@	Start execution of the PILOT program in memory
↑	Move edit pointer to start of program
£	Display next line of the program (list)
%	Pad to end of line with delete characters
[	Clear user RAM and variables
←	Exit to C16 monitor
b/s	Backspace to correct typing errors (current line)
c/r	Carriage return to indicate end of statement line (maximum 125 characters per line)

***** STATEMENTS *****

`A:` ACCEPT - Input up to 40 characters into ANSWER/CHRS field.

`?:` Accept NAME - Input up to 40 characters into the NAME and ANSWER/CHRS fields simultaneously (see `X:` statement).

`C:` COMPUTE - Performs arithmetic on variables named A through Z. Also may place a variable's value into the ANSWER area as ASCII, so it can be matched against a string (`C:$=x`). NOTE: allowed operations are: +, -, and =. Range limit is +/- 999. (We *told* you it was tiny!)

`E:` EXIT FROM SUBROUTINE - Return to address saved by prior USE statement.

`I:`***x*** INPUT - Input a positive number 0-9999 into variable *x*, which can be A-Z. (Sticking to the range 0 to 999 for C: calculations is advised)

`J:`***x*** JUMP - Jump to label *x* for next line.

`L:`***x*** LEAP TO ML ROUTINE - Calls machine language routine at address stored in special ML-calling variable *x*, which can be A - C.

`M:` MATCH - Compare text to last input from terminal (in CHRS), set match flag to `Y` if equal, `N` if not equal. Case-sensitive.

`P:`***x*** PSEUDO-RANDOM NUMBER - Puts a pseudo-random number 0-99 into variable ***x,*** which can be A-Z.

`R:` REMARK - Program remarks, not executed.

`S:` STOP - Stop program and return to editor. **Must** be at end of a program/script; otherwise, the interpreter crashes.

`T:` TYPE - Display text on the terminal. `$`***x*** will display the contents of a variable (-999 to 999). Text may be uppercase letters, numbers, symbols (other than those used by the editor), and PETSCII graphics characters. This statement is also used to change text color and produce sound.

`U:`***x*** USE SUBROUTINE - Saves the address of the start of the next line and then jumps to label *x*.

`X:` TRANSFER STRING - Moves the NAME string ('`?`') to a temporary storage location called STRING; moves whatever was in STRING into NAME *and* CHRS (the `A:` "answer" area).

*** CONDITIONALS, LABELS, VARIABLES, COLOR, SOUND ***

Labels and Variables do not use the same set of letters A-Z. Each has its own set of A-Z to work with. This means the programmer has 26 variables to use, and 25 labels. E.g., `*HC:H=H+4` is a legal construction: `*H` is a label for a jump-to point (or start of a subroutine), and `H` in the equation is a variable.

`N,Y` NO and YES Conditionals may precede any statement. E.g., `YJ:` means jump only if the MATCH flag is `Y`. `NT:` means print following text only if MATCH flag is `N`.

`*`*x* Labels may precede any statement or conditional. *x* = single letter B-Z. Acts as a destination for JUMP or USE. NOTE: The `A` and `*` labels are reserved for system use.

`$`*x* Variable marker. With `T:`, causes variable contents to be displayed, or with `C:`, prepares them to be matched. `$?` indicates the NAME field. `C:$=`*x* puts a variable into the ANSWER area as an ASCII string so it can be matched with `M:`.

`&`*n* Text color and control marker for `T:`. *n* may be from 0-9. If *n* is 1, the screen will be cleared. Colors available:

2 White 3 Red 4 Cyan 5 Purple 6 Green 7 Blue 8 Yellow

Control switches:

1 Clear screen 9 Reverse on 0 Reverse off

`]`*n* Sound generation marker for `T:`. *n* may from 1-4. The sounds available are:

1 Bell 2 Blip 3 Bong 4 Buzz

x Variables are a single letter A-Z. While the numerical content of a variable may be from -9999 to 9999, and `I:` will allow input of 0-9999, `T:` will only print out in the range -999 to 999.

E.g., if `C=2000` and the user enters `945` for `I:B`, `C:A=C-B` will leave 1055 in A, but only print out 55.

*** EXAMPLES ***

EDITOR COMMANDS

All commands are followed by pressing the *RETURN* key on the C16.

@ Start execution of the PILOT program in memory.

↑ Move edit pointer to start of program memory.

£ Display next line of the program/script. Similar to BASIC's LIST command. To use, move the edit pointer to the start of the program with the **↑** command, then type the **£** key followed by *RETURN*. Do this for each line of the program. When only the **£** appears (after the final `S:`), the end of the script has been reached. When editing a line, use the **£** command to list the script to *just before* the line you wish to re-type. Then, type the new line (see **%** below).

% Pad to the end of a program line with an non-printing character, namely CHR$(128). Used to "fill in" an edited line, and can only be used when the new line is shorter than the original line. In the script:

```
T:THIS LINE IS 34 CHARACTERS LONG.   (including the T:)
C:A=B+42
```

the first line could be replaced with:

```
T:THIS LINE HAS 30 CHARACTERS.%
```

but not

```
T:THIS LINE HAS MANY MORE CHARACTERS THAN THE FIRST.%
```

as it would wipe out the line `C:A=B+42` below it.

[Clears user RAM and variables, but not the special machine code variables. Needed so that a new program will not get merged with one already in memory, likely creating a useless and unexecutable script.

← Exit to C16 monitor. The *STOP* key is not scanned in Tiny PILOT. This command is how one gets out of the editor/Tiny PILOT. From the monitor, programs and scripts can be saved and loaded to/from disk or tape.

DEL The *DEL* key on the C16 is used to backspace and correct typing errors *only* on the line currently being entered.

LANGUAGE STATEMENTS

All statement lines in a program/script are followed by pressing the *RETURN* key on the C16.

A: `A:`*ENVELOPE*
The user input "ENVELOPE" can be compared with the M: statement. Input is stored in the ANSWER/CHRS area *backward* from $2A to $03. 40 characters input is the maximum.

?: `?:`*PENELOPE*
A string variable that may be printed with `$?`. Called NAME, this appears to be a holdover from PILOT-73, with educational programs in mind. It is stored in the NAME area, 40 characters, max. It is also stored in the ANSWER/CHRS area, simultaneously, so the script can match against user input with the `M:` statement. It is, essentially, the *one* string variable available, but can only be assigned by user input, not from within a program. And, it is also stored backward, as with ANSWER. Can (*should?*) be used with the `X:` statement.

C: `C:X=Z+Z+Z+Z` this is $X = 4 \times Z$
`C:X=5-2+6-12` this will leave *X* with the value -3
`C:X=A` this will leave *X* with value of *A*
It's very similar to BASIC, although operation is strict left-to-right; parentheses not allowed. `$` is a special case:

```
C:$=X
M:12
YJ:Z
  ...
```

Here, whatever is in *X* gets put in the ANSWER area as ASCII and is matched with the string '12', i.e., ASCII characters 49 and 50. If the two strings match, the Y flag is set, and the next line will see that and jump to module Z; if not, the program will continue on after `YJ:Z`.
*** Does not work for string variable `?` (NAME); for that, see the `X:` statement below.

E: Simply place at the end of a subroutine. The equivalent of BASIC's 'RETURN'.

```
*ZC:A=B+C
E:
```

Performs the calculation, then returns to the line right after the U:Z statement that called the subroutine.

I:*x* Prompted with a `?`, user input can be a positive integer 0-9999, and it is assigned to variable *x*. Although, due to limitations with `T:` and `C:`, staying within the range 0-999 is advised.

J:*x* To jump to another labeled section of the program/script, replace the *x* with the label latter. Letters B through Z are available for labels. `J:A` means jump to the previous `A:` (ACCEPT) so the user can re-enter string input. `J:*` means restart the program from the beginning.

`L:x` The variables referenced are *not* the numeric variables! Addresses *must* be pre-set in locations $10A0 to $10A5 (A-C, separate variables just for this routine) prior to running a Tiny PILOT program or script. This can be done by using the ← command from the editor to jump to the C16 monitor. From there, one can enter the address(es) of preloaded ML routines; e.g., enter *32* and *03* into addresses $10A0 and $10A1, respectively, to set ML variable A to point to $0332 (#818), which is the C16's cassette input buffer. To return to Tiny PILOT, type *G 1100* in the monitor. Your custom ML can be called with the statement `L:A`. Make sure the called ML routine ends with an *RTS* so it can return to the Tiny PILOT interpreter, right after the `L:A`. statement.

`M:` Example of use: `M:QUIT,HALT,STO` Explanation of this statement addresses its use in standard Tiny PILOT and with the "Co-PILOT Matching Extension" by Robert Schultz. The main difference between the two is that standard Tiny PILOT's `M:` only matches the *first word* of an `A:` answer from a user. However, the Co-PILOT extension allows for a match with *any word* in the `A:` response.

Standard Tiny PILOT - The example above will look at the last input to the ANSWER/CHRS area (`A: X:` or `?:`) and try to match the *first word only* with the parameters. If *QUIT* or *HALT* or *STOP* was entered by the user, the flag will be set to `Y`. However, if *STOMACH* or *QUITE* is the first word, they will *also* match. Spaces in the M: list will remove that issue (see Co-PILOT example below). Further, the following will set the flag to `N`:

```
T:WHAT'S YOUR FAVORITE COLOR?
A:I LIKE BLUE
M:BLUE
YT:I AGREE THAT BLUE IS BEST.
```

However, a response of *BLUE IS BEST.* would have resulted in a match. One (albeit complicated) way out of this is to use conditionals and cascade `M:` statements:

```
T:WHAT'S YOUR FAVORITE COLOR?
A:I LIKE BLUE
M:BLUE
NM:I LIKE BLUE
YT:I AGREE THAT BLUE IS BEST.
```

Tiny PILOT with Co-PILOT – In the first example above, the Y flag would be set, as the word *BLUE* is in the user's answer string. However, *BLUEISH-GREEN* will also match, but *GREENISH-BLUE* will not. One way to avoid a misinterpretation of the answer is to try to anticipate close-but-incorrect responses:

```
T:WHAT PRIMARY COLOR DO YOU LIKE BEST?
A:
M:BLUE-GREEN,BLUEISH-GREEN,YELLOW-GREEN,YELLOWISH-GREEN,GREENISH
YT:THAT IS NOT A PRIMARY COLOR. TRY AGAIN.
YJ:A
M:GREEN
YT:YES, GREEN IS BEST.
NT:THAT ONE'S NOT MY FAVORITE.
S:
```

Here, with the first `M:` statement, the programmer has anticipated possible responses like *YELLOWISH-GREEN* or *GREENISH-BLUE*. If GREEN isn't involved at all, the second `M:` will fail to a more generic response and end. Of course, there's only so much one can do...

Matching variable content is done with `M:$x` and `C:$=x` . The former will match the ASCII representation of the variable with the contents of the ANSWER/CHRS area. The latter places the value of a variable into the ANSWER/CHRS area. The following looping examples illustrate 1) matching against a fixed value, 2) matching against a second variable.

```
c:a=1                OUTPUT
*xt:$a               1
c:a=a+1              2
c:$=a                3
m:6                  4
nj:x                 5
s:

c:a=6                OUTPUT
c:b=1                1
*xt:$b               2
c:b=b+1              3
c:$=b                4
m:$a                 5
nj:x
s:
```

NOTE: Some programs that test variables and/or numbers with `M:` will fail under the Co-PILOT enhanced version of Tiny PILOT; the LESS-GREATER demonstration program that came with Tiny PILOT is an example. See the Notes section of this manual for information and advice about using the best version of Tiny PILOT for a given application.

`P:`*x* Will place a pseudo-random decimal number from 0-99 in variable *x*.

`R:` A BASIC 'REM' statement. However, don't use any of the editing command symbols in your remarks, as they will get misinterpreted. The same restriction applies to `?:` and `T:` .

`S:` This marks the end of the program and an exit back to the editor. Without it, the interpreter *will* crash. When `S:` is reached execution stops and the program pointer is reset to the first line.

`T:` Will print any string <126 characters long. Quotes around the string are not needed. To print out the contents of NAME (`?`) or a variable, preface it with `$`. E.g.,

```
T:YOU HAVE $X "MARBLES" LEFT, $?.
```

will print out YOU HAVE 12 "MARBLES" LEFT, STAN. assuming X=12 and ?=STAN. To make text or PETSCII graphics colorful, use the `&` marker in the text string. E.g.,

```
T:YOU HAVE &3$X "MARBLES" LEFT, &4$?.&8
T:BYE!
```

will print the same message (assuming the same variables), *x* MARBLES LEFT, will be red and STAN. will be blue. The text color is changed back to the default yellow before printing

BYE! on the next line. The color code `&8` could also have been placed on the next line, before BYE!.

To add one of four sounds to a script, use the `]` marker in the text string. E.g.,

```
T:]1YOU HAVE &3$X "MARBLES" LEFT, &4$?.&8
T:BYE!
```

Will add a “bell” sound before the message is printed.

`U:`*`x`* Saves a return address to the next line and then jumps to the label `*`*`x`*.

`X:` This transfers the contents of the `?:` variable (NAME) into the STRING area, and places whatever was in STRING into both NAME and ANSWER/CHRS. Using `X:` allows for an "additional" string variable that can be altered by the user and also be tested against anything specified for the `M:` statement. E.g.,

```
?:DAVE          (user input to NAME and CHRS area)
A:BOB           (user input, changes the CHRS)
M:BOB
YT:IT'S BOB
X:              (place current NAME into STRING)
X:              (then STRING back to NAME and CHRS area)
M:DAVE
YT:IT'S DAVE
S:
```

Will result in 'IT'S BOB' and 'IT'S DAVE' being printed on the screen. The two `X:` statements place NAME into STRING, then STRING into NAME *and* ANSWER/CHRS, the latter of which `M:` draws from for tests. The two `X:` statements are necessary because when the user enters *BOB* at the `A:` prompt, it also fills the ANSWER/CHRS area with *BOB*, wiping out the *DAVE* that was there upon the first input.

See the included TELLMEDUDE.PIL program for another example of printing use with `X:` .

==========

NOTES, COMMENTS, and ERRATA

First and foremost, PILOT is a blast, and super-easy for kids to learn all the fundamentals of programming: assignment, branching, loops, modular programming, comparison, calculation, etc. This subset of PILOT is appropriate for any kid (or kid at heart?) from about 6- or 7-years old on up. And, as I will mention again later, it is a superb storytelling and dialog engine – heck, that’s what it was made for in the first place.

That said, Tiny PILOT was not originally written for the C16, nor have I brought it “the modern conveniences”; this thing was created in 1979 for the SYM-1 6502 trainer computer built in 1975!

Some considerations: It is pretty easy to break Tiny PILOT; there is absolutely no error checking. Stuff like not assigning a variable a value and then trying to print it, or forgetting to put a ':' after a program statement letter will break it. The *INST/DEL* key works. The *CTRL* key is active, as is the keyboard upper/lowercase switch, but the *STOP* key is not: if Tiny PILOT crashes, it’s usually catastrophic. You may be able to reset the C16, then *SYS4352*, though. There is no "immediate mode"

as in most other classic high-level languages. The editor is *very* simple. You cannot insert a line of code in between lines. The screen editor is active, but won't do anything other than mess up your screen, and possibly your program. Just don't use the cursor keys. Frankly, I would not use the editor for anything other than sketching out ideas or very small programs. Better to just use Mousepad or something for program writing (see ideas below). You have the uppercase letters, numbers and symbols, and PETSCII graphics characters for input and output.

When returning from the C16 monitor, reenter Tiny PILOT at its start location of $1100. You can even reload Tiny PILOT and start the interpreter at $1100, and the program in user RAM will still be there unless you purposefully zap it with the [command. You can go do stuff in the monitor (but *not* BASIC), and as long as your program from $1700 (start of user program/script RAM) to its end was undisturbed, it'll run.

Oh – I use the words "program" and "script" interchangeably with PILOT. Many of the best computer-aided-learning softwares from back in the day look more like movie scripts than computer programs.

LOADING AND SAVING SCRIPTS

Loading and saving programs and scripts is is fairly easy.

To save a script, pop out to the C16 monitor with the ← command and start browsing user RAM with the monitor's M command, starting at $1700; e.g., *M1700* then *RETURN*. Keep tapping an *M*, then *RETURN*, to see more memory. Or, you can make an "educated guess" as to where to start looking. If the script is 3.5 KB in size, you could start looking for the end at about $2500. What you're looking for is the final `S:` (hex 53 3A) followed by a whole bunch of zeros or $FFs. (You *did* zap user RAM and variables before programming, didn't you?) That's the end. Note the address of the last zero after the colon, then add 1. That's your "save to" address. Insert your disk or tape.

Again from the monitor, type *S"FILENAME",8,1700,xxxx*, where "*filename*" is whatever you want (up to 16 characters), *8* can be *1* for tape saves, and "*xxxx*" is the end address you got from the method above. From there, it works just as you'd expect on any Commodore 8-bit machine. Return to Tiny PILOT with *G1100* from the monitor.

To load a script, clear/zap the user RAM, set up your storage device with the disk or tape with the program in question, and then head to the monitor with the ← command in the editor. Type *L"FILENAME",8* (or *,1* for tape) and wait for the magic to happen. Return to Tiny PILOT with the start address: *G1100*.

But as far as typing in a script goes, it's *so much easier* to write a program/script "offline" on a modern PC, to take advantage of the editing capabilities. Write your script with Mousepad, Notepad++, or whatever, save it as a text file, *but with line endings set to CR only*. This is required. Then, fire up an emulator that allows you to copy and paste text (i.e., VICE) and paste in the text to the emulator. Also, VICE likes text files to be pasted into its emulators in lowercase. Save to a .D64 or a .TAP file from the emulated C16 monitor, then move that to an SD card for your actual, physical C16 (or Plus/4) with its SD2IEC (or whatever) interface. Bob's yer uncle.

Speaking of "Bob", I tweaked Bob Applegate's `L:` statement so Tiny PILOT could be burned into an EPROM on the KIM-1, and/or clones and replicas these days. Bob's original approach was to used self-modifying code in a brute-force way that would have made my then-14-year-old self say "*Wow!*" Considering that Bob was just 17 when he wrote and submitted this to MICRO in 1980 makes it all that much more fun.

With the addition of my `X:` statement, I had to limit Bob's ML calling routine to just three instances, variables A-C, but that should be plenty.

This brings up the topic of where to put ML routines, if desired. The cassette buffer from $0332-$03F2 is dandy, assuming you aren't using a Datasette. There's the RS-232 buffer area at $03F7-$0436, seeing as the C16 does not have a UART, or even a User Port, for that matter. Small, 24-byte chunks exist at the end of both color ($0BE8-$0BFF) and screen ($0FE8-$0FFF) RAM, too. With the "standard" version of Tiny PILOT, you have 72 bytes from $16B8 to $16FF, as well.

Zero Page is fairly well sewn up, but you do have a little room (way more than on the VIC-20 or C64, it must be said!): $26-$2A (BASIC's multiplication work area), $D0-$D7 (speech synthesizer RAM), and $E5-$E8 (user application RAM not used by Tiny PILOT).

As for the pseudo-random number routine, it's Jim Butterfield's from page 172 in "The First Book of KIM," which Bob gratefully acknowledged in his original article for MICRO and then tweaked for decimal output. I added a seeding routine at startup to get a little better randomness right out of the gate. It's okay, good for simple games, but the results seem to be weighted toward the ends of the range. On 2,400 dice rolls, I got:

1=424 (17.7%) 2=389 (16.2%) 3=383 (16.0%) 4=394 (16.4%) 5=400 (16.7%) 6=410 (17.1%)

I think Bob's `I:` statement for entering numerical variables is genius. It really amplifies the utility and possibilities of Tiny PILOT. Damn good code, too.

I was very happy to learn that Bob knew that his additions to Tiny PILOT and the language itself were still kicking before he died of cancer – he mentioned that getting his additions and the article published in 1980 convinced his mother to let him leave high school early and start in on an engineering degree at college. Way cool. I hope he's pounding brass at 40 wpm somewhere…

Now let's talk about the `M:` statement. In the "standard" flavor of Tiny PILOT, the `M:` statement only works on the first word input and gives up without checking the rest of the string; and the only way to operate on a string, other than printing the NAME or STRING variables, is a simple `M:` via the `X:` statement. This has both strengths and weaknesses.

Strength: for more "math-oriented" programs, this is a good thing, because you're only going to test the exact ASCII in the ANSWER area for a match. Especially when doing loops, this is important.

Weakness: it stinks for language-oriented input from a user, as illustrated in the `M:` examples above, and makes it harder to guide the student through a lesson, or an adventurer through a dungeon or forest. With that in mind, I have provided a version of Tiny PILOT that addresses the needs of the dialog-oriented among us, namely TINYPILOTC16(XM), which features an augmented `M:` statement crafted by Robert Schultz and documented in the October 1980 issue of MICRO magazine (which he called Co-PILOT). It will give a positive match to the following:

```
T:WHAT'S YOUR FAVORITE COLOR?
A:I LIKE BLUE
M:BLUE
YT:I AGREE THAT BLUE IS BEST.
```

`Y` will be set here, but it would also get set with an answer of *I GOT THE BLUES*. If the match condition is found anywhere in the 40-character-max ANS/CHRS field at the start or after a space, you'll have a match. See the Examples section for a technique to solve part of this problem. Feel free to use the Co-PILOT version of Tiny PILOT if it meets your needs. It's what I use for my smaller text-based adventure games. For dialog engine stuff, the Co-PILOT is a great way to go.

The four sounds inherent in Tiny PILOT for C16 were inspired by the ORIC-1's BASIC. It has keywords like *ZAP* and *EXPLODE* and *DING*, that work just like *PRINT* or *LET*. I had to learn about

it to translate an ancient A.J. Shepherd adventure game, but that's another story. Four sounds "sounded" reasonable to me to imbed – hey, it's supposed to be tiny anyway, right? :^) I'm especially pleased with the "bell" sound, and I was, indeed, thinking Control-G all the way. The sounds are quite loud (except for BLIP), so adjust your TV or monitor accordingly.

Tiny PILOT -- or regular, full-sized PILOT for that matter -- is no mathematical champ. Tiny PILOT is limited to only addition and subtraction; although, it's easy enough to do multiplication and division with those, plus use a loop to test for <= and > when values are between 0 and 999. See the demonstration programs DIVISION and LESS-GREATER on the disk image, or consult the relevant script included in the package (neither program will not work with the Co-PILOT-enhanced version – stick with standard Tiny PILOT for calculation-oriented stuff, or most things, actually). Still, not much to work with, regarding computation. Consider it a challenge! I wrote a full baseball-playing dice game in Tiny PILOT (ARUBA BB DICE, provided in the ZIP file from whence this document came) and it plays very well.

As for this program and PILOT, itself, this version by Nicholas Vrtis appears to be based upon PILOT-73, which was the defacto standard on minicomputers in the '70s. I think it's a pared down version of '73. One interesting note is that Tiny PILOT shares a lot of code with Larry Kheriaty's ultra-tiny WADUZITDO in the September 1978 issue of BYTE -- right down to using the same labels in the source. Both programs must have been made from PILOT-73.

In the article from MICRO #16, Nicholas goes into some detail describing how the program works, section by section. It's a good read and very informative. I recommend it, in addition to his well-commented source code (which I typed in by hand off a printout ... the complete '80s experience!). I know *I* certainly learned a lot about how to build a high-level language interpreter.

PILOT *excels* as an interactive dialog engine. PILOT stands for Programmed Inquiry, Learning, Or Teaching, and was written by Dr. John Starkweather in 1968 at UCSF. It was designed to be a vehicle for computer-aided instruction (CAI) in classrooms, and it's wonderful for that. Super-easy for a teacher or parent to use to create an interactive script that can guide a student through a lesson or exercise. At least, back then. Or, perhaps, today, as a retro adventure game? Maybe something like the "Choose Your Own Adventure" books from the '80s? Who knows?

However you use it ... Have Fun! -- Dave Hassler, 6.Feb.25