0 Stack

00 To know a bit more about the Forth language, let's launch *gforth* in a terminal.

Here we go:

```
gforth ↩
Gforth 0.7.2, Copyright (C) 1995-2008
Free Software Foundation, Inc.
Gforth comes with ABSOLUTELY NO WARRANTY;
for details type `license'
Type `bye' to exit
```

01 Forth uses a *Stack* as a way to pass parameters to, and get results from operations. Entering numbers will put these numbers on the Stack. Enter two numbers.

```
42 ↩ok
17 ↩ok
4807 ↩ok
```

02 You can print the number that is currently at the top of the Stack, using the *dot* symbol: `.`

```
. ↩ 4807 ok
. ↩ 17 ok
. ↩ 42 ok
```

03 Printing the number removes it from the Stack. Put two numbers on the Stack again. You can enter several numbers on the same line.

```
47 150 ↩ok
```

04 You can add the two numbers at the top of the Stack by entering the `+` sign.

Let's try:

```
+ ↩ok
. ↩ 197 ok
```

05 Just like numbers, the + and . symbols should be separated by space. Try not separating them and see what you get. (We don't need to know about the trace information for now).

```
42 17 +. ↩
:8: Undefined word
42 17 >>>+.<<<
Backtrace:
$103D82A08 throw
$103D98C90 no.extensions
$103D82CC8 interpreter-notfound1
```

06 Other arithmetic operations are available as well. They all use the Stack as a container for integer values. Try them.

```
47 150 - .  ↩ -103 ok
47 150 * .  ↩ 7050 ok
150 47 / . ↩ 3 ok
4807 47 / .  ↩ 102 ok
```

07 Enter two numbers on the Stack, add them, then multiply the result by another number.

OK.

```
4807 3 + 42 * . ↩ 202020 ok
```

08 To get the remainder of a division, use MOD.

```
4807 42 MOD . ↩ 19 ok
```

09 You can also obtain both quotient and remainder, using the word /MOD. The quotient will be at the top of the Stack, and the remainder will be just below the top.

```
4807 7 /MOD ↩ok
. ↩ 686  ok
. ↩ 5  ok
```

0A Enter a number, then print its opposite, using NEGATE.

```
17 NEGATE . ↩ -17 ok
```

0B Print 32% of 4807 (approximately).

```
4807 32 * 100 / . ↩ 1538 ok
```

0C Give a more precise result (still using integer values). You can print the integer part and the fractional part separately.

```
480700 32 * 100 / 100 /MOD . . ↩ 1538 24
```

0D Enter two numbers, then print the greatest number, using MAX.

```
42 17 MAX . ↩ 42 ok
```

0E Enter two numbers, then print the smallest number, using MIN.

```
42 17 MIN . ↩ 17 ok
```

0F Enter four numbers on the Stack, and print the greatest number.

```
42 17 255 -13 MAX MAX MAX . ↩ 255 ok
```

→ Forth programs are made of *numbers* and *words*, separated by space.

→ Numbers are pushed on a data *Stack*, words are interpreted and executed on the fly.

→ Numbers enter and leave the Stack in a *First In, First Out* fashion.

→ . , + , etc. are words, as well as MOD or MAX.

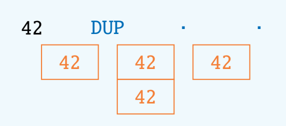
→ Words *consume their arguments*, removing them from the Stack.

→ Forth uses *Reverse Polish Notation* for all arithmetic expressions; the order of operations determines the order of evaluation (no need for parentheses).

1 Arrange

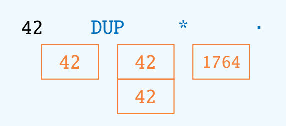
10 Numbers are usually removed from the Stack by the words that use them. If you want to keep a number on the Stack *and* use it as a parameter for a word, you can duplicate it, with the word DUP.
Draw the successive states of the Stack to better understand the way it works.

```
42 DUP ↩ ok
. . ↩ 42 42  ok
```



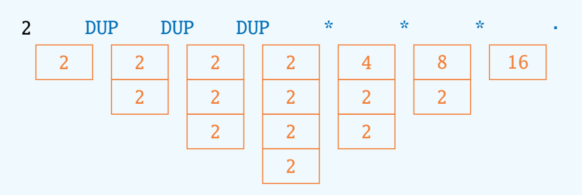
11 Find a sequence of words that computes the square of a number without typing the number twice. Try your sequence on several values.

```
42 DUP * . ↩ 1764 ok
-7 DUP * . ↩ 49 ok
```

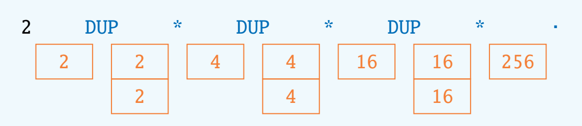


12 Ask *gforth* to compute the value of 2^4 and 2^8.

```
2 DUP DUP DUP * * * . ↩ 16 ok
```

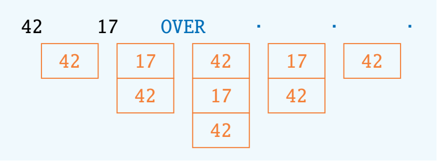


```
2 DUP * DUP * DUP * . ↩ 256 ok
```



13 When you want a copy of the number *just below* the top instead of a copy of the top, use OVER. Try it.

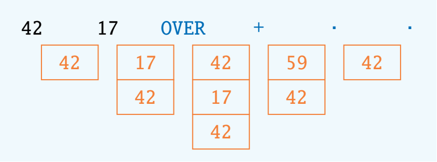
```
42 17 OVER . . . ↩ 42 17 42 ok
```



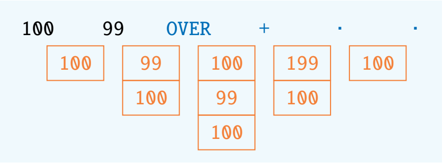
---

14 Find a sequence of words that given two numbers a and b, leaves the Stack with the numbers a, $a + b$. (Remember that repeating . will print the stack content in *reverse order*).

```
42 17 OVER + . . ↩ 59 42 ok
```



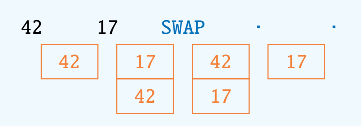
```
100 99 OVER + . . ↩ 199 100 ok
```



---

15 Sometimes you need to exchange the top of the Stack with the number just below. That's when you use the word SWAP.

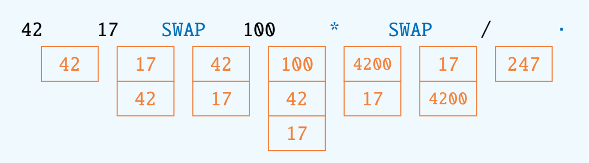
```
42 17 SWAP . . ↩ 42 17 ok
```



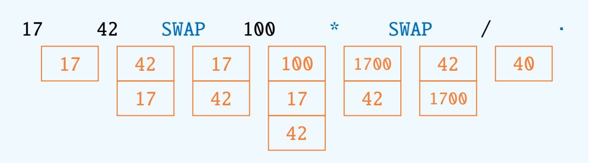
---

16 Find a sequence of words that given two numbers a and b, will compute (approximately) $\frac{100a}{b}$.

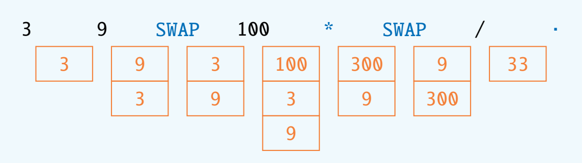
```
42 17 SWAP 100 * SWAP / . ↩ 247 ok
```



```
17 42 SWAP 100 * SWAP / . ↩ 40 ok
```

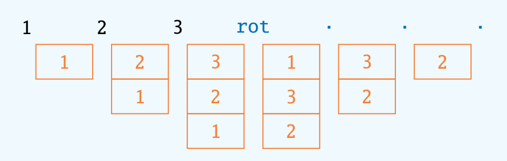


```
3 9 SWAP 100 * SWAP / . ↩ 33 ok
```

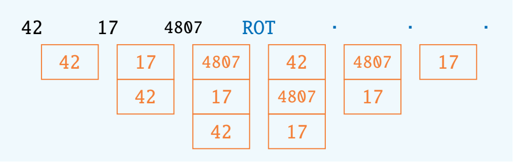


17 You can also move the value that is *in the third position* to the top of the Stack using ROT.

```
1 2 3 ROT . . . ↩ 1 3 2 ok
```

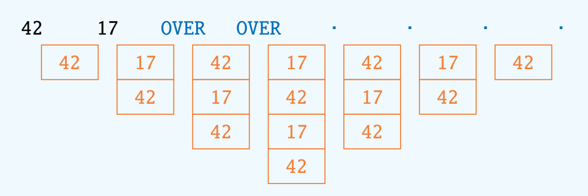


```
42 17 4807 ROT . . . ↩ 42 4807 17 ok
```



18 Using OVER twice duplicates the two numbers at the top of the Stack[1].

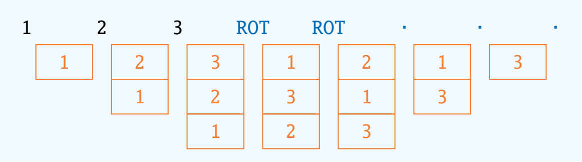
```
42 17 OVER OVER ↩ok
. . . . ↩ 17 42 17 42 ok
```



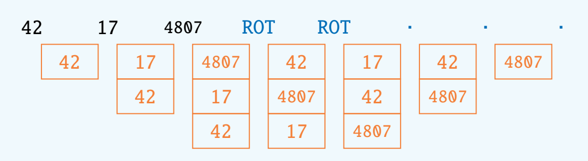
19 Using ROT twice rotates the number at the top of the Stack *under* the number just below the top[2].

Ok

```
1 2 3 ROT ROT . . . ↩ 2 1 3 ok
```



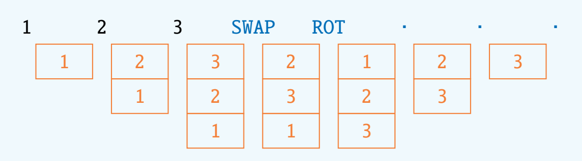
```
42 17 4807 ROT ROT . . . ↩ 17 42 4807 ok
```



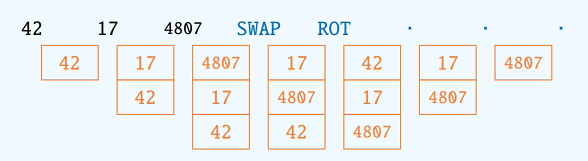
1A Enter three numbers, then print them *in the order they were entered.*

Easy:

```
1 2 3 SWAP ROT . . . ↩ 1 2 3 ok
```



```
42 17 4807 SWAP ROT . . . ↩ 42 17 4807
```



[1] The word 2DUP is a faster equivalent of OVER OVER.

[2] The word -ROT is a faster equivalent of ROT ROT.

1B Enter two numbers, then print them in ascending order. Test your sequence by entering the numbers in a different order.

```
42 17 OVER OVER MAX ROT ROT MIN ↩ok
. . ↩ 17 42 ok
```

42	17	OVER	OVER	MAX	ROT	ROT	MIN
42	17	42	17	42	42	17	17
	42	17	42	17	42	42	42
		42	17	42	17	42	
			42				

```
17 42 OVER OVER MAX ROT ROT MIN ↩ok
. . ↩ 17 42 ok
```

17	42	OVER	OVER	MAX	ROT	ROT	MIN
17	42	17	42	42	17	42	17
	17	42	17	42	42	17	42
		17	42	17	42	42	
			17				

1C In your last test, replace the sequences `OVER OVER` and `ROT ROT` with faster words `2DUP` and `-ROT`.

```
42 17 2DUP MAX -ROT MIN ↩ok
. . ↩ 17 42 ok
```

42	17	2DUP	MAX	-ROT	MIN
42	17	17	42	17	17
	42	42	17	42	42
		17	42	42	
		42			

```
17 42 2DUP MAX -ROT MIN ↩ok
. . ↩ 17 42 ok
```

17	42	2DUP	MAX	-ROT	MIN
17	42	42	42	42	17
	17	17	42	17	42
		42	17	42	
		17			

1D Enter three numbers, then print them in ascending order, using the sequence for sorting two numbers. You can use `(` and `)` to comment on what happens on the Stack[3].

```
42 17 4807 ↩ok
2DUP MAX -ROT MIN ( 42,4807,17 ) ↩ok
ROT               ( 4807,17,42 ) ↩ok
2DUP MAX -ROT MIN ( 4807,42,17 ) ↩ok
-ROT              ( 17,4807,42 ) ↩ok
2DUP MAX -ROT MIN ( 17,4807,42 ) ↩ok
ROT               ( 4807,42,17 ) ↩ok
. . . ↩ 17 42 4807 ok
```

42	17	4807	2DUP	MAX	-ROT	MIN	ROT	2DUP	MAX	-ROT	MIN	-ROT	2DUP	MAX	-ROT	MIN	ROT
42	17	4807	4807	4807	4807	17	42	42	42	42	17	42	42	4807	42	42	17
	42	17	17	4807	17	4807	17	17	42	17	42	4807	4807	42	4807	4807	42
		42	4807	17	4807	42	4807	42	17	42	4807	17	42	4807	4807	17	4807
			17	42	42			17	4807	4807			4807	17	17		
			42					4807					17				

There must be less tedious ways to do this!

[3] Be careful: `(` is a Forth word, i.e. it must be separated from the first word of comment by a space.

1E When you don't need a number on the Stack any more, you can get rid of it with DROP.

```
42 17 DROP . ↩ 42 ok
```

1F The word 2DROP is a faster way to execute DROP DROP in order to eliminate 2 values from the Stack.

```
42 17 23 2DROP . ↩ 42 ok
```

- → Values on the Stack can be duplicated, exchanged or removed.
- → `DUP ( n -- n,n )` : duplicates the top of the Stack.
- → `DROP ( n -- )` : removes a number from the top of the Stack.
- → `OVER ( a,b -- a,b,a )` : copies the value under the top of the Stack.
- → `SWAP ( a,b -- b,a )` : exchanges the top of the Stack with the value below the top.
- → `ROT ( a,b,c -- b,c,a )` : rotates the value in the third position to the top of the Stack.
- → `-ROT ( a,b,c -- c,b,a )` : rotates the top of the Stack to the third position.
- → `2DUP ( a,b -- a,b,a,b )` : duplicates the two values at the top of the Stack.
- → `2DROP ( a,b -- )` : removes two numbers from the top of the Stack.

2 Display

20 We can display characters instead of numbers. The word EMIT consumes the value at the top of the Stack and prints the corresponding character on the terminal.

```
65 EMIT ↩ A ok
```

21 We will use the following ASCII codes to represent the elements of the puzzle. Try them if you want.

wall	: 35	crate	: 36
filled goal	: 42	worker on goal	: 43
goal	: 46	worker	: 64

```
35 EMIT ↩ # ok
36 EMIT ↩ $ ok
42 EMIT ↩ * ok
43 EMIT ↩ + ok
46 EMIT ↩ . ok
64 EMIT ↩ @ ok
```

22 The code for space (or *Blank*) is frequently used, so there is a word for it: BL, and even an equivalent of the sequence BL EMIT, called SPACE.

```
BL . ↩ 32 ok
SPACE SPACE SPACE ↩    ok
```

23 The word CR sends a *Carriage Return* on the terminal, forcing the display to start on a new line.

```
CR CR CR ↩



ok
```

24 One way to put a character on the Stack is to enter its ASCII code, if you know that code. Another way is to use the word CHAR. CHAR reads the following word on the entry as the litteral char you want to have on the Stack. Try it!.
Display the word SOKOBAN on the terminal.

```
CHAR N ↩ok
CHAR A ↩ok
CHAR B ↩ok
CHAR O ↩ok
CHAR K ↩ok
CHAR O ↩ok
CHAR S ↩ok
CR EMIT EMIT EMIT EMIT EMIT EMIT EMIT ↩ok
SOKOBAN ok
```

This takes some work!

25 Here is a faster way to display characters: use the word `."`, and all the following non space characters in the flow of entry will be printed until a `"` is met.
Don't forget that `."` is a word in itself and must be separated from the rest of the entry.

```
." SOKOBAN" ↩ SOKOBAN ok
." Foo Bar" ↩ Foo Bar ok
```

26 The terminal can do other actions than just display characters. For example, the character with code 7 will ring a bell, and 9 will send a tabulation.

```
CR 9 EMIT 9 EMIT 35 EMIT ↩
                        # ok
```

27 Some complex actions on the terminal are initiated by the character with code 27 (ESC), followed by a `[` and a command.
For example, to clear the entire screen, display the escape character followed by the string `[2J`.

```
27 EMIT ." [2J" ↩







            ok
```

28 Another terminal escape command allows you to select the column and row of the terminal where you want to display the next characters. Try the escape command `5;3H` for example.

```
27 EMIT ." [5;3H" 42 EMIT ↩


     * ok
```

29 The *gforth* vocabulary includes special words that use terminal escape commands: `PAGE` will clean the screen; `AT-XY` will take two numbers on the Stack and use them as the *x* and *y* coordinates of the next thing to be displayed.

```
PAGE 2 2 AT-XY 46 EMIT 5 3 AT-XY 42 EMIT ↩



  .
     * ok
```

2A The *gforth* word `ESC[` is doing the same as the sequence: `27 EMIT 91 EMIT` would: it sends these control characters to the terminal.
For example try to print words using underlined (`4m`) mode, and then get back to normal mode (`0m`).

```
ESC[ ." 4m" ." Foo" ESC[ ." 0m" ." Bar" ↩
FooBar ok
```

2B The terminal can also print characters in color. Just print the escape sequence, then the color number, for instance `31`, for red, followed by `m`.
Try to print lines using different colors.

```
ESC[ ." 31mFoo" ↩ Foo ok
CR ESC[ ." 32mFoo" ↩ Foo ok
CR ESC[ ." 34mBar" ↩ Bar ok
```

2C To reset all the terminal display attributes, use the word `ESC[` then print the string `0m`.

```
CR ESC[ ." 0mQux" ↩
Qux ok
```

2D Find a sequence of words that given a color code on the Stack, like 34 for example, changes the terminal color. Your sequence should start with `ESC[`, then print the number, then a `m`.

```
34 ESC[ . CHAR m EMIT ."Foo" ↩ Foo
```

That doesn't work, because in the sequence

```
34 . CHAR m EMIT ↩ 34 m ok
```

the `.` word inserts a space after printing the number.

2E Try with the word `.R` (*dot-R*). This word takes two numbers *n*, *w* and prints *n* aligned on the right on *w* columns. If *w* columns are not enough to print the number, `.R` will display the whole number anyway. The important thing is that it will do it *without adding a trailing space* like `.` does.

```
4807 10 .R ↩       4807 ok
42 2 .R 17 2 .R ↩ 4217 ok
32 0 .R CHAR m EMIT ↩ 32m ok
```

I see.

2F Try again this time using `0 .R` in your sequence.

```
34 ESC[ 0 .R 109 EMIT ." Foo" ↩ Foo ok
35 ESC[ 0 .R CHAR m EMIT ." Foo" ↩ Foo ok
32 ESC[ 0 .R CHAR m EMIT ." Foo" ↩ Foo ok
 0 ESC[ 0 .R CHAR m EMIT ." Foo" ↩ Foo ok
```

→ `EMIT ( c -- )` : displays a character on the terminal.

→ `CHAR {c} ( -- c )` : reads a character on the entry and puts its ASCII code on the Stack.

→ `." {CCCCC"}` : reads a sequence of characters on the entry flow until `"`, then prints the string.

→ `PAGE` : clears the screen.

→ `AT-XY ( x,y -- )` : sets the position *x*,*y* for the next display on the terminal.

→ `ESC[` : starts an escape sequence on the terminal.

→ `.R ( n,w -- )` : prints the number *n* aligned on the right on *w* columns, with no trailing space.

3 Define

30 Forth lets you define your ow words.
Here's how to create a new word:

→ start with : (*colon*), a space, and the name you want to give to your new word,

→ write all the Forth words that this definition should execute,

→ finish the definition with ; (*semicolon*).

Let's try! Define a word called STAR that will display the character with the code 42.

Cool!

```
: STAR 42 EMIT ; ↩ok
STAR ↩ * ok
STAR STAR STAR ↩ *** ok
```

31 Create a definition for a word called SQUARE that takes a number *n* on the top of the Stack and replaces it with n^2.
Then create a word called CUBE that takes a number *n* on the top of the Stack and replaces it with n^3. Use the previous word you just created.
Try your definition with several examples.

Ok!

```
: SQUARE DUP * ; ↩ok
42 SQUARE . ↩ 1764 ok
-7 SQUARE . ↩ 49 ok

: CUBE DUP SQUARE * ; ↩ok
42 CUBE . ↩ 74088 ok
-3 CUBE . ↩ -27 ok
```

32 Create a word named SORT2 that given 2 values on the Stack, sorts them so that the greater value is below the top, and the smaller value is at the top.

```
: SORT2 2DUP MAX -ROT MIN ; ↩ok
42 17 SORT2 . . ↩ 17 42 ok
17 42 SORT2 . . ↩ 17 42 ok
```

33 Create a word named SORT3 that given 3 values on the Stack sorts them so that the greatest value is below the two others on the Stack, and the smallest is at the top.

```
: SORT3 SORT2 ROT SORT2 -ROT SORT2 ROT ; ↩
42 17 4807 SORT3 . . . ↩ 17 42 4807 ok
243 39 -55 SORT3 . . . ↩ -55 39 243 ok
```

34 Create a word named MODE that given a number, sends an escape command to the terminal with that number.
Try your word with different modes.

That is the sequence I defined some time ago:

```
: MODE ESC[ 0 .R CHAR m EMIT ; ↩ok
:23: Undefined word
: MODE ESC[ 0 .R CHAR >>>m<<< EMIT ;
```

Hey! What's happening?

35 Oh. I forgot to mention that CHAR cannot be used *inside* a definition[4]. Use [CHAR] instead.

OK.

```
: MODE ESC[ 0 .R [CHAR] m EMIT ; ↩ok
31 MODE ↩ ok
34 MODE ↩ ok
35 MODE ↩ ok
0 MODE ↩ok
CR 4 MODE ." Foo" 0 MODE ." Bar" ↩
FooBar ok
```

That is better!

36 Create a word BLUE that switches the display color to red, and a word NORMAL that restores all display attributes to normal.

Easy:

```
: BLUE 34 MODE ; ↩ok
: NORMAL 0 MODE ; ↩ok

CR BLUE STAR SPACE NORMAL STAR ↩
** ok
```

37 You can keep your programs in *script files*. When *gforth* is launched with the name of a script file as an argument, the words in the file are automatically executed as *gforth* starts.
Edit a Forth script file called *Sokoban.fs*. Enter your definitions, and execute a simple sequence of actions using these definitions. Note that ending the file script with the word BYE will tell *gforth* to quit right after executing the last word, giving us a stand-alone Forth program. Try it!

That's cool, now I can write a program!

```
: MODE ESC[ 0 .R [CHAR] m EMIT ;

: BLUE 34 MODE ;

: NORMAL 0 MODE ;

BLUE CHAR @ EMIT NORMAL CR BYE
```

Sokoban.fs

```
gforth Sokoban.fs ↩
@
```

It works!

38 Comments can be entered after the word \ or between (and). Stack comments, like in this instance

```
: NIP ( a,b -- b )
  SWAP DROP ;
```

are very usual.

Ok.

```
\ Sokoban.fs  A Game of Sokoban in Forth!!

: MODE ESC[ 0 .R [CHAR] m EMIT ;

: BLUE 34 MODE ;

BLUE CHAR @ ( col,chr -- )
EMIT     ( col -- )
NORMAL   ( -- )
CR BYE
```

[4] Here's the reason: CHAR reads the entry flow, looking for the next word, and then *puts the char value on the Stack*, while [CHAR] reads the entry flow, looking for the next word, and then *compiles the char value in the definition* that is currently going on. CHAR used inside a definition, is inactive. Thus the following item in the entry, m causes an Undefined word error.

39 You should keep your definitions small and elegant. For that purpose, you can always create some helper words. For example:

→ replace `[CHAR] m EMIT` with a word called `.M`

→ replace `0 .R` with a word called `.N`

Note that the *gforth* vocabulary already includes the word `.N`. When executing your script file, *gforth* will simply emit a warning and make the new definition replace the existing one.

Ok.

```
\ Sokoban.fs  A Game of Sokoban in Forth!!

: .M [CHAR] m EMIT ;

: .N 0 .R ;  \ n -- print n w/o trailing space

: MODE ESC[ .N .M ; \ N -- print Esc Nm
```

3A Create new words to display the elements of the game. Here they are:

element	*display*	*mode*
empty space		0
worker	@	34
worker on goal	+	34
walls	#	31
crates	$	32
goal	.	32
filled goal	*	33

```
: RED      31 MODE ;
: GREEN    32 MODE ;
: YELLOW   33 MODE ;
: BLUE     34 MODE ;
: NORMAL   0 MODE ;
: DISPLAY-EMPTY  NORMAL  BL EMIT ;
: DISPLAY-WORKER BLUE   [CHAR] @ EMIT ;
: DISPLAY-ONGOAL BLUE   [CHAR] + EMIT ;
: DISPLAY-CRATE  GREEN  [CHAR] $ EMIT ;
: DISPLAY-WALL   RED    [CHAR] # EMIT ;
: DISPLAY-GOAL   GREEN  [CHAR] . EMIT ;
: DISPLAY-FILLED YELLOW [CHAR] * EMIT ;
\ testing
DISPLAY-WORKER DISPLAY-CRATE
DISPLAY-WALL   DISPLAY-EMPTY
DISPLAY-GOAL   DISPLAY-FILLED
DISPLAY-ONGOAL BYE
```

```
gforth Sokoban.fs ↩
@$# .*+
```

It works!

3B Your program can be made simpler. Do you see all these repeated patterns in the definitions? Instead, we can define one general word: `DISPLAY` that given an ascii code and a color number, will display that character in that color.

Ok.

```
: DISPLAY MODE EMIT ; \ chr,col --
```

3C Then you can change your `DISPLAY-xxx` definitions so that they call this word.

```
: DISPLAY MODE EMIT ; \ chr,col --
: DISPLAY-EMPTY  BL  0 DISPLAY ;
: DISPLAY-WORKER [CHAR] @ 34 DISPLAY ;
: DISPLAY-ONGOAL [CHAR] + 34 DISPLAY ;
: DISPLAY-WALL   [CHAR] # 31 DISPLAY ;
: DISPLAY-CRATE  [CHAR] $ 33 DISPLAY ;
: DISPLAY-GOAL   [CHAR] . 32 DISPLAY ;
: DISPLAY-FILLED [CHAR] * 35 DISPLAY ;
\ testing
DISPLAY-WORKER DISPLAY-CRATE
DISPLAY-WALL   DISPLAY-EMPTY
DISPLAY-GOAL   DISPLAY-FILLED
DISPLAY-ONGOAL BYE
```

```
gforth Sokoban.fs ↩
@$# .*+
```

3D We can simplify the code a bit more. All these DISPLAY-xxx have the same structure. We can define specialized words, and use them by combining them with DISPLAY.
Create words WORKER, ONGOAL, WALL, etc. that will push the right codes on the Stack.

Ok.

```
\ Sokoban.fs  A Game of Sokoban in Forth!!

: .M [CHAR] m EMIT ;

: .N 0 .R ;  \ n -- print n w/o trailing space

: MODE ESC[ .N .M ; \ N -- print Esc Nm

: DISPLAY MODE EMIT ; \ chr,col --

: WORKER [CHAR] @ 34 ;
: ONGOAL [CHAR] + 34 ;
: WALL   [CHAR] # 31 ;
: CRATE  [CHAR] $ 33 ;
: GOAL   [CHAR] . 32 ;
: FILLED [CHAR] * 35 ;
: EMPTY  BL       0  ;

\ testing
WORKER DISPLAY CRATE  DISPLAY
WALL   DISPLAY EMPTY  DISPLAY
GOAL   DISPLAY FILLED DISPLAY
ONGOAL DISPLAY BYE
```

That is much simpler! But having words like RED, BLUE, etc. instead of numbers would be better.

3E Words like RED, BLUE, etc. that just push a number on the Stack can be declared as *constants* rather than colon definitions.
The word CONSTANT takes a number on the Stack, and creates a new definition with the name that follows. Try it with *gforth* .

```
42 CONSTANT ASTERISK ↩
ASTERISK EMIT ↩ * ok
```

I see.

3F Since Forth uses space as a delimiter in order to separate words in the entry, you can use any other symbol to define your words. For instance 34YP! makes a valid Forth word, albeit not a very clearly named one.
Define all the constants you need in the Sokoban. Place them at the beginning of the program.

```
\ Sokoban.fs  A Game of Sokoban in Forth!!
31 CONSTANT RED       32 CONSTANT GREEN
33 CONSTANT YELLOW    34 CONSTANT BLUE
0 CONSTANT NORMAL

: M 109 EMIT ;
: .N 0 .R ; \ n -- print n w/o trailing space
: MODE ESC[ .N M ;     \ N -- print Esc Nm
: DISPLAY    \ c,m --  display c  c in mode m
  MODE EMIT ;

: WORKER [CHAR] @ BLUE ;
: ONGOAL [CHAR] + BLUE ;
: WALL   [CHAR] # RED ;
: CRATE  [CHAR] $ GREEN ;
: GOAL   [CHAR] . GREEN ;
: FILLED [CHAR] * YELLOW ;
: EMPTY  BL       NORMAL ;

\ testing
WORKER DISPLAY CRATE  DISPLAY
WALL   DISPLAY EMPTY  DISPLAY
GOAL   DISPLAY FILLED DISPLAY
ONGOAL DISPLAY BYE
```

→ Forth lets you create new words that can be used just like existing words.

→ `: ( {XXX ... ;} )` : creates a new definition named *XXX* for the following sequence.

→ At run time, a word created with `:` will execute the words contained in its definition.

→ `;` : ends a colon definition.

→ You can *redefine* words simply by writing their new definition with `:` and `;`.

→ `[CHAR] {X} )` : inside a colon definition, reads the next character on the entry and compiles its ASCII code in the definition.

→ `CONSTANT ( {XXX} n -- )` : defines a constant named *XXX* for the value *n*.

→ At run time, a word created with `CONSTANT` will put its value on the Stack.

→ Invoking *gforth* with a script name executes all the words in the script file.

→ `BYE` : leaves *gforth* .

→ Create and combine together simple specialized words to avoid big repetitive ones.