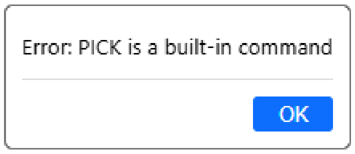
Procedures

A procedure is a set of code that has a name.

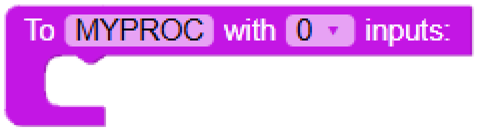
Note that you cannot name a procedure with the same name as a command in the Logo language. Of course, you don't know what those names are, but Logo Blocks will tell you if you can't use that name. If you enter a name with a space it in, Logo Blocks will replace the space with an underscore character.


Error: PICK is a built-in command

To run the code in your procedure, you need to call it by its name. It is very efficient to create procedures, as you will see!

Logo Blocks lets you create procedures that have no inputs or the number of inputs you want.

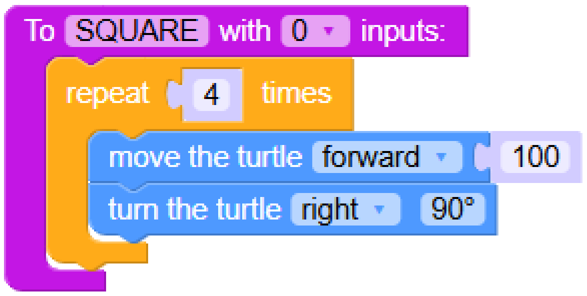
Here are some examples of how to create and use procedures, with and without inputs.

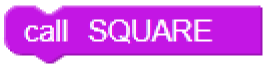


Let's create a simple procedure to draw a square. Drag this MYPROC block to the workspace area.

Change MYPROC to SQUARE so that it has a meaningful name. Keep the number of inputs at 0.

Add blocks to draw a square inside the procedure. You can use the ***repeat*** block, along with ***forward*** and ***turn*** blocks. Your procedure might look like this:


```
To SQUARE with 0 inputs:
  repeat 4 times
    move the turtle forward 100
    turn the turtle right 90°
```

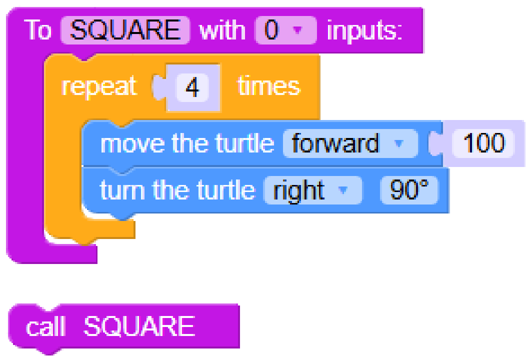

```
call SQUARE
```

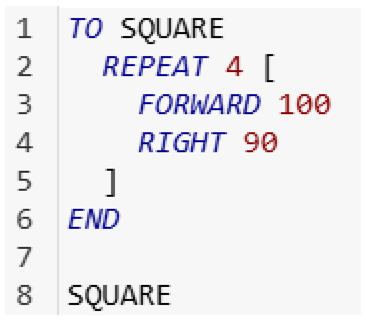
If you click **Run**, nothing happens. Your procedure is now defined, but you haven't told Logo Blocks to use it.

If you look at the Procedures section, you see a new block:
```
call SQUARE
```

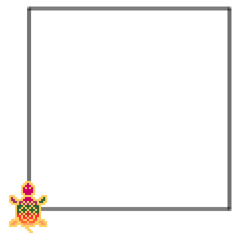
Use this block to tell the computer to run your SQUARE procedure.

Your workspace, Editor panel, and Graphics panel look like this:





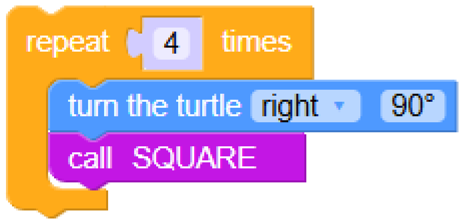
```
TO SQUARE
  REPEAT 4 [
    FORWARD 100
    RIGHT 90
   ]
END

SQUARE
```

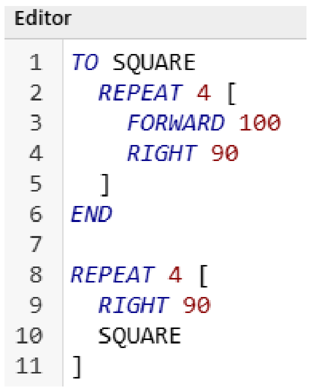


Congratulations! You have just created your first procedure!

How can you use your procedure? Just use its name, SQUARE, whenever you want a square. Don't delete the SQUARE procedure. You need to keep it around to call it again. If you delete it, Logo Blocks will forget that.

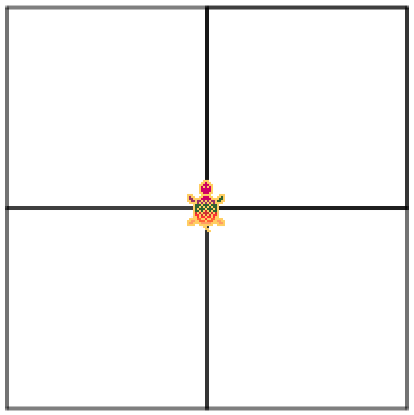
Here is an example of using a procedure inside a *repeat* block.





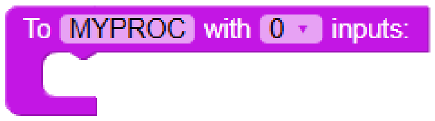
Editor

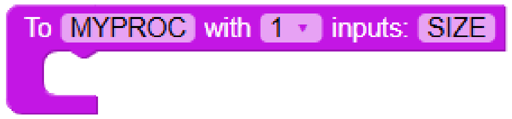
```
TO SQUARE
  REPEAT 4 [
    FORWARD 100
    RIGHT 90
  ]
END

REPEAT 4 [
  RIGHT 90
  SQUARE
]
```



What if you want to have a procedure that will draw any size square you want? To do that, you need to use a *variable*. A *variable* is a placeholder. You give it a name, like SIZE.

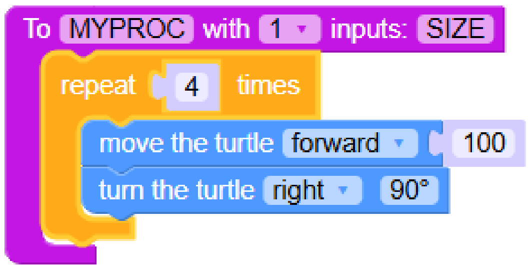
Start with the MYPROC procedure. Edit the procedure to use 1 input named SIZE.





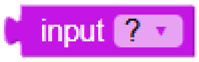
You are creating a procedure that uses a variable. Think of a variable as a container that can hold information, such as a number, word, or list. A variable that is part of the name of a procedure is called a *local variable* because only this procedure can use it. Later you will learn how to create *global variables*, which can be seen and used by any procedure in your program.

Next, add the blocks to draw a square.
Your procedure will now look like this:



```
To MYPROC with 1 inputs: SIZE
  repeat 4 times
    move the turtle forward 100
    turn the turtle right 90°
```

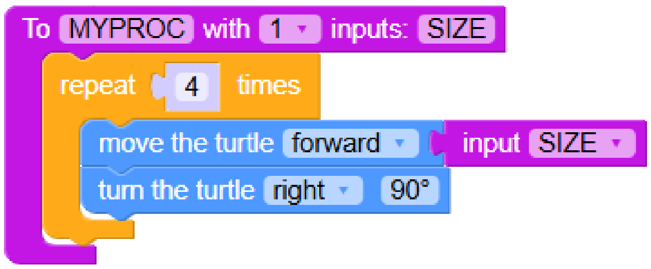
Now we need to tell the procedure to use the variable. In the **Procedures** section is a block for an input.



```
input ?
```

Drag the ***input*** block to replace the 100 in the ***forward*** block. It now looks like this:

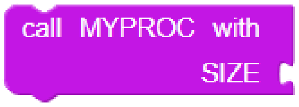
Logo Blocks even knew to call it SIZE!



```
To MYPROC with 1 inputs: SIZE
  repeat 4 times
    move the turtle forward input SIZE
    turn the turtle right 90°
```

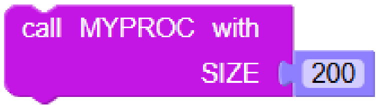
Now, to use this new procedure that has a variable, you need to call the procedure with a number. This gives a value to the variable named SIZE. Your code will now use the number you give the procedure whenever it sees the name SIZE.

In the **Procedures** area, look for this block that you can use to call your procedure.



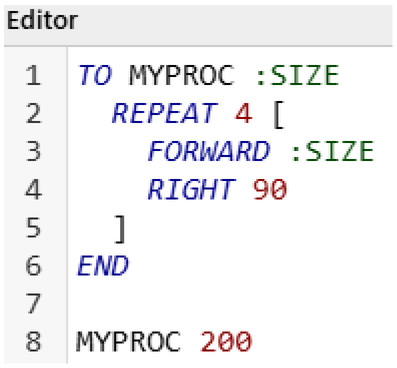
```
call MYPROC with
            SIZE
```

Go to the **Math** section and grab the number block. Drag it next to the variable name SIZE. Change the number to the size you want. For example, try 200.



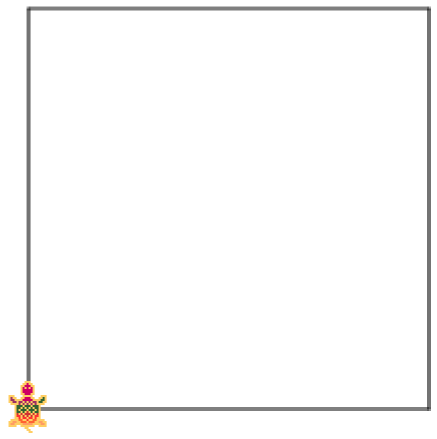
```
call MYPROC with
            SIZE 200
```

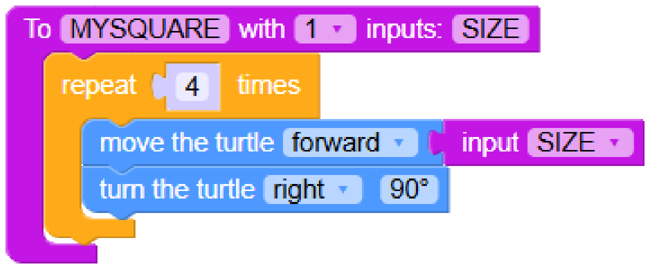
Now click **Run**. You will see a size 200 square. Also look at the code in the Editor to see how Logo created the code for the procedure. It puts "dots," or a colon, before a variable name.

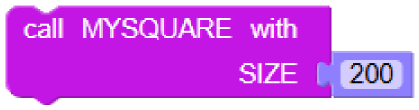


Editor

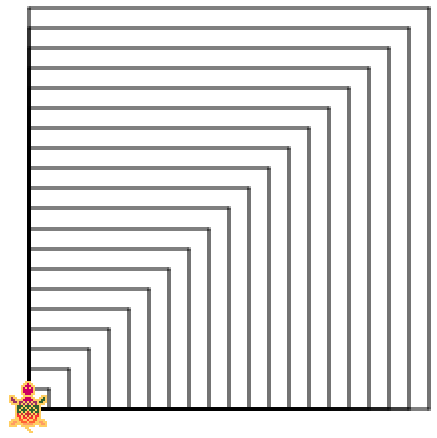
```
TO MYPROC :SIZE
  REPEAT 4 [
    FORWARD :SIZE
    RIGHT 90
  ]
END

MYPROC 200
```







You can rename the procedure at any time. Just click in the MYPROC text field and type the new name. We'll rename it to MYSQUARE. The procedure, its calling block, and the code in the Editor will change to match the new name.



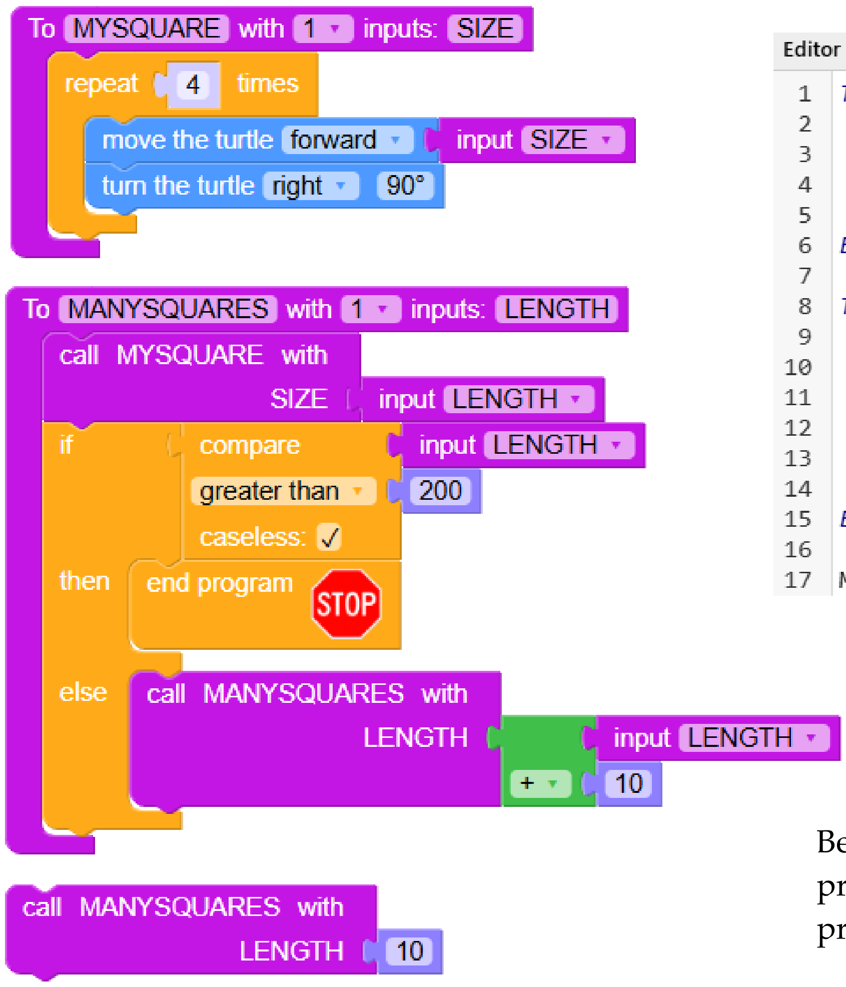
Try calling the MYSQUARE procedure with different input numbers.

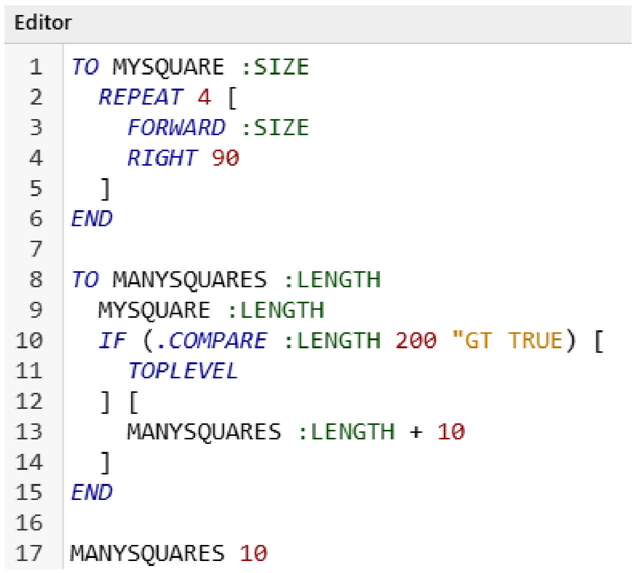
This design was created using inputs of 10, 20, 30, 40, etc., all the way to 200!

Let's save you a lot of typing by automating the process. Try the set of code below.

You'll need to create a new procedure, MYSQUARE, which takes an input for the size.

You also need to create a new variable that calls MYSQUARE multiple times with the length of the side growing by 10 each time. The program stops when the number is greater than 200.

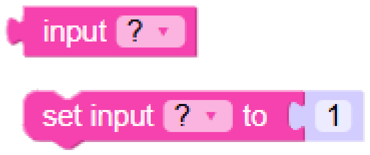




```
TO MYSQUARE :SIZE
  REPEAT 4 [
    FORWARD :SIZE
    RIGHT 90
  ]
END

TO MANYSQUARES :LENGTH
  MYSQUARE :LENGTH
  IF (.COMPARE :LENGTH 200 "GT TRUE) [
    TOPLEVEL
  ] [
    MANYSQUARES :LENGTH + 10
  ]
END

MANYSQUARES 10
```

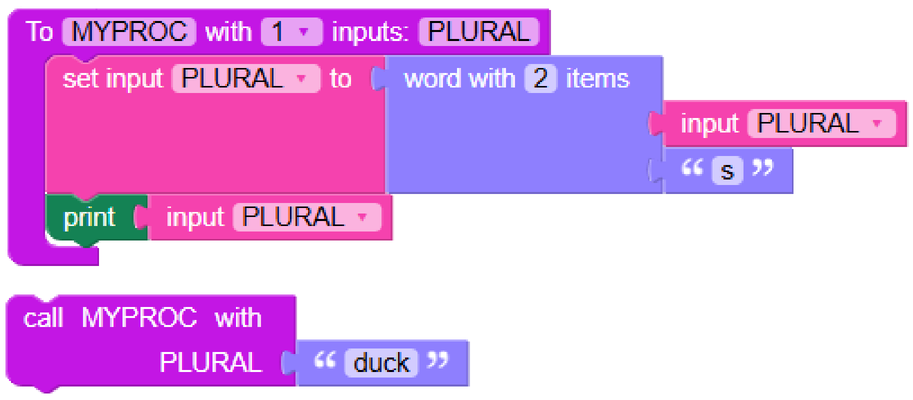
Be sure to put the MYSQUARE procedure above the MANYSQUARES procedure in the Editor.

Do you see how this code works? The Logo command TOPLEVEL in the Editor is the same as the ***end the program*** block. Play with the numbers to draw different shapes in different sizes.



These two blocks are colored pink, not purple, because they are related to variables, and the Variables section's color is purple. In the **Procedures** category, the variables you use are *local variables*, which only can be seen by the procedures you create with an input. (Next, in the Variables section, you will meet *global variables*, which are visible to your entire program.)

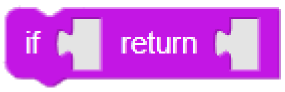
Here is a short program that uses these blocks.



```
To MYPROC with 1 inputs: PLURAL
    set input PLURAL to  word with 2 items
                             input PLURAL
                             " s "
    print  input PLURAL

call MYPROC with
    PLURAL  " duck "
```

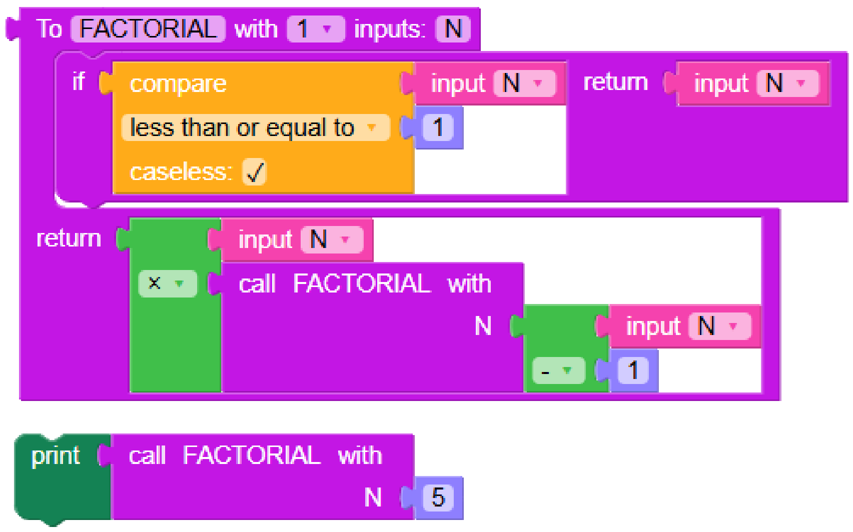
You call the procedure with a singular noun and it replaces that input with the word followed by the letter 's' to make it plural.

Try different singular nouns. Does this program create the correct plural for all of them? If not, perhaps you can code a program that handles exceptions to the rule!



```
if   return
```

This program looks at a condition and returns a value that you define. This program uses it. An explanation follows.

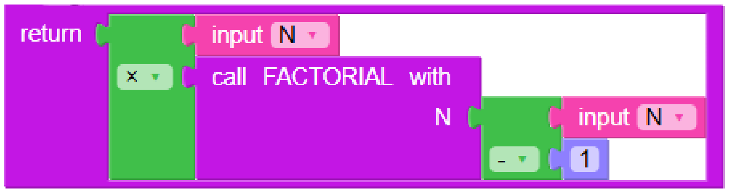


```
To FACTORIAL with 1 inputs: N
    if  compare                      input N     return  input N
        less than or equal to        1
        caseless: ✓
    return      input N
             ×  call FACTORIAL with
                    N      input N
                        -  1

print  call FACTORIAL with
           N  5
```

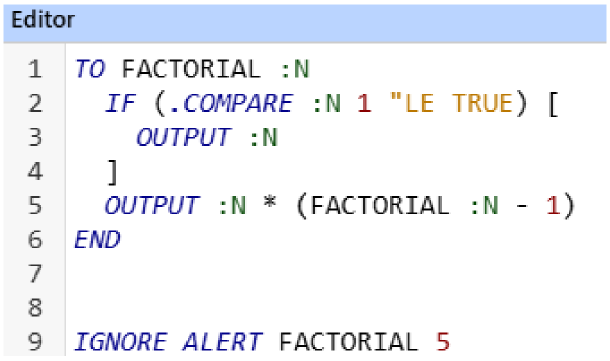
The program is called FACTORIAL because it gives you the factorial of the number you call it with. The factorial of 5 is 120. A factorial is computed by multiplying the number by the number minus one repeatedly until the number reaches 1.

So, 5 x 4 x 3 x 2 x 1 = 120.

This program is recursive, which means that the definition of the program includes a call to itself. Do you see the call to FACTORIAL in the return section, here?

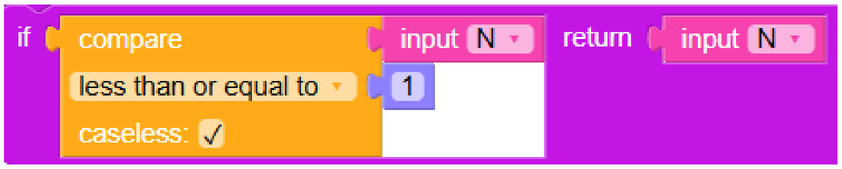


The code in the Editor looks like this:



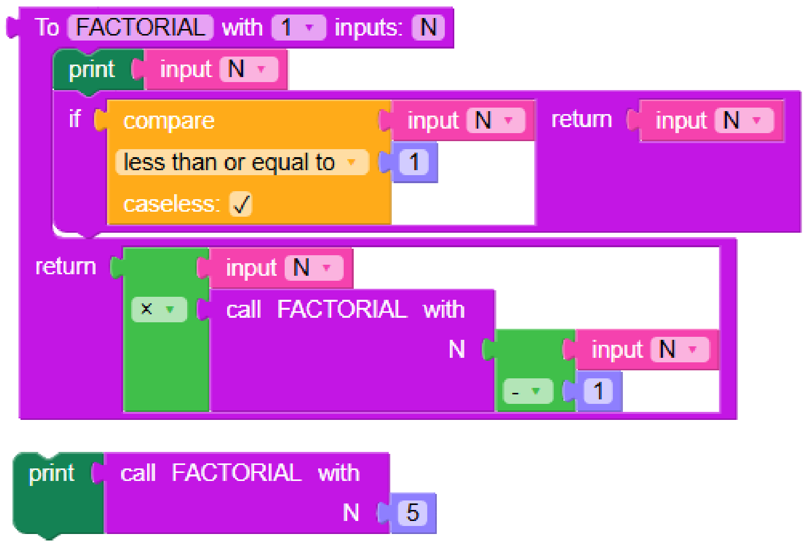
```
TO FACTORIAL :N
  IF (.COMPARE :N 1 "LE TRUE) [
    OUTPUT :N
  ]
  OUTPUT :N * (FACTORIAL :N - 1)
END


IGNORE ALERT FACTORIAL 5
```

You give FACTORIAL a starting number when you call it.
The procedure calls this input N.
The IF part compares the input N to the number 1.

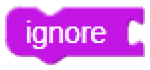


If it is less than or equal to 1, where "LE means <=
(Logo uses: .COMPARE :N 1 "LE TRUE.), the program returns the number stored by the variable N.
If the number is still greater than 1, then FACTORIAL is called again, this time with the value of N minus 1.

Recursion can be complicated to understand at first.
Adding a ***print*** block to follow the value of N might help.



```
To FACTORIAL with 1 inputs: N
  print input N
  if compare input N less than or equal to 1 caseless: ✓
    return input N
  return input N × call FACTORIAL with N (input N - 1)

print call FACTORIAL with N 5
```

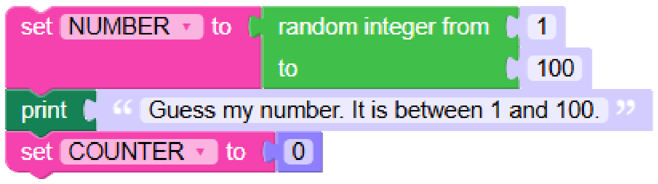


This final block in Procedures allows you to ignore any value you give it. You never know when you might need it!

Before we move to the final section called **Variables**, let’s look at a more advanced version of the number guessing game introduced in the Intermediate tutorial.

Guessing Game

Do you remember the simple number guessing game we explored in the Intermediate tutorial? Here is a more advanced version. Let's take a look at it.

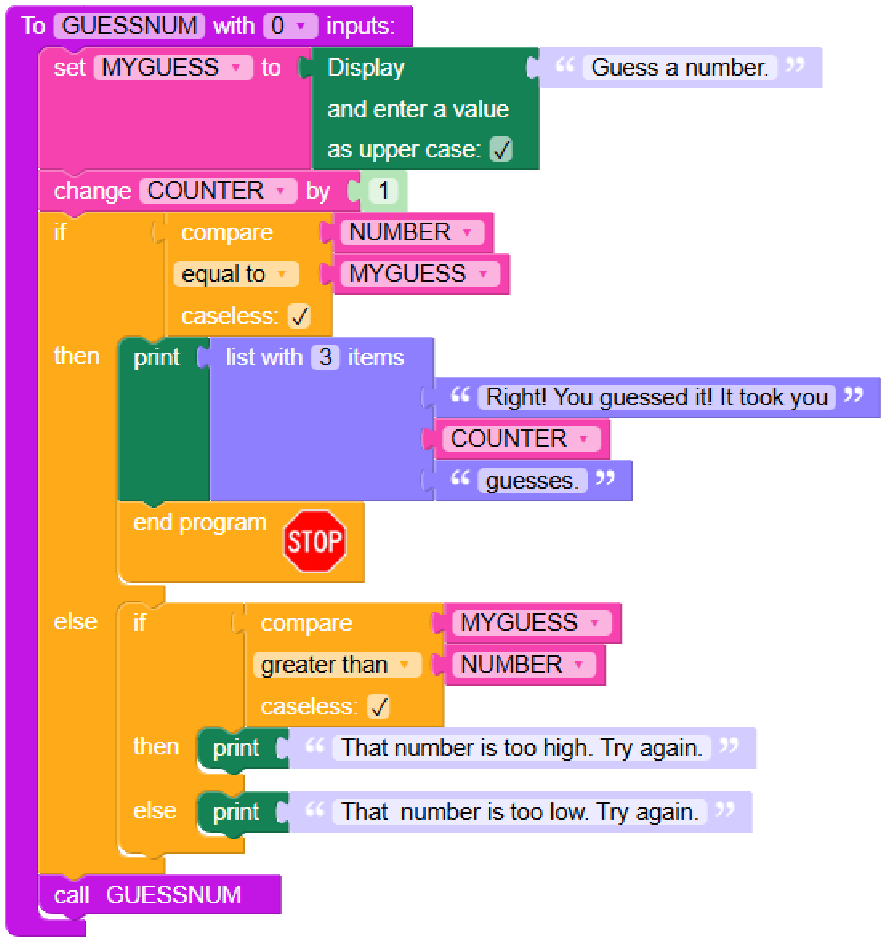


```
set NUMBER to random integer from 1
                                to 100
print "Guess my number. It is between 1 and 100."
set COUNTER to 0
```

First, we need to set up the program. We create a variable called NUMBER, which stores a random number from 1 to 100.

The ***print*** block gives the user instructions.

We then set the COUNTER variable to 0. It will keep track of how many tries it takes to guess the number.



```
To GUESSNUM with 0 inputs:
  set MYGUESS to Display "Guess a number."
                 and enter a value
                 as upper case: ✓
  change COUNTER by 1
  if    compare NUMBER
        equal to MYGUESS
        caseless: ✓
  then  print list with 3 items
                "Right! You guessed it! It took you"
                COUNTER
                "guesses."
        end program STOP
  else  if    compare MYGUESS
              greater than NUMBER
              caseless: ✓
        then  print "That number is too high. Try again."
        else  print "That  number is too low. Try again."
  call GUESSNUM
```

Next, we create the main procedure, called GUESSNUM. The ***display*** block gives instructions and the player types a number. The variable MYGUESS stores that number.

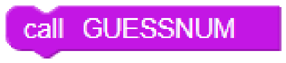
We increase the variable COUNTER by one. It started at zero, so if the player guesses the correct number on the first try, the value of COUNTER is 1.

We compare the variable NUMBER (the answer) to the user's guess (MYGUESS).

If they are the same, we tell the user they got it right and tell them how many tries it took (the value of COUNTER). Then we stop the program.

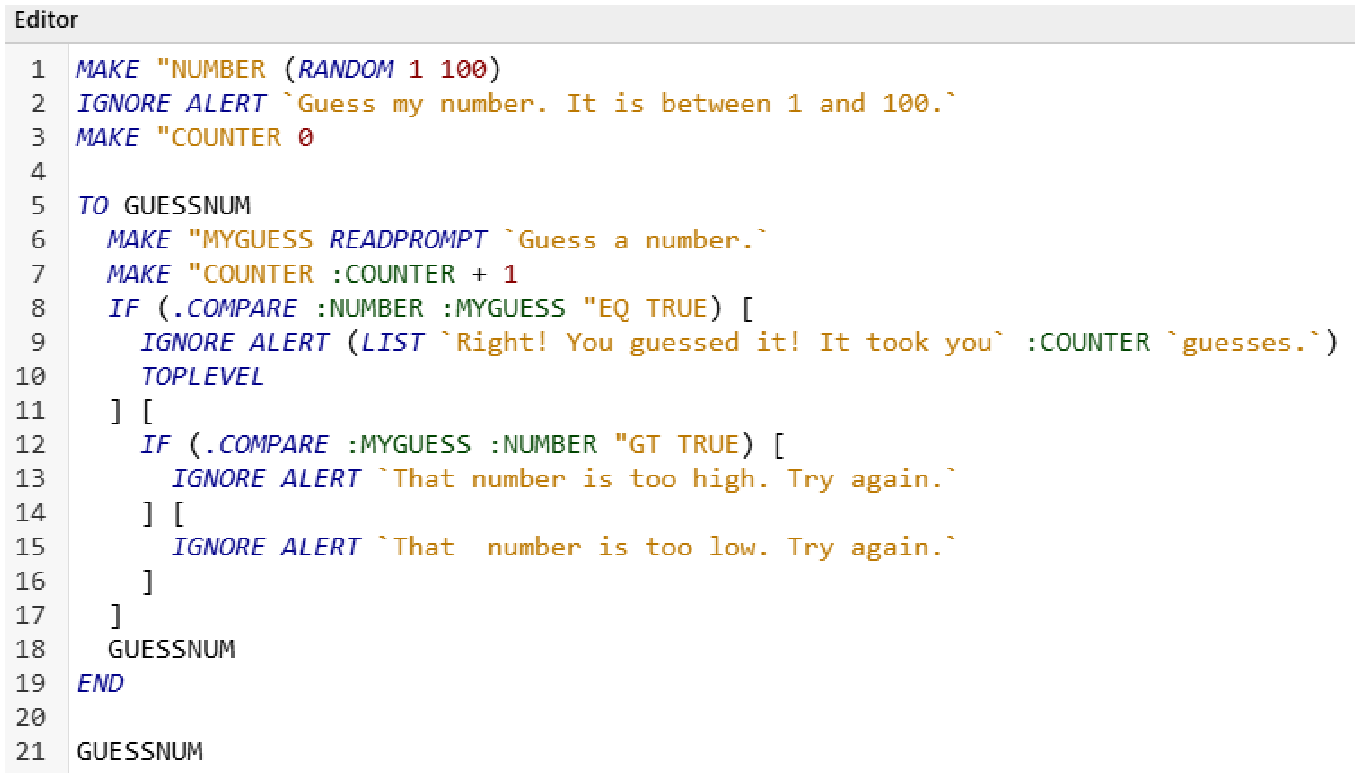
If they get it wrong, we compare their guess (MYGUESS) to the answer (NUMBER). If the MYGUESS was greater than NUMBER, we tell the player and ask them to try again. If MYGUESS is too low, we tell them, and ask them to try again.

After we give the player the message, we call the GUESSNUM procedure again so they can try a different number. When a procedure calls itself, it is known as **recursion**.



The last block in the program is to call GUESSNUM. That starts the program off!

Let’s see what the Logo code in the Editor looks like.



```
MAKE "NUMBER (RANDOM 1 100)
IGNORE ALERT `Guess my number. It is between 1 and 100.`
MAKE "COUNTER 0

TO GUESSNUM
  MAKE "MYGUESS READPROMPT `Guess a number.`
  MAKE "COUNTER :COUNTER + 1
  IF (.COMPARE :NUMBER :MYGUESS "EQ TRUE) [
    IGNORE ALERT (LIST `Right! You guessed it! It took you` :COUNTER `guesses.`)
    TOPLEVEL
  ] [
    IF (.COMPARE :MYGUESS :NUMBER "GT TRUE) [
      IGNORE ALERT `That number is too high. Try again.`
    ] [
      IGNORE ALERT `That  number is too low. Try again.`
    ]
  ]
  GUESSNUM
END

GUESSNUM
```

`MAKE` is the Logo command to create or change a variable.

`MAKE "NUMBER` is set to the random number, which Logo gets using `(RANDOM 1 100`.

`MAKE "COUNTER` sets the value of the variable to 0. That is called initializing the variable.

When we increase the value of COUNTER, we use `MAKE "COUNTER :COUNTER + 1`.
When we refer to a variable that already has a value, we use “dots” or the : before its name.

The `IGNORE` before the `ALERT` command tells Logo it doesn’t have to do anything with what is displayed. `ALERT` can give information back to Logo, but here, Logo can just ignore it.

`IF` needs something to look at; in this case it compares the guess to the random number. If the result is TRUE (they are the same), then Logo runs the instructions in the list that follows it. If the result is FALSE (the guess is wrong), then it runs the list after that. That list is comparing the guess to the answer so it can tell the player it the guess is too high or too low.
Logo is using this syntax or coding strategy to handle these conditional IF/THEN statements.

IF *this-condition-is-true* [then *run this code*] [else *run this code*]

`TOPLEVEL` is the same as Stop the program. That happens if the number is guessed!

If the guess is wrong, the procedure GUESSNUM is called to give the player another chance to guess the number.

Spend some time looking at the program so you can figure out how it works.