In previous lessons, we have covered the pseudocode for multiple algorithms (e.g., searching and sorting). This lesson explores how to translate that pseudocode into actual code that a computer can understand and execute.

Machine language

Definition: ***Machine language*** *is a set of specific instructions that can be executed directly by a computer. This language is typically made up of binary digits (1s and 0s).*

A computer can only understand machine language. Machine language instructions are entirely made up of binary digits and can be directly executed by the CPU. Since we have a particularly hard time understanding 1s and 0s, early programmers assigned a set of mnemonics to represent machine code instructions so that they would be a bit more readable. This mapping became known as **assembly language**. People still write code in assembly language, but it is not typically used to create large scale applications.

Programming language

The kinds of languages that are widely used today are known as programming languages. Programming languages allow us to represent algorithms in a way that is similar to English but is more structured and much less ambiguous.

Definition*: A* ***programming language*** *is a precisely constructed language that is specifically used to communicate instructions to a computer.*

English is a spoken language. As such, it was spoken first, rules were later defined and written down. Therefore, there are many exceptions (i.e., words and phrases that are grammatically correct but don't conform to the general grammar/spelling rules). Spoken languages are sometimes ambiguous and open to interpretation. This means that a single statement can have multiple meanings. For example, the statement “I made the robot fast” can mean several different things. Does it mean that the robot was built quickly? Or does it mean that the robot was modified so that it would move around more quickly than it did before? Perhaps it means that the robot is named Fast. Or maybe that we managed to make the robot stop eating nuts and bolts.

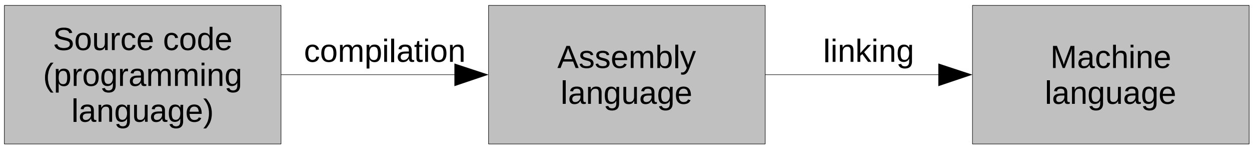
Humans rely on external factors like context and body language to understand the true meaning of a statement in a spoken language. And even then mistakes in interpretation still happen. With computers however, we need a language that is so structured and unambiguous that every computer can understand and interpret a given statement in the exact same way. For example, we don't want two different computers giving us two completely different answers to the arithmetic expression `1 + 1`.

In contrast to spoken languages, programming languages are first defined with rules. The language itself is then derived form those rules. Programming languages are therefore quite structured and not ambiguous. They are very precise and logical.

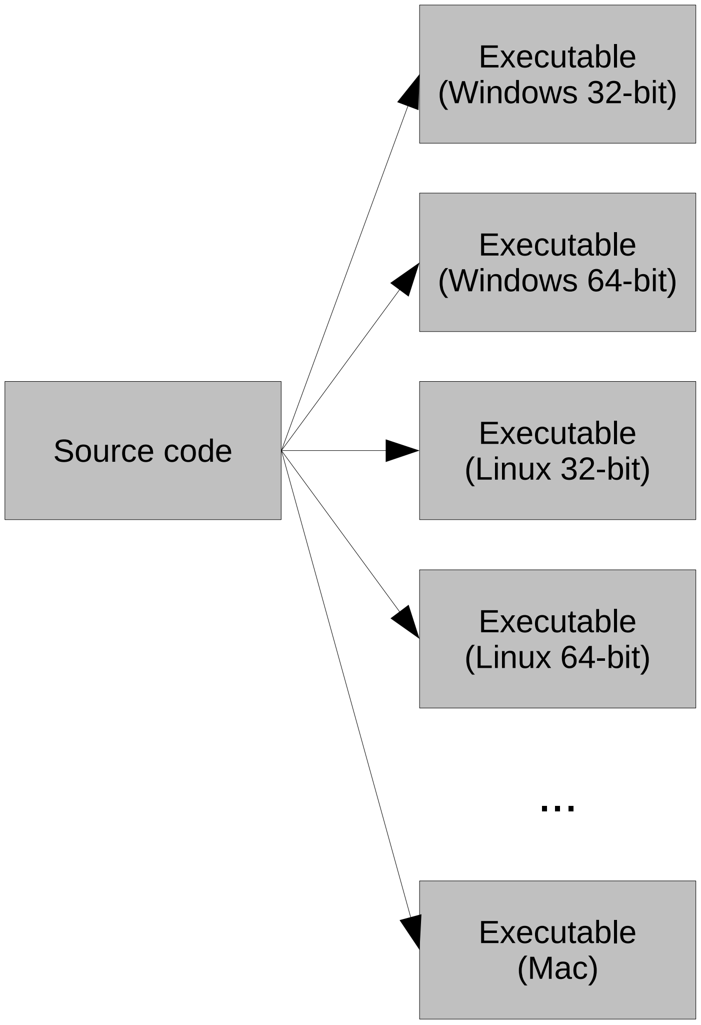
There are many different programming languages that can be used to describe an algorithm. One of them is called Scratch, and it is what we will be using for a part of the Living *with* Cyber curriculum. It is the duty of the programmer to write down the tasks that he/she wants done in a given programming language. Since computers can only understand machine language, we utilize an application known as a **compiler** that translates this programming language into machine language.

***Definition**: A **compiler** is a tool used to translate an algorithm expressed in a programming language to machine language. The process by which this conversion from programming language to machine language is done is called **compilation**.*

The compilation process takes an algorithm written in a programming language and translates it to assembly language. From there, a process known as *linking* converts the assembly language to machine language. This is illustrated by the figure below:



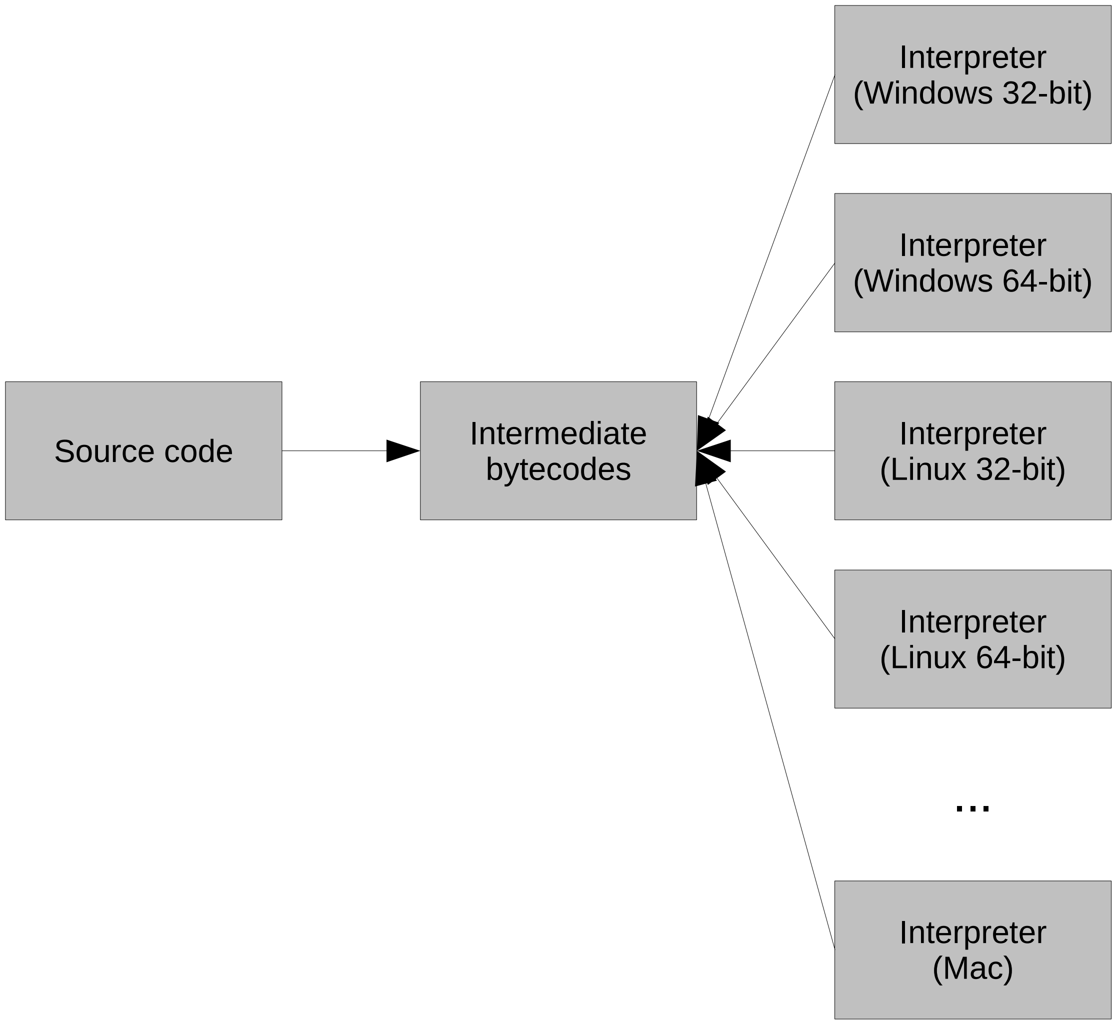
Once machine language is generated to match a program, the computer can then directly execute the program and implement the algorithm. A fully compiled language is only executable by a CPU with the same characteristics and operating system (often, including version) as that which it was compiled for. A programmer who wants wide distribution of his software will need to compile source code to the various destination computing architectures and operating systems that are the most likely to be used by the target audience for the application. Of course, the programmer could simply distribute source code and let users compile that themselves. Often, however, programmers do not wish to distribute source code for a variety of reasons (e.g., intellectual property). The figure below shows how a program would need to be compiled numerous times to cover a range of target computing architectures and operating systems:


Source code
Executable (Windows 32-bit)
Executable (Windows 64-bit)
Executable (Linux 32-bit)
Executable (Linux 64-bit)
...
Executable (Mac)

Not all programming languages are compiled to machine language. Some are never compiled and are executed, one instruction at a time, by an **interpreter**. An interpreter can be thought of as a real time compiler that executes high level programming language instructions, one at a time. Interpreted languages are much slower to execute than compiled languages. Examples of interpreted languages are Python, PHP, JavaScript, and Perl. To execute a program written in an interpreted language, you must have an appropriate interpreter installed on your computing system.

Interpreted languages also require programmers to distribute their source code, and users to have an appropriate interpreter installed on their system. Maintaining code privacy is therefore not possible with interpreted languages.

Partially compiled and interpreted languages combine the convenience of interpreted languages (i.e., not having to compile source code to a large number of target machine language executables) and the privacy and speed of compiled languages (i.e., not having to distribute source code). These types of programming languages are partially compiled (to an intermediate language) and then interpreted from there. Examples of partially compiled languages are Java, Python, and Lisp. Note that Python can be strictly interpreted or partially compiled depending on the programmer's preferences. The intermediate language is distributed and subsequently executed on any computing platform that has an interpreter for the intermediate language. For example, Java source code is typically expressed in a `.java` file and partially compiled to Java bytecodes (in a `.class` file) that can then be distributed. A Java Virtual Machine (JVM) executes the bytecodes by interpreting each instruction, one at a time. The benefit of this method is that a programmer can distribute a single file to everyone, regardless of CPU architecture and operating system. Anyone wanting to execute the file simply needs to have a version of the JVM for their computing system. this is illustrated in the figure below:


Source code
Intermediate bytecodes
Interpreter (Windows 32-bit)
Interpreter (Windows 64-bit)
Interpreter (Linux 32-bit)
Interpreter (Linux 64-bit)
...
Interpreter (Mac)

Programming paradigms

Over the past forty years or so, three general classes, or paradigms, of programming languages have emerged. These paradigms include the imperative paradigm, the functional paradigm, and the logical paradigm. In addition, during the past decade or so these paradigms have been extended to include object-oriented features. A language is classified as belonging to a particular paradigm based on the programming features it supports.

Object-oriented imperative languages are, by far, the most popular type of programming language. Both Java and C++ (two of the most used programming languages in industry) are object-oriented imperative languages. Scratch and Python are imperative languages – although Python does contains object-oriented attributes, Scratch does not.

The **imperative paradigm** is based on the idea that a program is sequence of commands or instructions (usually called **statements**) that the computer is to follow to complete a task. The imperative style of programming is the oldest, and now with object-oriented extensions, continues to be far and away the most popular style of programming.

The Living *with* Cyber curriculum first utilizes Scratch as the programming language. This is quickly followed by Python. Scratch is not intended to be used to create applications designed for production systems. Instead, it is a teaching tool aimed at simplifying the process of learning to program. Scratch purposefully omits many features available in other popular programming languages in order to keep the language from becoming overly complex. This allows you to focus on the *big picture* rather than get bogged down in the complexities inherent in *real* programming languages and their development environments.

One way of thinking about writing Scratch programs is to compare it to programming in a *production* programming language with training wheels on. Complex and useful programs can be written in Scratch; however, there are many things that programmers are allowed to do in production languages that are not possible (at least not straightforward) in Scratch. For example, Scratch does not support functions and function calls directly, nor does it support recursion directly. These terms may not be familiar right now; however, these restrictions are designed to help beginning programmers avoid making common mistakes.

Scratch
Scratch is a basic programming language that utilizes puzzle pieces to represent the properties in the language (e.g., sequence, selection, and repetition). The programmer decides what pieces to use in order to implement the algorithm, and the puzzle pieces help identify what actions or statements can fit with each other and in what order they will be executed. More robust languages such as Python are entirely text-based where statements are text instructions used to represent actions.

In real physical puzzles, certain pieces have meaning and can only be used in certain places (e.g., edge pieces and corner pieces). This is similar in Scratch, in that certain pieces have meaning and must be used in certain places in our programs. Some of a puzzle's pieces can only be combined with certain other pieces so that they make sense.

Scratch programs consist of scripts and is sprite driven. That is, a set of scripts can be defined for each sprite in a Scratch program (which we can more appropriately call a project). Scripts that are executed when the green flag is clicked can be defined for each sprite, and these scripts will execute *simultaneously* for each sprite! Sprites can communicate by way of broadcasting messages that can be received by other sprites. This is, in a way, a characteristic of object-oriented programming languages, where objects can communicate with each other by sending messages.

As covered in a previous lesson, Scratch scripts are made up of various puzzle pieces (or blocks) that serve various functions. Blocks in the *motion* group provide programming constructs that deal with the movement or placement of sprites, while blocks in the *looks* group control anything related to the appearance of sprites (e.g., costume, graphical effect). *Sound* blocks provide the ability to incorporate sound in our Scratch programs, and *pen* blocks allow us to draw on the stage. *Control* blocks provide some of the most powerful functionality in Scratch. They allow us to implement selection and repetition quite easily, and in a variety of useful ways. They also provide ways to allow communication among sprites and to specify scripts to perform when events occur (e.g., when the green flag is clicked). Blocks in the *sensing* group provide ways of specifying input to our programs. We can, for example, detect if a sprite is touching another (and then specify some sort of action if desired). Blocks in the *operators* group provide math and string functions, something quite useful in our programs. These blocks allow us to compare values in order to determine an action to perform. Lastly, blocks in the *variables* group permit us to define variables and lists (i.e., a group of values). This is useful in virtually all programs, and you will find that declaring variables will become pretty routine.

Data types, constants, and variables
The kinds of values that can be expressed in a programming language are known as its **data types**. Scratch supports only two data types: text and numbers. The text data type provides the ability to represent non-numeric data such as names, addresses, English phrases, etc. The numeric data type allows the language to manipulate numbers, both positive and negative, whole numbers and fractions.

A **constant** is defined as a value of a particular type that does not change over time. Both numbers and text may be expressed as constants in Scratch. **Numeric constants** are composed of the digits 0 through 9 and, optionally, a negative sign (for negative numbers), and a decimal point (for floating point numbers). Numeric constants may not contain commas, dollar signs, or any other special symbols. The following are valid Scratch numeric constants: +15, -150, 15.01, 3200.

A **text constant** consists of a sequence of characters (also known as a string of characters – or just a **string**). The following are examples of valid string constants:

> "She turned me into a newt."
> "I got better."
> "Very small rocks."

Note that the quotes surrounding the strings are not actually necessary to define a text constant in Scratch.

A **variable** is defined to be a named object that can store a value of a particular type. Scratch supports two types of variables: text variables and numeric variables. Before a variable can be used, its name must be declared. In many programming languages, both its name and type must be declared; however, Scratch only requires a variable's name to be declared. Variables are declared in Scratch through the *variables* blocks group.

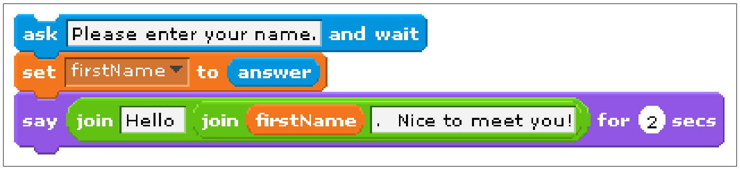
It is important to realize that, while human programmers generally try to give variables names that reflect the use to which they will be put, the variable name itself doesn't mean anything to the computer. For example, the numeric variable `age` can be used to hold any number, not just an age. It is perfectly legal for `age` to hold the number of students in a class or the number of eggs in your refrigerator. The computer couldn't care less. Human programmers, on the other hand, generally care a great deal. They expect a variable's name to accurately reflect its purpose; so while it is possible to do so, it would be considered poor programming practice to use the variable `age` to store anything other than an age.

Input and output statements
In order for a computer program to perform any useful work, it must be able to communicate with the outside world. The process of communicating with the outside world is known as input/output (or I/O). Scratch includes various input and output statements, although they are not implemented in the same way as other *real* programming languages such as Python or Java.

For example, in Scratch, individual sprites can **say *something* for n secs** or **think *something* for n secs**, displaying voice or thought bubbles with text. These are located in the *motion* blocks group. Sprites can also switch costumes, and programs can play sounds, draw with the pen, and so on. These are all output statements. Input statements include sensing when sprites are touching (or near) other sprites, or at the edge of the stage. Scratch can also ask the user for input (either text or numeric), and store this input to a variable. Many input statements in Scratch are located in the *sensing* blocks group. Most imperative languages include mechanisms for performing other kinds of I/O such as detecting where the mouse is pointing and accessing the contents of a disk drive.

The flexibility and power that input statements give programming languages cannot be overstated. Without them the only way to get a program to change its output would be to modify the program code itself, which is something that a typical user cannot be expected to do.

General-purpose programming languages allow human programmers to construct programs that do amazing things. When attempting to understand what a program does, however, it is vitally important to always keep in mind that the computer does not comprehend the meaning of the character strings it manipulates or the significance of the calculations it performs. Take, for example, the following simple Scratch program:

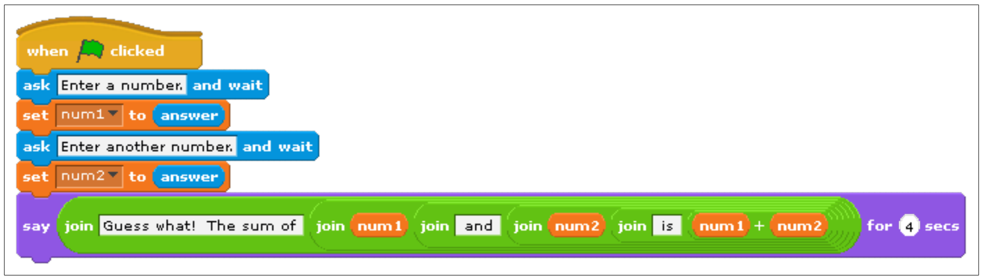


```
ask [Please enter your name.] and wait
set [firstName] to (answer)
say (join [Hello] (join (firstName) [, Nice to meet you!])) for (2) secs
```

This program simply displays strings of characters, stores user input, and echoes that input back to the screen along with some additional character strings. The computer has no clue what the text string "Please enter your name." means. For all it cares, the string could have been "My hovercraft is full of eels." or "qwerty uiop asdf ghjkl;" (or any other text string for that matter). Its only concern is to copy the characters of the text string onto the display screen.

Only in the minds of human beings do the sequence of characters "Please enter your name." take on meaning. If this seems odd, try to remember that comprehension does not even occur in the minds of all humans, only those who are capable of reading and understanding written English. A four year old, for example, would not know how to respond to this prompt because he or she would be unable to read it. This is so despite the fact that if you were to ask the child his or her name, he or she could immediately respond and perhaps even type it out on the keyboard for you.

Consider this Scratch program:



```
when [flag] clicked
ask [Enter a number.] and wait
set [num1] to (answer)
ask [Enter another number.] and wait
set [num2] to (answer)
say (join [Guess what! The sum of ] (join (num1) (join [ and ] (join (num2) (join [ is ] ((num1) + (num2))))))) for (4) secs
```

Here, input is numeric instead of text. The program prompts the user for two numbers, which it then computes the sum for and displays to the user. Note that two variables were declared in the *variables* blocks group: `num1` and `num2`. The first number is captured and stored in the variable `num1`. The second number is captured and stored in the variable `num2`. What do you think would happen is the user did not provide numeric input and, for example, inputted "Bob" for the first number? In the *real world*, programmers must create robust programs that examine user input in order to verify that it is of the proper type before processing that input. If the input is found to be in error, the program must take appropriate corrective action, such as rejecting the invalid input and requesting the user try again.

Primary control constructs
What makes computers more than simple calculating devices is their ability to respond in different ways to different situations. In order to create programs capable of solving more complex tasks we need to examine how the basic instructions we have studied can be organized into higher-level constructs. The vast majority of imperative programming languages support three types of control constructs which are used to group individual statements together and specify the conditions under which they will be executed. These control constructs are: sequence, selection, and repetition.

Sequence requires that the individual statements of a program be executed one after another, in the order that they appear in the program. Sequence is defined implicitly by the physical order of the statements. It does not require an explicit program structure. This is related to our previous discussion on **control flow**.

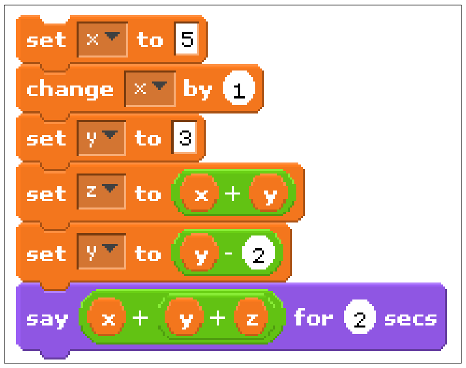
Selection constructs contain one or more blocks of statements and specify the conditions under which the blocks should be executed. Basically, selection allows a human programmer to include within a program one or more blocks of *optional* code along with some tests that the program can use to determine which one of the blocks to perform. Selection allows imperative programs to choose which particular set of actions to perform, based on the conditions that exist at the time the construct is encountered during program execution.

Repetition constructs contain exactly one block of statements together with a mechanism for repeating the statements within the block some number of times. There are two major types of repetition: iteration and recursion. **Iteration**, which is usually implemented directly in a programming language as an explicit program structure, often involves repeating a block of statements either (1) while some condition is true or (2) some fixed number of times. **Recursion** involves a subprogram (e.g., a function) that makes reference to itself. As with sequence, recursion does not normally have an explicit program construct associated with it.

Sequence
Sequence is the most basic control construct. It is the *glue* that holds the individual statements of a program together. Yet, when students who are new to programming try to understand how a particular program works, they often just glance over the various statements making up the program to get a *feel* for what it does rather than methodically tracing through the sequence of actions it performs. One reason such an approach is tempting is because students tend to believe they can figure out what a program is *supposed* to do based on contextual clues such as the meaning of variable names and character strings.

While it is often possible to gain a superficial knowledge of a program simply by reading it, this approach will not give you the kind of detailed understanding that is frequently required to accurately predict a program's output. Being able to carefully trace through a program to determine exactly what it does is an important skill. Failure to carefully follow the sequence of instructions often leads to confusion when trying to understand the behavior of a program.

The following program illustrates the importance of sequence. It contains a little *do nothing* program that displays the value 16. What makes this program interesting is not so much what its output is, as the way in which that output is computed. Without carefully tracing through the program, one statement at a time, it would be difficult to correctly predict the final output generated by the program.



```
set x to 5
change x by 1
set y to 3
set z to (x + y)
set y to (y - 2)
say ((x + (y + z))) for 2 secs
```

Note that the variables x, y, and z were declared in the variables blocks group. The following illustrates the state of the program's memory after executing each line of code. After performing the first declaration, the program knows only about the variable x. After the second declaration, it knows of x and y, and after the third, it knows of x, y, and z. Since these variables have not yet been assigned values, their values are considered to be undefined at this point.

declaring *x*	*x*					
declaring *y*	*x*		*y*			
declaring *z*	*x*		*y*		*z*	
set *x* to 5	*x*	5	*y*		*z*	
change *x* by 1	*x*	6	*y*		*z*	
set *y* to 3	*x*	6	*y*	3	*z*	
set *z* to *x* + *y*	*x*	6	*y*	3	*z*	9
set *y* to *y* – 2	*x*	6	*y*	1	*z*	9
say *x* + *y* + *z* for 2 secs	*x*	6	*y*	1	*z*	9

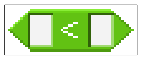
Program output: `16`

Selection

Selection statements give imperative languages the ability to make choices based on the results of certain condition tests. These condition tests take the form of **Boolean expressions**, which are expressions that evaluate to either *true* or *false*. There are various types of Boolean expressions, but the types supported in Scratch are based on relational operators. **Relational operators** compare two expressions of like type to determine whether their values satisfy some criterion. The general form of all Boolean expressions supported in Scratch is:

expression relational_operator expression

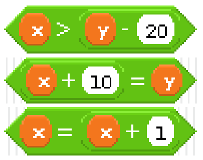
For example:



Scratch includes three relational operators for comparing numeric expressions:

< meaning *less than*
= meaning *equal to*
> meaning *greater than*

For example, when *x* is 15 and *y* is 25, the expression *x > y* evaluates to false, since 15 is not greater than 25. Here are some additional examples of Boolean expressions that use these relational operators. These examples assume that the variable *x* holds 15 and *y* holds 25:



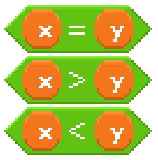
true `15 > 25 - 20`
true `15 + 10 = 25`
false `15 = 15 + 1`

Notice that the last expression always evaluates to *false* regardless of the value of x. This is because there is no possible value for x that will be equal to that value plus one. Another point illustrated by these examples is that relational operators have a lower precedence than mathematical operators. During expression evaluation, all multiplication, division, addition, and subtraction operations are performed before any relational operations.

The relational operators also work for text expressions as follows:

< meaning *precedes*
= meaning *equal to*
> meaning *follows*

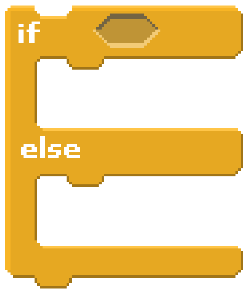
Note that Scratch does not differentiate between uppercase and lowercase letters. That is, `A` is equal to `a`. Here are some examples, assuming that the variable *x* holds `Bob` and *y* holds `bobcat`:



false `Bob = bobcat`
false `Bob > bobcat`
true `Bob < bobcat`

Since `Bob` precedes `bobcat` in alphabetical order, `Bob < bobcat` is *true*.

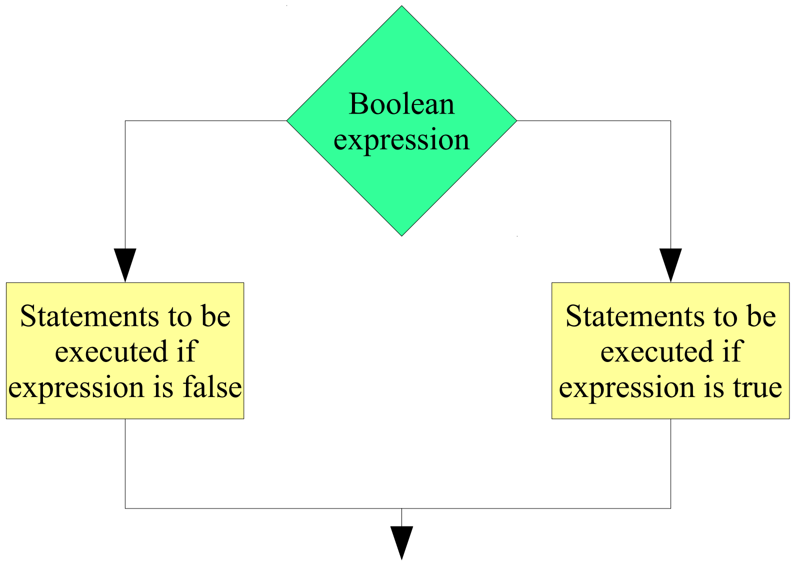
Selection statements use the results of Boolean expressions to choose which sequence of actions to perform next. Scratch supports two different selection statements: ***if-else*** and ***if***. An *if-else* statement allows a program to make a two-way choice based on the result of a Boolean expression. The general form of an *if-else* statements is shown below:



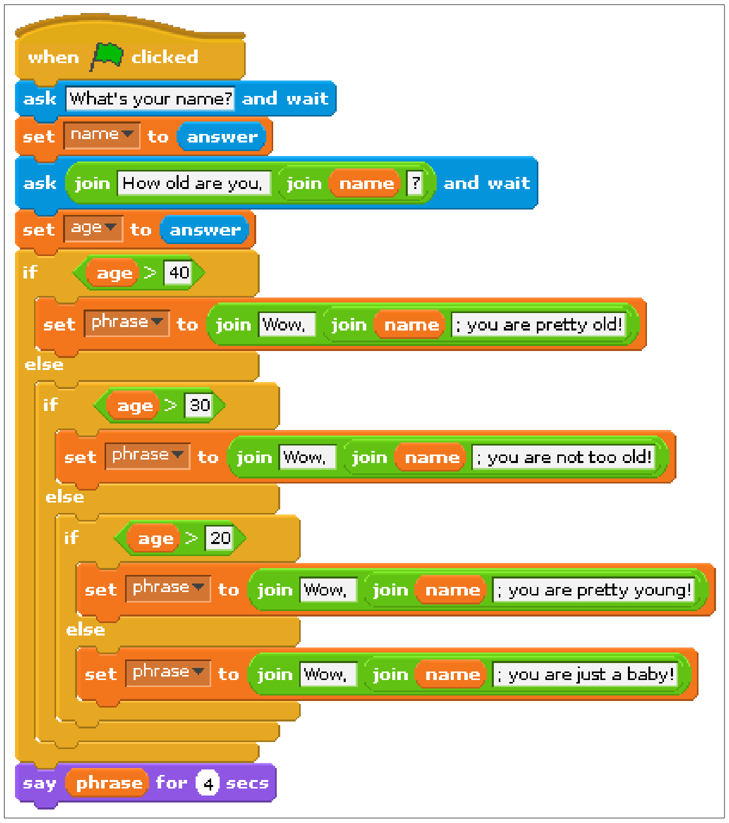
← statements to be executed if the expression is *true*

← statements to be executed if the expression is *false*

If-else statements specify a Boolean expression and two separate blocks of code: one that is to be executed if the expression is true, the other to be executed if the expression is false. Here's a flowchart for an *if-else* statement:


Boolean expression
Statements to be executed if expression is false
Statements to be executed if expression is true

And here's an example of a program that implements an *if-else* statement (several, actually):

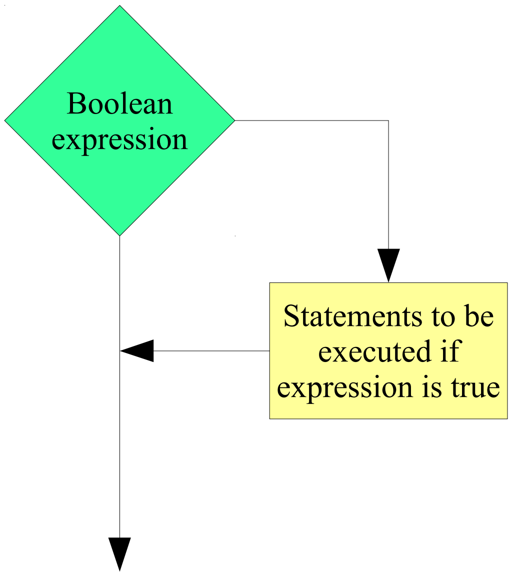


```
when [flag] clicked
ask [What's your name?] and wait
set [name] to (answer)
ask (join [How old are you, ] (join (name) [?])) and wait
set [age] to (answer)
if <(age) > [40]>
    set [phrase] to (join [Wow, ] (join (name) [; you are pretty old!]))
else
    if <(age) > [30]>
        set [phrase] to (join [Wow, ] (join (name) [; you are not too old!]))
    else
        if <(age) > [20]>
            set [phrase] to (join [Wow, ] (join (name) [; you are pretty young!]))
        else
            set [phrase] to (join [Wow, ] (join (name) [; you are just a baby!]))
say (phrase) for (4) secs
```

This program prompts the user to enter a name and age, and responds appropriately. It compares the user's age to several constants (40, 30, and 20), and sets the variable phrase depending on the user's age. If the user's age is more than 40, then the program responds that the user is pretty old; otherwise, the program then checks to see if the user's age is more than 30. If so, the program responds that the user is not too old; otherwise, it then checks to see if the user's age is more than 20. If so, it responds that the user is pretty young; otherwise (any age less than or equal to 20), it responds that the user is just a baby.

Note that we can nest an *if-else* statement inside of another *if-else* statement to provide more than two alternatives or paths. Try to represent the program above in pseudocode in the space below:

The *if* statement is similar to the *if-else* statement except that it does not include an *else* block. That is, it only specifies what to do if the Boolean expression is true. Here's a flowchart for an *if* statement:

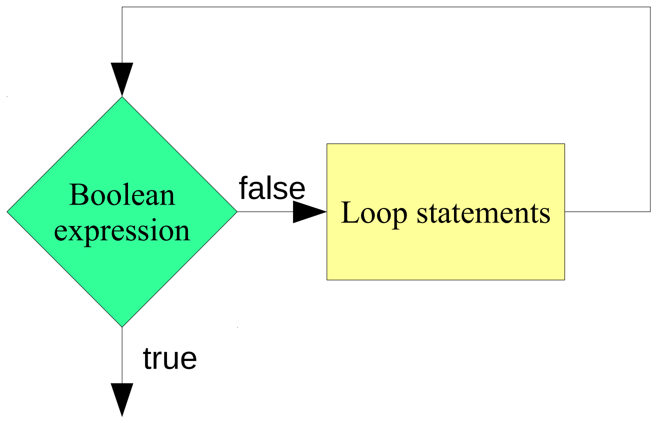


If statements are generally used by programmers to allow their programs to detect and handle conditions that require *special* or *additional* processing. This is in contrast to *if-else* statements, which can be viewed as selecting between two (or more) *equal* choices.

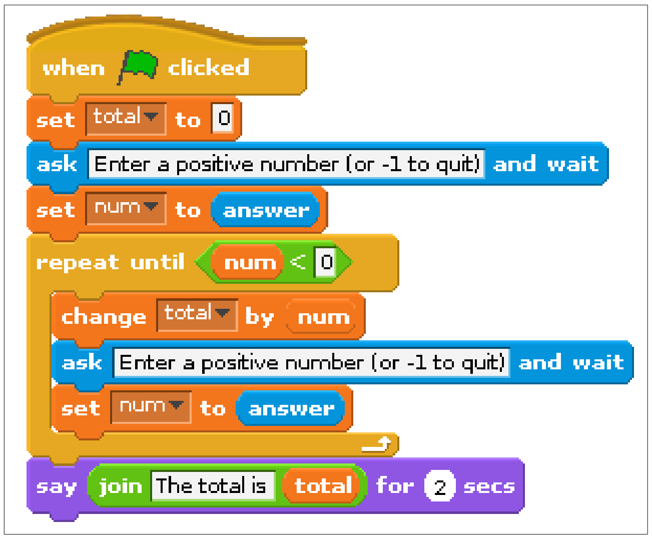
Repetition
Repetition is the name given to the class of control constructs that allow computer programs to repeat a task over and over. This is useful, particularly when considering the idea of solving problems by decomposing them into repeatable steps. There are two primary forms of repetition: iteration and recursion.

Scratch supports iteration in two main forms: the *repeat* loop and the *forever* loop. The *repeat* loop has two forms: *repeat-until* and *repeat-n* (where *n* is some fixed or known number of times). The *repeat-until* loop is condition-based; that is, it executes the statements of the loop until a condition becomes true. The *repeat-n* loop is count-based; that is, it executes the statements of the loop *n* times.

Here's a flowchart of the *repeat-until* loop:



The Boolean expression is first evaluated. If it evaluates to false, the loop statements are executed; otherwise, the loop halts. Here is an example:

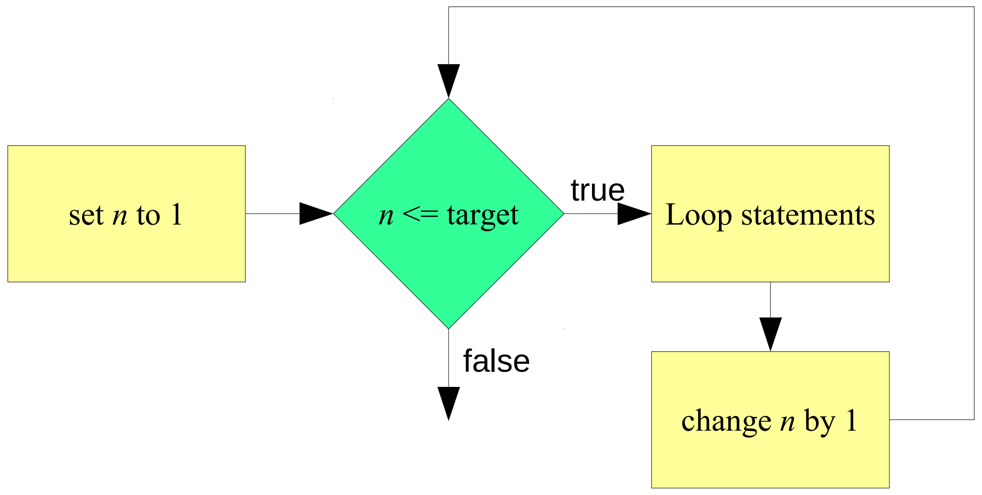


```
when [flag] clicked
set total to 0
ask Enter a positive number (or -1 to quit) and wait
set num to answer
repeat until num < 0
    change total by num
    ask Enter a positive number (or -1 to quit) and wait
    set num to answer
say join The total is  total for 2 secs
```

This program asks the user to enter a positive number or -1. If a positive number is entered, it is added to a running total. If -1 is entered, the program displays the total and halts. The *repeat-until* loop is used here to repeat the process of asking the user for input until the value entered is less than 0. It is interesting to note that although the program instructs users to enter -1 to quit, the condition that controls the loop is actually `num < 0` (which will be false for any negative number). Thus, the loop will actually terminate whenever the user enters any number less than zero (e.g., -5).

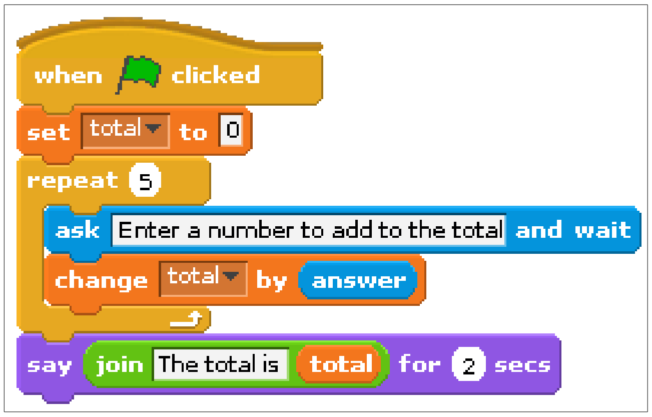
Try to represent the program above in pseudocode in the space below:

The *repeat-n* loop executes the loop statements a fixed (or known) number of times. Here's a flowchart of the *repeat-n* loop:



Although the programmer does not have access to a variable that counts the specified number of times (shown as *n* in the figure above), the process works in this manner. A counter is initially set to 1. A Boolean expression is then evaluated that checks to see if that counter is less than or equal to the target value (e.g., 10). If so, the loop statements execute. Once the loop statements have completed, the counter is incremented, and the expression is reevaluated.

Here is an example:

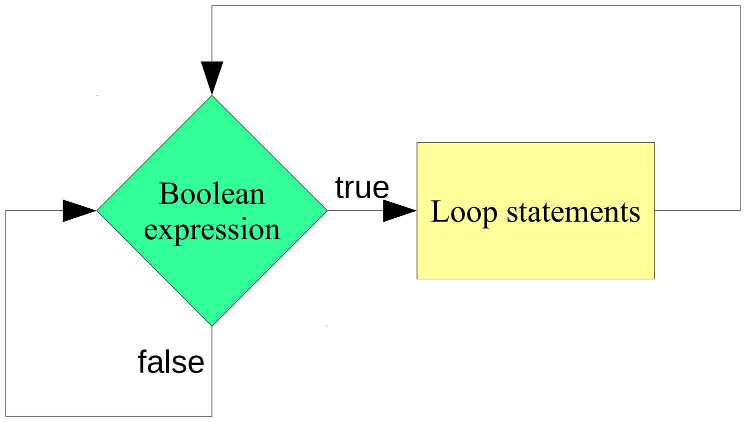


```
when [flag] clicked
set total to 0
repeat 5
    ask Enter a number to add to the total and wait
    change total by answer
say join The total is total for 2 secs
```

This program asks the user to enter five numbers. Each time, the number is added to a running total. After all five numbers have been entered, the total is displayed.

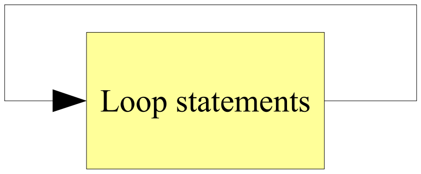
The forever loop also has two forms: *forever* and *forever-if*. The *forever* loop executes the statements of the loop forever. Well, it's not technically forever, since we can click the stop button to halt all scripts at any time. The *forever-if* loop is condition-based (like the *repeat-until* loop), and executes the statements of the loop if a condition is true. Note that the *forever-if* loop also runs forever (until the stop button is clicked). the difference is that the loop statements are only executed if the condition is true.

Here is a flowchart of the *forever-if* loop:



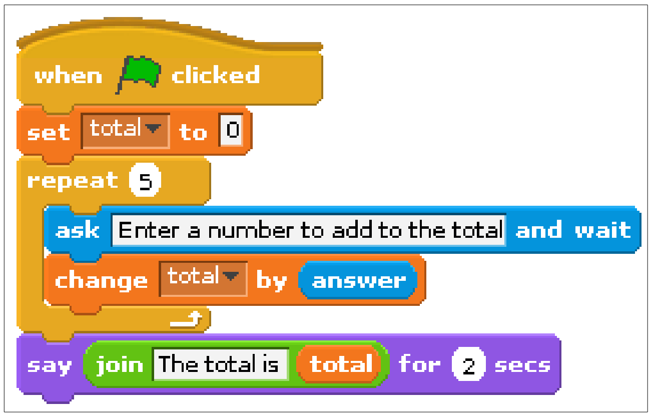
This loop construct is often used to perform real time checking of sprites and execute statements if, for example, the sprite is at a certain position on the stage. Another example is to constantly check the value of a variable that is changed by some other sprite. In this way, a sprite can monitor a variable, and when changed to an appropriate value, perform some action.

Lastly, here is a flowchart of the *forever* loop:



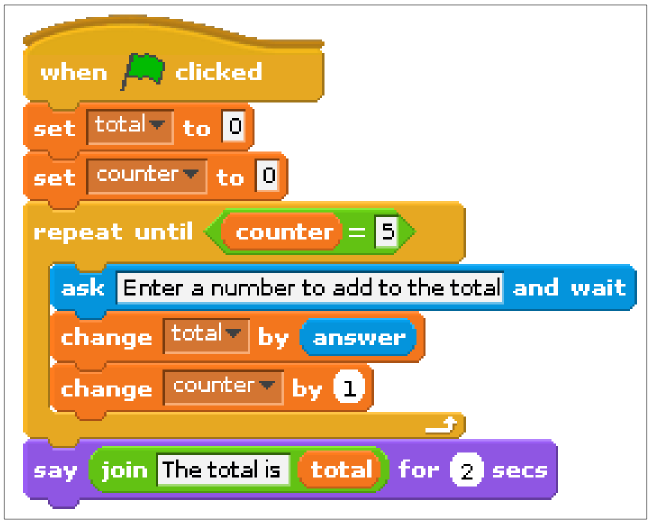
Pretty straightforward...

Note that any program written using a *repeat-n* loop can be rewritten as a *repeat-until* loop. Take, for example the *repeat-n* loop shown earlier:



```
when [green flag] clicked
set total to 0
repeat 5
    ask Enter a number to add to the total and wait
    change total by answer
say join The total is total for 2 secs
```

Here it is, rewritten using a *repeat-until* loop:



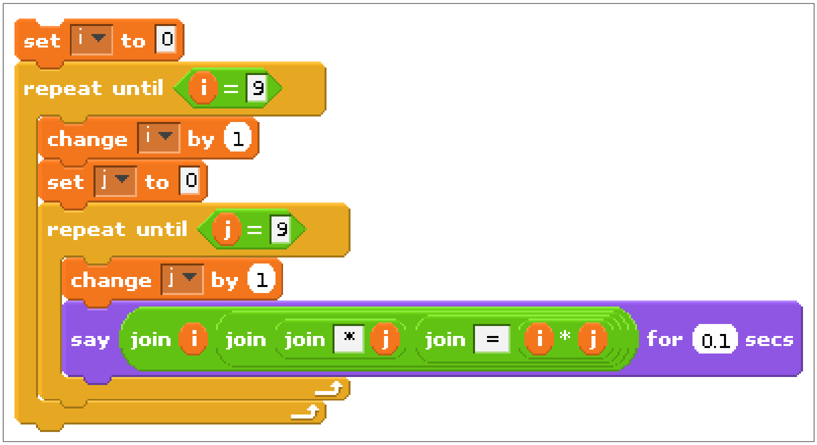
```
when [green flag] clicked
set total to 0
set counter to 0
repeat until counter = 5
    ask Enter a number to add to the total and wait
    change total by answer
    change counter by 1
say join The total is total for 2 secs
```

The only difference is that, in the *repeat-until* loop, the programmer must keep track of (and modify) the counter each time the loop statements execute. In the *repeat-n* loop, Scratch automatically takes care of this.

Is the opposite true? That is, can every program that uses a *repeat-until* loop be rewritten using a *repeat-n* loop? The answer is no. A *repeat-n* loop simply loops a fixed or known number of times. A *repeat-until* loop repeats until some condition is true. That condition could be, for example, when the user inputs -1 to terminate. The idea of expressing a condition that represents a sentinel value in a manner that requires knowing how many times the loop statements will execute is nonsensical. There is no way to tell how many times the loop statements will execute until the user inputs -1. It could be the very first time (or the 10,000th). Because *repeat-n* loops can always be replaced with *repeat-until* loops, but not all *repeat-until* loops can be replaced with *repeat-n* loops, we say that the *repeat-until* loop is more general than the *repeat-n* loop.

Before leaving the topic of iteration, we should say a few words about the idea of *nested* loops. Two loops are nested when one loop appears within the body of another loop. Since there are no restrictions as to what statements can appear within the body of a loop, any of the *repeat* or *forever* loops can certainly appear within the body of another. Nesting of loops is quite common, and in fact may be carried out to an arbitrary depth. However, to keep the logic of a program from becoming too hard to follow, programmers try to limit nesting depths to no more than three or four levels deep.

The following script presents a compact little program for displaying the multiplication table. This program consists of two nested *repeat-until* loops. The loop variable of the outer loop is `i`, and the loop variable for the inner loop is `j`. Both of these loops count from one to nine:



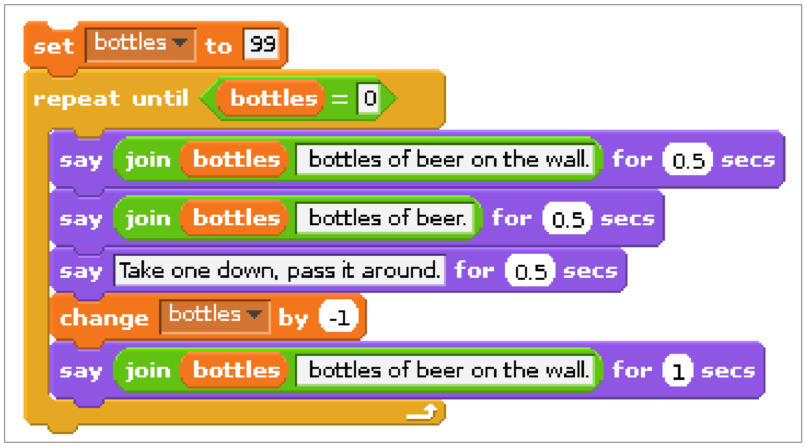
```
set i to 0
repeat until (i = 9)
    change i by 1
    set j to 0
    repeat until (j = 9)
        change j by 1
        say (join (i) (join (join [*] (j)) (join [=] ((i) * (j))))) for (0.1) secs
```

The program's output is of the form `i * j = x`, where `i` and `j` represent the values of the respective variables, and *x* is their product. Notice that `j` runs through its entire range, from 1 to 9, before `i` is incremented by 1. This behavior is easy to understand when you think about the structure of the program.

Let's look at the outer loop. What does it do? Well, first `i` is initialized to 0, it is then compared to 9, and since it is not equal, the first iteration of the loop commences. The first statement of the loop increments `i` and then initializes `j` to 0. The next statement is another *repeat-until* loop. In order for the first iteration of the outer loop to complete, the program must execute this inner loop to completion.

The process repeats until all 81 entries in the multiplication table, from 1 × 1 to 9 × 9, are computed and displayed.

We conclude this section with the following program:


```
set bottles to 99
repeat until bottles = 0
    say join bottles bottles of beer on the wall. for 0.5 secs
    say join bottles bottles of beer. for 0.5 secs
    say Take one down, pass it around. for 0.5 secs
    change bottles by -1
    say join bottles bottles of beer on the wall. for 1 secs
```

This program displays the lyrics to the song, "99 Bottles of Beer on the Wall." As you most likely know, the song begins as follows:

> 99 bottles of beer on the wall.
> 99 bottles of beer.
> Take one down, pass it around.
> 98 bottles of beer on the wall.
>
> 98 bottles of beer on the wall.
> 98 bottles of beer.
> Take one down, pass it around.
> 97 bottles of beer on the wall.

It continues in this manner, with one less bottle in each verse, until it finally runs out of beer. An interesting feature of the program is that it decrements `bottles` in the middle of the loop rather than at the end. You should trace through the program with a few bottles to convince yourself that it does work properly. One thing you will probably notice as you do so is that when the program gets down to one beer, it reports that as "1 bottles of beer on the wall." While this lack of grammatical correctness might not seem like such a big deal, especially after 98 beers, we can correct it easily with an *if* statement.

Functions

This section is concerned with creating and using function subprograms. A **subprogram** is a program within a program. Recall an earlier lesson on representing algorithms as to-do lists. One algorithm represented the steps necessary to *get to class*. One of those steps was *eat breakfast*. We noted how we could zoom in to that step and identify the sub-steps necessary to complete the *eat breakfast* step. Control flow shifted from the main to-do list to the *eat breakfast* to-do list when the *eat breakfast* step was encountered, and then returned to the main to-do list at the point where it left earlier. We can consider the *eat breakfast* to-do list as a subprogram.

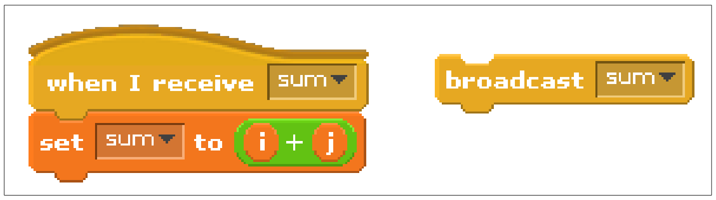
Very few *real* programs are written as one long piece of code. Instead, traditional imperative programs generally consist of large numbers of relatively simple subprograms that work together to accomplish some complex task. While it is theoretically possible to write large programs without the use of subprograms, as a practical matter any significant program must be decomposed into manageable pieces if humans are to write and maintain it.

Subprograms make the construction of software libraries possible. A **software library** is a collection of subprograms, or *routines* as they are sometimes called, for solving common problems that have been written, tested, and debugged. Most programming languages come with extensive libraries for performing mathematical and text string operations and for building graphical user interfaces. These languages allow programmers to include library routines in their code. Using subprograms from the library speeds up the software development process and results in a more reliable finished product.

When a subprogram is invoked, or called, from within a program, the *calling* program pauses temporarily so that the *called* subprogram can carry out its actions. Eventually, the called subprogram will complete its task and control will once again return to the *caller*. When this occurs, the calling program *wakes up* and resumes its execution from the point it was at when the call took place.

Subprograms can call other subprograms (including copies of themselves as we will see later). These subprograms can, in turn, call other subprograms. This chain of subprogram invocations can extend to an arbitrary depth as long as the *bottom* of the chain is eventually reached. It is necessary that infinite calling sequences be avoided, since each subprogram in the chain of subprogram invocations must eventually complete its task and return control to the program that called it.

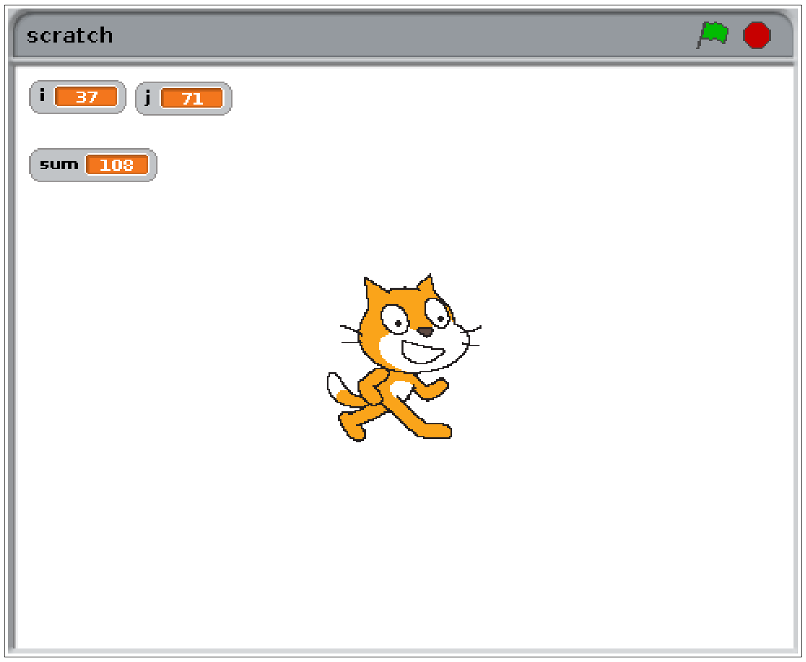
In Scratch, we define subprograms as broadcasts. That is, a sprite can broadcast a message that can be received by another sprite (or even the same sprite). We can think of this as calling a subprogram. When receiving a broadcast, we can specify the script (subprogram) associated with it. Here is a simple example of a subprogram that computes the sum of two numbers stored in the variables `i` and `j`:



```
when I receive [sum]
set [sum] to ((i) + (j))
```

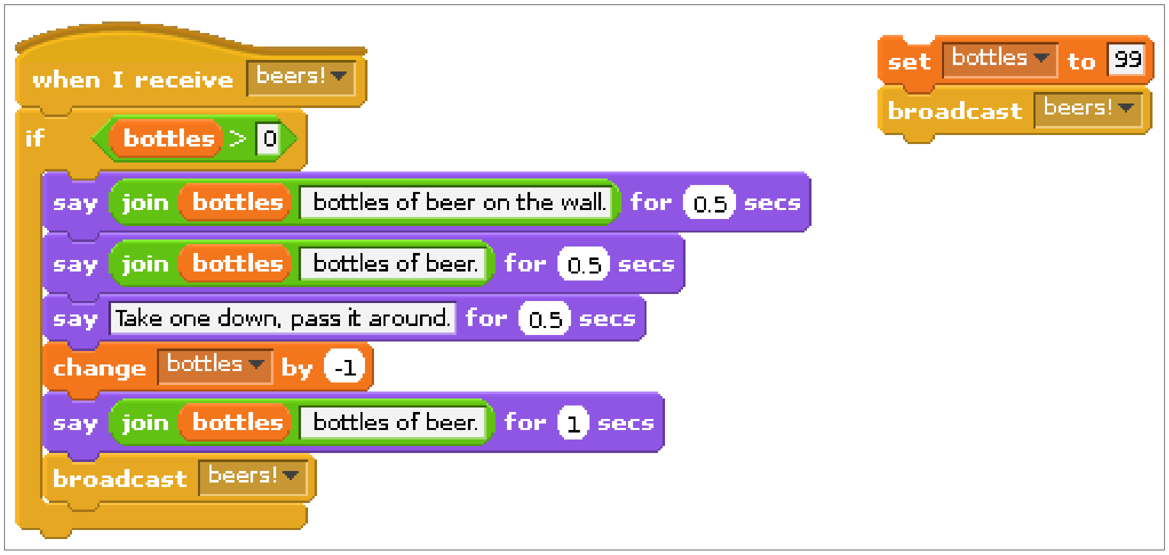
```
broadcast [sum]
```

The left side shows the subprogram defined by receiving the broadcast `sum`. The right side shows the broadcast (or the subprogram *call*). Assuming that the variables `i` and `j` have been declared and given numeric values, then the *sum* subprogram would add the two variables together and store the result to the variable `sum`. Here is an example when `i` is 37 and `j` is 71 (note that the *sum* subprogram has been *called* by being broadcasted):



Recursion
This section is concerned with the topic of recursion. **Recursion** is a type of repetition that is implemented when a subprogram calls itself. When a recursive call takes place, control is passed to what appears to be a brand new copy of the subprogram. This copy of the subprogram may, in turn, call another copy of the subprogram. That copy may call another copy, and so on. Eventually, these recursive calls must terminate and return control to the original calling program.

Recall the "99 Bottles of Beer on the Wall" program that was shown earlier. It illustrated an iterative method of singing the song. Here's an example of the same program implemented using recursion:



```
when I receive [beers!]
if <(bottles) > [0]>
    say (join (bottles) [bottles of beer on the wall.]) for (0.5) secs
    say (join (bottles) [bottles of beer.]) for (0.5) secs
    say [Take one down, pass it around.] for (0.5) secs
    change [bottles] by (-1)
    say (join (bottles) [bottles of beer.]) for (1) secs
    broadcast [beers!]

set [bottles] to [99]
broadcast [beers!]
```

At first glance you might think that this program is an identical copy of the program shown earlier. There are, however, two major differences between the two. First, instead of a *repeat-until* loop, this program has an *if* statement. Also, just before the end of the *if* statement, the same subprogram (*beers!*) is called via a broadcast. This is the recursive call!

Prior to that, note that the value of the variable `bottles` is decremented by 1 (i.e., changed by -1). When control enters the subprogram, the value is checked to see if it is greater than 0. This ensures that, so long as `bottles` is decremented each time the subprogram executes, it will eventually reach 0. When this occurs, it will cause the Boolean expression in the *if* statement to evaluate to false, thereby not executing its block (and calling itself again) and stopping the recursion.

One way to envision recursion is to think of it as a spiral. Each time a subprogram calls itself, we descend down a level of the spiral until we eventually reach the bottom. At that point, execution begins to *unwind* as the subprogram calls complete and we retrace our path back up through the various levels until finally arriving at the *top* level where execution began.

The recursive program above illustrates what is called *tail recursion*, because the recursive call is the last action taken by the subprogram. In tail recursion, there is no work to be done during the *unwinding* process because it was all done on the way *down* the spiral. In Scratch, this is the only way of implementing recursion. As we will find out later, other programming languages permit other forms of recursion, where the recursive call can occur before other statements.

Many students, upon learning how recursion works, worry that programs that employ this form of repetition might be very inefficient in terms of their utilization of machine resources. After all, you have all of those *copies* of the subprogram hanging around. The good news is that recursion is not nearly as expensive as you probably think. For one thing, only one copy of the actual subprogram code is needed. All that is reproduced during each call is the *execution environment*, the variables and whatnot that are used by that *version* of the subprogram. While it is true that recursion generally involves more overhead than iteration, recursive calls are really no more expensive than any other kind of function call. In fact, some optimizing compilers convert tail recursion into iteration so there is often no additional expense in using that form of recursion at all.

Aside from the efficiency issue, you may be wondering why programming languages would support recursion. After all, whenever the need for repetition arises the programmer could always use one of the iteration constructs. The reasons for supporting both recursion and iteration are the same as those for supporting two types of selection statements (*if* and *if-else*) as well as two types of iteration constructs (*repeat* and *forever*): clarity and convenience. Some problems are simply easier to solve using recursion than iteration. For these types of problems, a recursive solution is often more compact and easier to read than an iterative one.