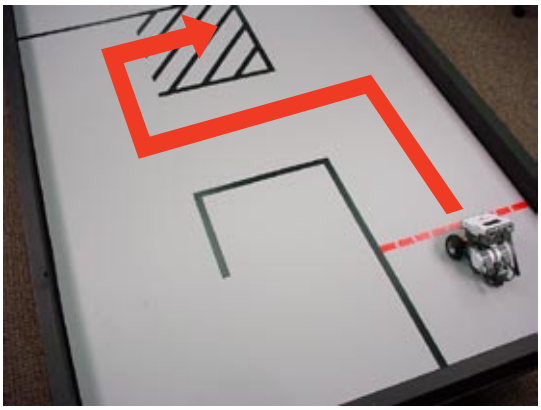
Behaviors

A **behavior** is anything your robot does: turning on a single motor is a behavior, moving forward is a behavior, tracking a line is a behavior, navigating a maze is a behavior. There are three main types of behaviors that we are concerned with: **basic** behaviors, **simple** behaviors, and **complex** behaviors.

Basic Behaviors

Example: Turn on Motor C at 100% power



At the most basic level, everything in a program must be broken down into tiny behaviors that your robot can understand and perform directly. In ROBOTC, these are behaviors the size of **single statements**, like **turning on a single motor**, or **resetting a timer**.

Simple Behaviors

Example: Move forward for 3 seconds

Simple behaviors are small, bite-size behaviors that allow your robot to perform a **simple, yet significant task**, like **moving forward for a certain amount of time**. These are perhaps the most useful behaviors to think about, because they are big enough that you can describe **useful actions** with them, but small enough that you can program them easily from basic ROBOTC commands.

Complex Behaviors

Example: Follow a defined path through an entire maze

These are behaviors at the **highest levels**, such as **navigating an entire maze**. Though they may seem complicated, one nice property of complex behaviors is that they are always composed of smaller behaviors. If you observe a complex behavior, you can always break it down into smaller and smaller behaviors until you eventually reach something you recognize.

```
task main()
{
  motor[motorC] = 50;
  motor[motorB] = 50;
  wait1Msec(2000);

  motor[motorC] = -50;
  motor[motorB] = 50;
  wait1Msec(800);

  motor[motorC] = 50;
  motor[motorB] = 50;
  wait1Msec(2000);
}
```

Basic behavior
This code turns the left motor on at 50% power.

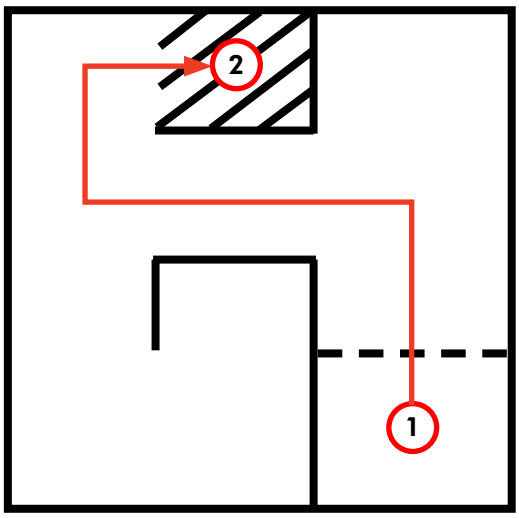
Simple behavior
This code makes the robot go forward for 2 seconds at 50% power.

Complex behavior
This code makes the robot move around a corner.

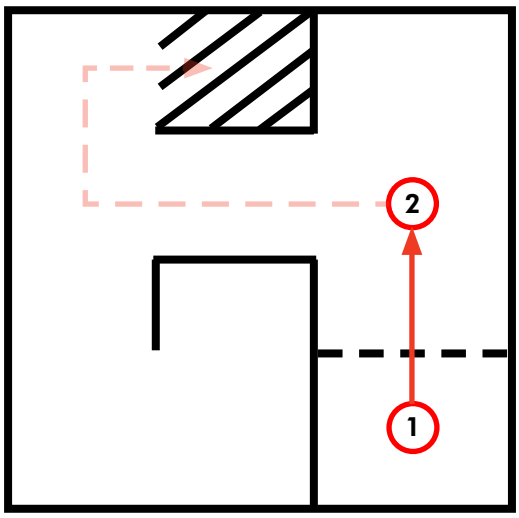
Thinking About Programming Planning & Behaviors

In this lesson, you will learn how thinking in terms of "behaviors" can help you to see the logic behind your robot's actions, and break a big plan down into practical parts.

"Behaviors" are a very convenient way to talk about what the robot is doing, and what it must do. Moving forward, stopping, turning, looking for an obstacle… these are all behaviors.



Complex Behavior
Some behaviors are big, like "solve the maze."



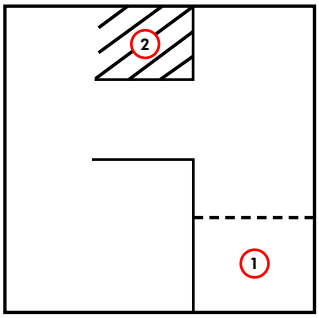
Basic or Simple Behavior
Some behaviors are small, like "go forward for 3 seconds." Big behaviors are actually made up of these smaller ones.

As you begin the task of programming, you should also begin thinking about the robot's actions in terms of behaviors. Recall that as programmer, your primary responsibilities are:

- **First**, to formulate a plan for the robot to reach the goal,
- And **then**, to translate that plan into a program that the robot can follow.

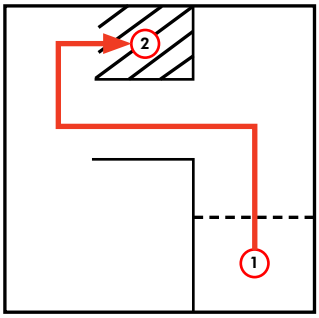
The plan will simply be *the sequence of behaviors that the robot needs to follow*, and the program will just be those behaviors translated into the programming language.

Thinking about Programming Planning & Behaviors (cont.)



1. Examine problem
To find a solution, start by examining the problem.

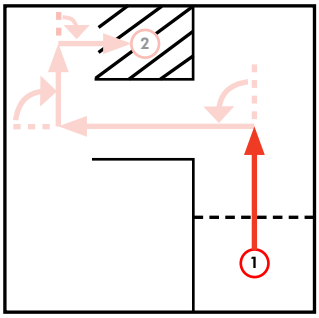
Here, the problem is to get from the starting point (1) to the goal (2).



2. Broad solution
Try to see what the robot needs to do, at a high level, to accomplish the goal.

Having the robot follow the path shown on the left, for example, would solve the problem.

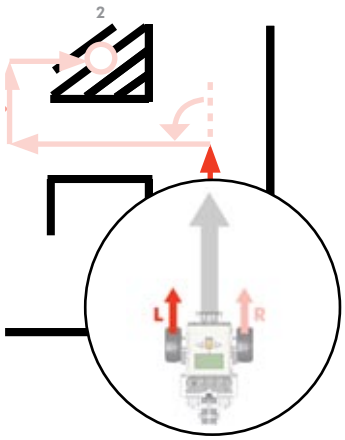
You've just identified the first behavior you need! Write it down.

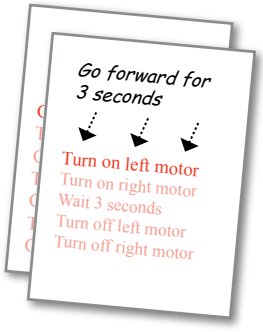


3. Break solution into smaller behaviors
Now, start trying to break that behavior down into smaller parts.

Following this path involves moving forward, then turning, then moving forward for a different distance, then turning the other way, and so on. Each of these smaller actions is also a behavior.

Write them down as well, taking care to keep them in the correct sequence.





4. Break into even smaller pieces
If you then break down these behaviors into even smaller pieces, you'll get smaller and smaller behaviors, with more and more detail. Keep track of them as you go.

Eventually, you'll reach commands that you can express directly in the programming language.

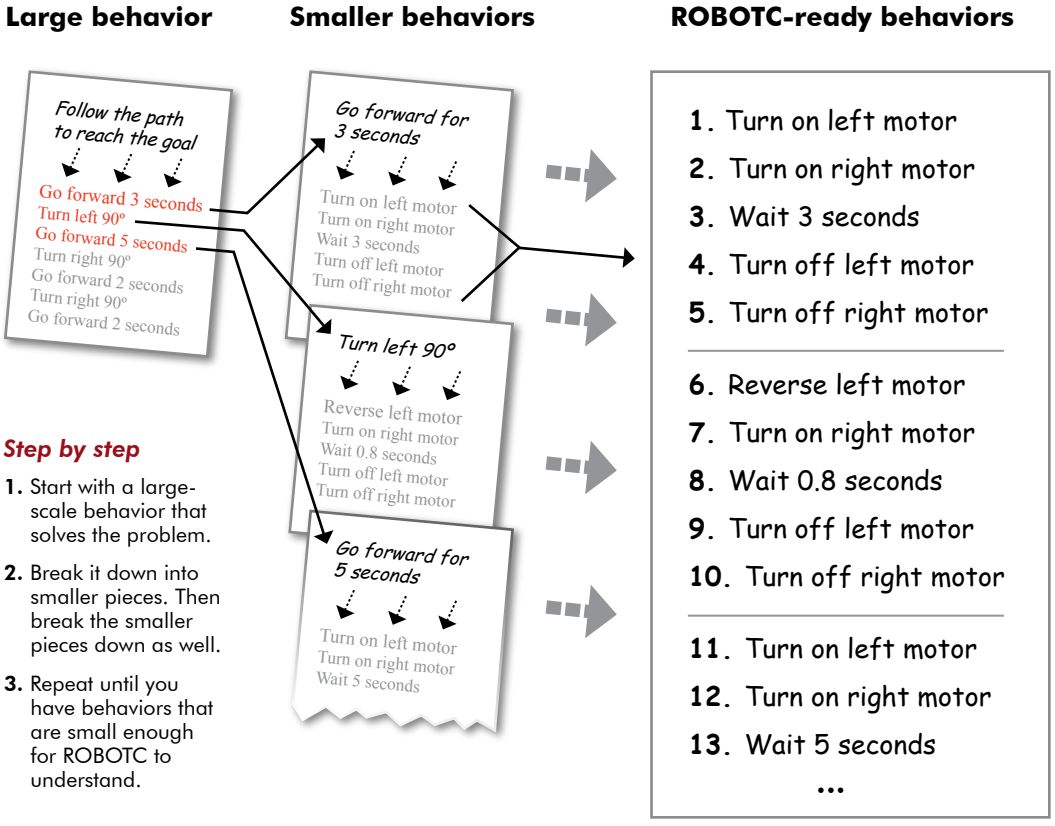
For example, ROBOTC has a command to turn on one motor. When you reach a behavior that says to turn on one motor, you can stop breaking it down, because it's now ready to translate.

Behaviors

Composition and Analysis

Perhaps the most important idea in behaviors is that they can be **built up or broken down into other behaviors**. Complex behaviors, like going through a maze, can always be broken down into **smaller, simpler behaviors**. These in turn can be broken down further and further until you reach simple or basic behaviors that you recognize and can program.

By looking back at the path of behaviors you broke down, you can also see how the smaller behaviors should be programmed so that they **combine back together**, and produce the larger behavior. In this way, analyzing a complex behavior **maps out the pieces** that need to be programmed, then allows you to **program them**, and **put them together** to build the final product.



Step by step

1. Start with a large-scale behavior that solves the problem.
2. Break it down into smaller pieces. Then break the smaller pieces down as well.
3. Repeat until you have behaviors that are small enough for ROBOTC to understand.

Sometimes it can be hard to tell whether a behavior is "simple" or "complex". Some programs are so complex they need multiple layers of simple behaviors before they reach the basic ones!

"Basic," "Simple," and "Complex" are categories of behaviors which are meant to help you **think about the structure of programs**. They are points of reference in the world of behaviors. Use these distinctions to help you, but don't worry if your "complex" behavior suddenly becomes a "simple" part of your next program... just pick the point of reference that's most useful for what you need.

Pseudocode & Flow Charts

Pseudocode is a **shorthand notation for programming** which uses a combination of **informal programming structures and verbal descriptions of code**. Emphasis is placed on expressing the behavior or outcome of each portion of code rather than on strictly correct syntax (it does still need to be reasonable, though).

In general, pseudocode is used to outline a program before translating it into proper syntax. This helps in the initial planning of a program, by creating the logical framework and sequence of the code. An additional benefit is that because pseudocode does not need to use a specific syntax, it can be translated into different programming languages and is therefore somewhat universal. It captures the **logic and flow of a solution** without the bulk of strict syntax rules.

Below is some pseudocode written for a program which moves as long as a touch sensor is not pressed, but stops and turns to the right if its sonar detects an object less than 20cm away.

```
task main()
{
 while ( touch sensor is not pressed )
  {
    Robot runs forward

    if (sonar detects object < 20 cm away)
    {
      Robot stops

      Robot turns right
    }
  }
}
```

Some intact syntax
The use of a while loop in the pseudocode is fitting because the way we read a while loop is very similar to the manner in which it is used in the program.

Descriptions
There are no actual motor commands in this section of the code, but the pseudocode suggests where the commands belong and what they need to accomplish.

This pseudocode example includes elements of both programming language, and the English language. Curly braces are used as a visual aid for where portions of code need to be placed when they are finally written out in full and proper syntax.

Pseudocode & Flow Charts

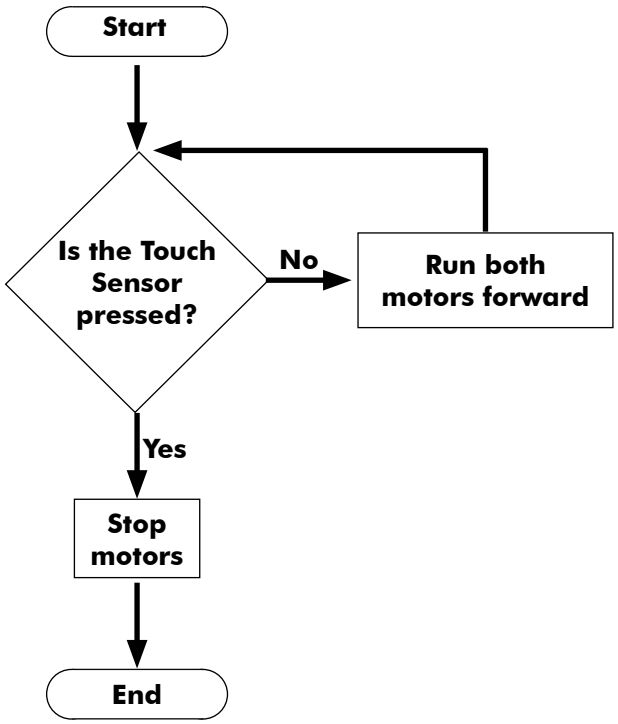
Flow Charts are a **visual representation of program flow**. A flow chart normally uses a combination of **blocks** and **arrows** to represent actions and sequence. Blocks typically represent **actions**. The **order** in which actions occur is shown using arrows that point from statement to statement. Sometimes a block will have multiple arrows coming out of it, representing a step where a **decision** must be made about which path to follow.

Start and End symbols are represented as rounded rectangles, usually containing the word "Start" or "End", but can be more specific such as "Power Robot Off" or "Stop All Motors".

Actions are represented as rectangles and act as basic commands. Examples: "wait1Msec(1000)"; "increment LineCount by 1"; or "motors full ahead".

Decision blocks are represented as diamonds. These typically contain Yes/No questions. Decision blocks have two or more arrows coming out of them, representing the different paths that can be followed, depending on the outcome of the decision. The arrows should always be labeled accordingly.

To the right is the flow chart of a program which instructs a robot to run forward as long as its touch sensor is not pressed. When the touch sensor is pressed the motors stop and the program ends.



To read the flow chart:

- Start at the "**Start**" block, and follow its arrow down to the "Decision" block.
- The **decision block** checks the status of the touch sensor against two possible outcomes: the touch sensor is either pressed or not pressed.
- If the touch sensor is not pressed, the program follows the **"No" arrow** to the action block on the right, which tells the motors to run forward. The arrow leading out of that block points back up and around, and ends back at the Decision block. This forms a **loop**!
- The **loop** may end up repeating many times, as long as the Touch Sensor remains unpressed.
- If the touch sensor is pressed, the program follows the "Yes" arrow and stops the motors, then ends the program.