cc3.12/cis1.0
computing: nature, power and limits—robotics applications
fall 2007
lecture # VI.1

solvability and feasibility

- conditional execution
- conditional repetition
- programmer-defined functions
- the halting problem

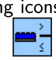
resources:

- reading: Reed chapters 7 and 10

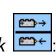
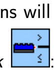
conditional execution

- there are times when you want your code to behave differently under different conditions
- for example, in the assignment for unit V:
  IF your robot sees something black, THEN it should stop for one second, then go backwards for two seconds, then go forward again.
  IF your robot sees something silver or gold, THEN it should stop for one second, then turn to the left and go forward again.
- the notion of *conditional execution* means that you define in your program multiple *branches*, and the code will follow a different branch depending on the conditions it encounters while running
- conditional execution is sometimes referred to as *IF-THEN* or *IF-THEN-ELSE* execution
- if the IF condition is true, then the THEN branch is executed;
  otherwise (if the IF condition is false), the ELSE branch is executed

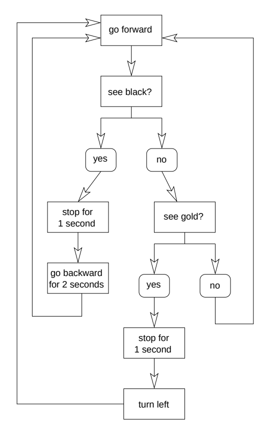
- in RoboLab, the following icons facilitate *conditional execution* in your robot:

touch-sensor-fork *light-sensor-fork* *fork-merge*

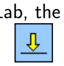
- note that these are different from *event-driven* icons since the program will NOT wait for an event to happen but will simply evaluate the condition of the *fork* icon and execute a branch accordingly
- for example, when using the *touch-sensor-fork* :
  IF the touch sensor is not pressed when the program comes to the *touch-sensor-fork* icon in its execution,
  THEN the top branch of icons will be executed;
  ELSE the bottom branch of icons will be executed
- when using the *light-sensor-fork* :
  IF the light sensor reads a value greater than the one specified (you have to hang a numeric constant below the icon containing the *threshold value* for the IF-THEN-ELSE decision),
  THEN the top branch of icons will be executed;
  ELSE the bottom branch of icons will be executed

- when writing a program that uses conditional execution, it is often easier to design your code first using a *flowchart*, before trying to write anything on the computer
- for example, here is a flowchart for the last challenge in the assignment for unit V:



conditional repetition

- there are times when you want your code to execute the same thing over and over again, repeatedly
- this is called *looping* or *iteration*
- we talked about three types of loops:
  - "forever" (or *infinite*) loops
  - *counter-controlled* loops
  - *condition-controlled* loops
- in RoboLab, the following icons facilitate infinite loops:



yellow land
goes BEFORE the code
that you want to repeat

yellow jump
goes AFTER the code
that you want to repeat

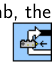
- in RoboLab, the following icons facilitate counter-controlled loops:

start-of-loop
goes BEFORE the code
that you want to repeat

end-of-loop
goes AFTER the code
that you want to repeat

- you have to hang a *loop counter* (numeric constant) from the *start-of-loop* icon indicating the number of times you want the loop to run

- in RoboLab, the following icons facilitate condition-controlled loops:



loop-touch-sensor-pushed

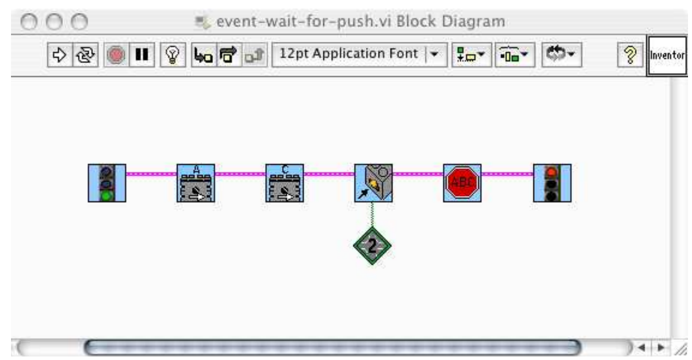
loop-touch-sensor-released

loop-light-sensor-less-than

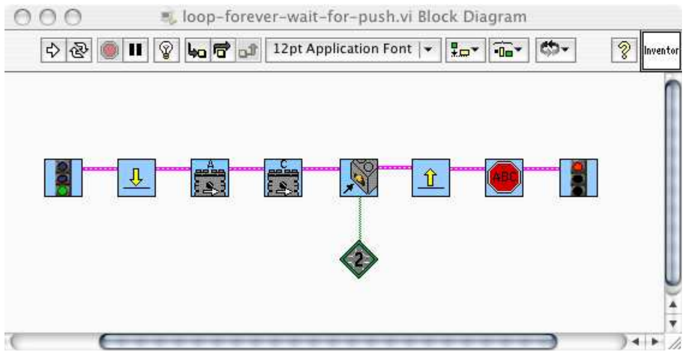
loop-light-sensor-greater-than

- for the light sensor loops, you have to hang a *loop counter* (numeric constant) from the "start of loop" icon indicating the number of times you want the loop to run
- for all the sensor-based loops, you have to hang the *port number* from the "start of loop" icon indicating which port the sensor is connected to
- for all the loops, the icons above show the "start of loop" icon; to end the loops, you use the *end-of-loop*  at the end of the loop

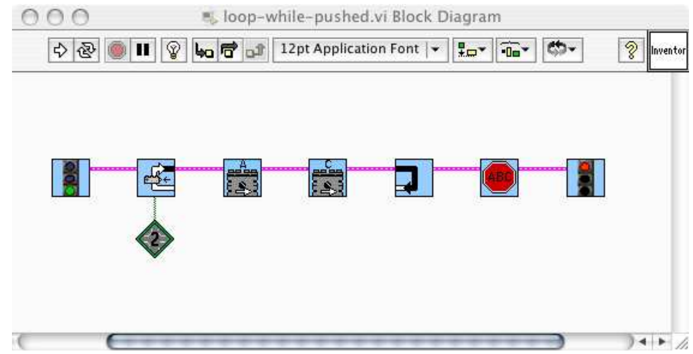
- compare the following programs:



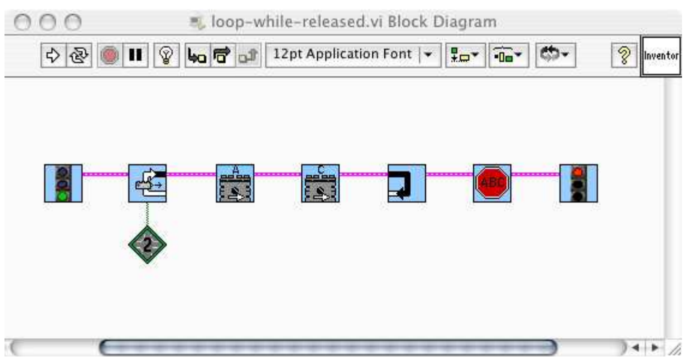
this program tells the robot to go forward until the touch sensor is pushed, after which it stops and the program ends



this program tells the robot to go forward until the touch sensor is pushed, after which it loops back and starts going forward again—forever; the robot never stops and the program never ends



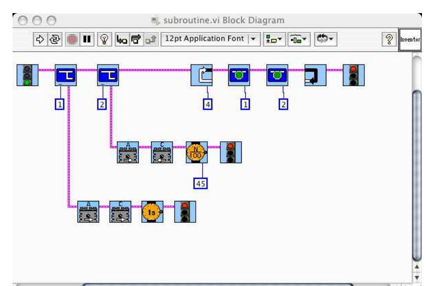
this program tells the robot to go forward as long as the touch sensor is pressed; it loops until the touch sensor is NOT pressed, i.e., until it is released; after which, the robot stops and the program ends



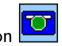
this program tells the robot to go forward as long as the touch sensor is released; it loops until the touch sensor is NOT released, i.e., until it is pushed; after which, the robot stops and the program ends

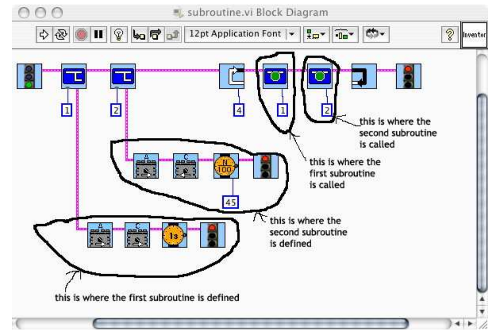
programmer-defined functions

- in RoboLab, "progammer-defined functions" are called *subroutines*
- the idea behind a *subroutine* is if you have some piece of code that is useful and you might want to use it many times—not just in a loop, but other times too—then you can group the icons together into something called a *subroutine*
- example:



- subroutines work by having two parts:
  - first, you have to *define* the subroutine
  - second, you have to *invoke* or *call* the subroutine
  - the subroutine only runs when you *call* it
  - it does NOT run when you *define* it
- the subroutine is defined with the *create-subroutine* icon 
- hanging from the *create-subroutine* icon is a numeric constant, assigning a number to the subroutine
- this is in case you want to define more than one subroutine—you give each a number so that you can distinguish between them later
- from the lower right corner of the *create-subroutine* icon, you string the icons that you want to belong to the subroutine
- you end the subroutine with the *end* icon 
- from the top right corner of the *create-subroutine* icon, you continue with your program code

- when you want to *call* or *invoke* the subroutine, then you use the *run-subroutine* icon 
- here's the example again:



the halting problem

- a *loop* is a set of instructions that repeats several times
- we talked about 3 kinds of loops in RoboLab (counter-controlled, condition-controlled, forever) These concepts are the same in any computer programming language!
- here is an example in computer *pseudo-code*:

```
x=0;
do 3 times {
  add 1 to x
}
```

How many times does this loop execute?
What is the value of x when this code completes?

- another example:

```
x=3;
while ( x > 0 ) {
  subtract 1 from x;
}
```

How many times does this loop execute?
What is the value of x when this completes?

- and another example:

```
x=1;
while ( x < 5 ) {
  y = x;
}
```

How many times does this loop execute?
What is the value of x when this completes?

- a program containing an *infinite loop* will run *forever*—it will never HALT or TERMINATE
- this is called the HALTING PROBLEM in computer science—being able to look at a computer program and determine if it will ever halt (stop).
- sometimes, whether a program stops or not depends on *input* that it receives (like your robot receiving sensor input)
- here is an example:

```
myProgram( input: x ) {
  while ( x > 0 ) {
    add 1 to x;
  }
}
```

  How many times does this loop execute if x = 0?
  How many times does this loop execute if x = 1?

computability

- a problem is *computable* if it is possible to write a computer program that can solve it.
- a *non-computable* problem is also called *non-solvable*.
- the *halting problem* is not computable!
  i.e., can we write a computer program that will determine if any computer program and its input will halt?
  how would you answer this question?
  - could you try running the program? (what if it never halted?)
  - suppose we wrote a program ("A") that would take two inputs:
    another program ("P") and the input ("X") for the other program
    "A" works like this:
    if "P(X)" halts, then "A(P,X)" should run forever
    if "P(X)" does not halt, then "A(P,X)" should halt
  - the *paradox* is: what if we call program "A" on itself?, i.e., A(A,X)
    the program cannot produce an answer!

- this is called *proof by contradiction*—we assume that a program does exist that can solve the halting problem; then we show that it cannot possibly exist.
- computability in general is an important question
- it was considered by concerned mathematicians even before digital computers were developed!
- in the 1930's, much work was devoted to this.
- the Church-Turing thesis (1940's) states basically that any computation that can be defined in an algorithm can be processed on a computer
- named after Alonzo Church and Alan Turing (really famous guy)

feasibility

- even if a problem is *computable*, it is not always *feasible* to write a program to compute it because sometimes it takes too long to solve a problem
- we talked about this with robot soccer:
  a robot might be able to find a ball using a complicated algorithm, but if it takes longer than 20 minutes to find the ball, the soccer game will be over! (extreme example...)
- could you write a program that could count the number of atoms in the universe (estimated at about 10^{80})?
  suppose it took 1 second to count one atom.
  how many seconds would the program need to run to count all of them?
  here's another way to look at it:
  how many atoms could the program count in a year?
  $num\_atoms\_per\_year = 60sec/min \times 60min/hour \times 24hours/day \times 365days/year$
  $= 31,536,000sec$
  $= 3x10^7$
  how does that compare to 10^{80}??