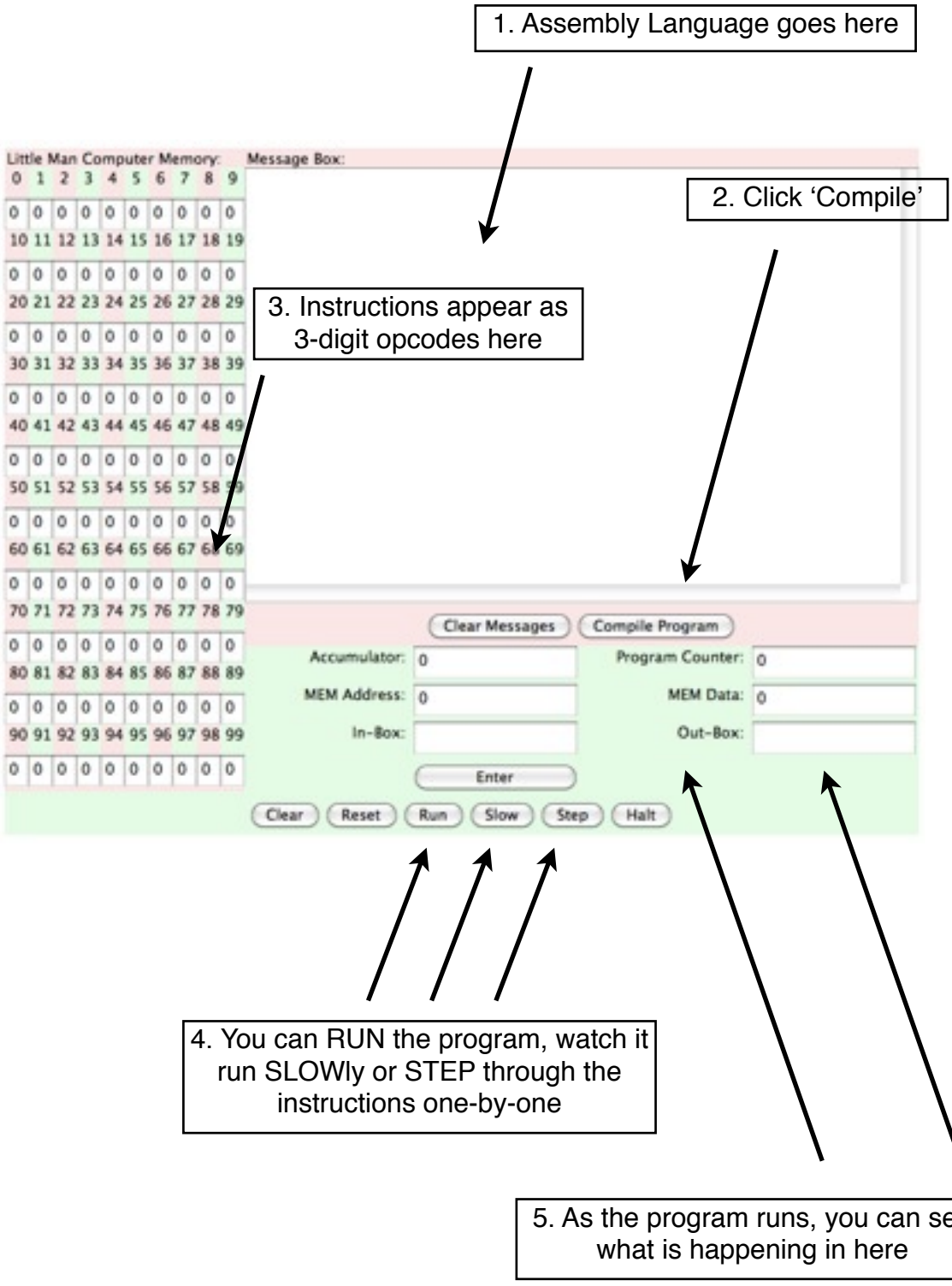
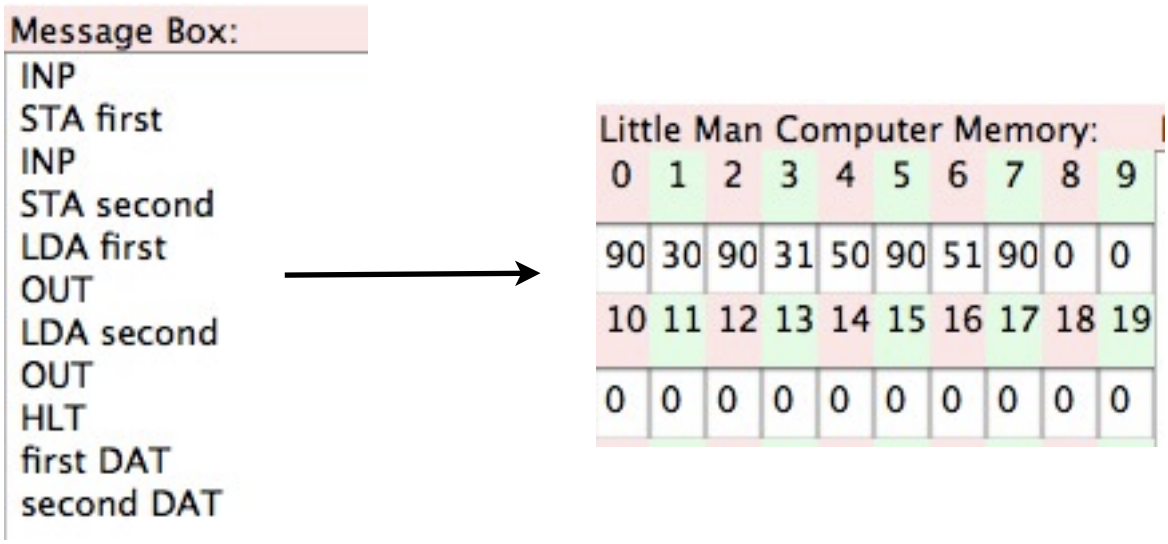
The Little Man Computer - Interface



Little Man Computer - Input, Storage and Output

Type in the following mnemonics, click compile and the following opcodes should appear in the memory:



Line By Line

```
INP             <-- Prompt the user for an input [stored in accumulator temporarily]
STA first       <-- Store the answer [currently in accumulator] in a variable called first
INP             <-- Prompt the user for an input [stored in accumulator temporarily]
STA second      <-- Store the answer [currently in accumulator] in second
LDA first       <-- Load the number in variable first back into the accumulator
OUT             <-- Put the contents of the accumulator into the out-box
LDA second      <-- Load the number in variable second back into the accumulator
OUT             <-- Put the contents of the accumulator into the out-box
HLT             <-- End of program
first DAT       <-- Declare that first is data [i.e. a variable]
second DAT      <-- Declare that second is data [i.e. a variable]
```

The LMC compiler (technically an assembler) converts each *mnemonic* into an *opcode*.

```
INP             <-- 901 [Input]
STA first       <-- 309 [Store in memory address 09]
INP             <-- 901 [Input]
STA second      <-- 310 [Store in memory address 10]
LDA first       <-- 509 [Load the data from memory address 09]
OUT             <-- 902 [Output]
LDA second      <-- 510 [Load the data from memory address 10]
OUT             <-- 902 [Output]
HLT             <-- 0 [End of program]
first DAT       <-- [Declare first as a variable]
second DAT      <-- [Declare second as a variable]
```

Points to Note

- The LMC is not case sensitive. It is a good idea to write mnemonics and variables using different cases to make it easier to read, but the compiler does not notice the difference.

- Variables are declared at the *end* of the program, rather than the start. This is so the compiler can assign memory addresses immediately after the program instructions - it doesn't know which addresses are free initially.

- If there is an error in the program then the compiler will fail and the code will be lost. It is therefore a good idea to *copy (CTRL-C)* the program before you compile. This is especially true with more complex programs.

- Each mnemonic is converted to one opcode - there is a 1:1 relationship. This is one of the major differences between an assembler and a compiler.

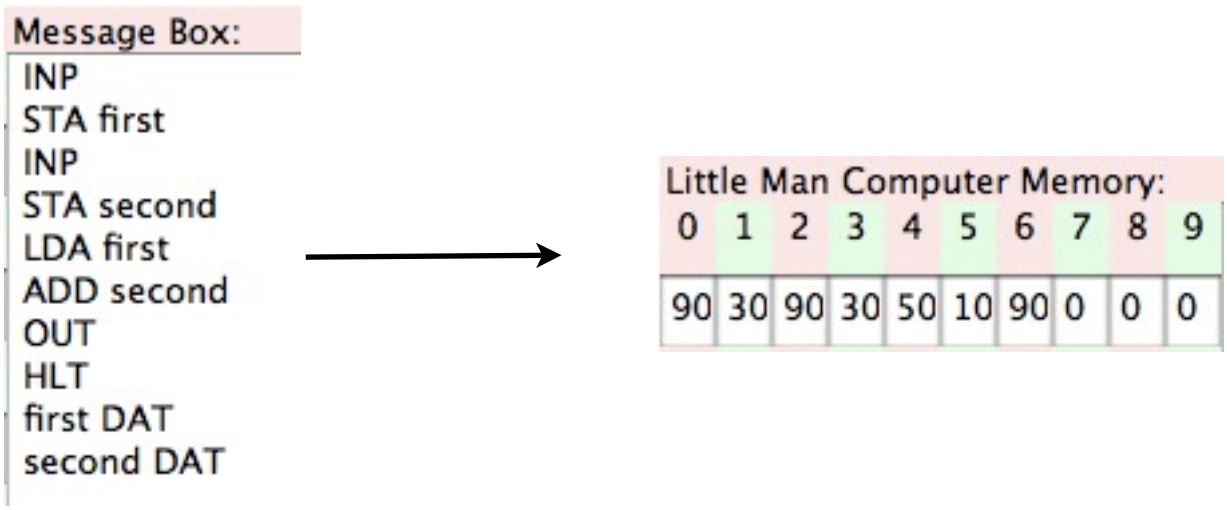
Step Through

By clicking the *step* button the program will execute one line at a time. You can watch the following boxes to see how the registers inside the processor work.

`Accumulator`	<-- Temporarily stores the most current piece of data
`Program Counter`	<-- Keeps track of which instruction to carry out next
`MEM Address`	<-- Points to the currently addressed memory block
`MEM Data`	<-- Loads the contents of the addressed memory
`In-Box`	<-- Used for data input
`Out-Box`	<-- Used for data output

Little Man Computer - Addition and Subtraction

To add or subtract, load a number into the accumulator and add or subtract number from memory:



Line By Line

```
INP             <-- Prompt the user for an input [stored in accumulator temporarily]
STA first       <-- Store the answer [currently in accumulator] in a variable called first
INP             <-- Prompt the user for an input [stored in accumulator temporarily]
STA second      <-- Store the answer [currently in accumulator] in second
LDA first       <-- Load the number in variable first back into the accumulator
ADD second      <-- Add the contents of second to whatever is in the accumulator
OUT             <-- Put the contents of the accumulator into the out-box
HLT             <-- End of program
first DAT       <-- Declare that first is data [i.e. a variable]
second DAT      <-- Declare that second is data [i.e. a variable]
```

The LMC compiler (technically an assembler) converts each *mnemonic* into an *opcode*.

```
INP             <-- 901 [Input]
STA first       <-- 308 [Store in memory address 09]
INP             <-- 901 [Input]
STA second      <-- 309 [Store in memory address 10]
LDA first       <-- 508 [Load the data from memory address 09]
ADD second      <-- 109 [Load the data from memory address 10]
OUT             <-- 902 [Output]
HLT             <-- 0 [End of program]
first DAT       <-- [Declare first as a variable]
second DAT      <-- [Declare second as a variable]
```

Points to Note

- The *accumulator* (think: short term memory) stores the last number the computer was dealing with. The ADD command loads the data into the *MEM Data* register to be added to the *accumulator*.

Subtraction

Line By Line

```
INP             <-- Prompt the user for an input [stored in accumulator temporarily]
STA first       <-- Store the answer [currently in accumulator] in a variable called first
INP             <-- Prompt the user for an input [stored in accumulator temporarily]
STA second      <-- Store the answer [currently in accumulator] in second
LDA first       <-- Load the number in variable first back into the accumulator
SUB second      <-- Subtract the contents of second from what is in the accumulator
OUT             <-- Put the contents of the accumulator into the out-box
HLT             <-- End of program
first DAT       <-- Declare that first is data [i.e. a variable]
second DAT      <-- Declare that second is data [i.e. a variable]
```

The LMC compiler (technically an assembler) converts each *mnemonic* into an *opcode*.

```
INP             <-- 901 [Input]
STA first       <-- 308 [Store in memory address 09]
INP             <-- 901 [Input]
STA second      <-- 309 [Store in memory address 10]
LDA first       <-- 508 [Load the data from memory address 09]
SUB second      <-- 209 [Load the data from memory address 10]
OUT             <-- 902 [Output]
HLT             <-- 0 [End of program]
first DAT       <-- [Declare first as a variable]
second DAT      <-- [Declare second as a variable]
```

Points to Note

- It is important to get the subtraction the right way round. This sometimes requires a good bit of thought! A good general rule is to always store each number in a variable - just in case you need it later.

- The LMC can cope with negative numbers as well as positive numbers.

Little Man Computer - Beginner Tasks

Write LMC programs to complete the following tasks. The only commands you will need are INP, OUT, STA, LDA, ADD, SUB and HLT. For each task you should submit annotated mnemonics:

Task 1:

Ask the user for three numbers and then repeat them in reverse order.

Test data:	Expected output:
7, 8, 9	9, 8, 7
16, 12, 5	5, 12, 16

Task 2:

Ask the user for three numbers, add them and print out the answer.

Test data:	Expected output:
7, 8, 9	24
-2, 1, -3	-4

Task 3:

Ask the user for one number, double it and print out the answer.

Test data:	Expected output:
7	14
-3	-6

Task 4:

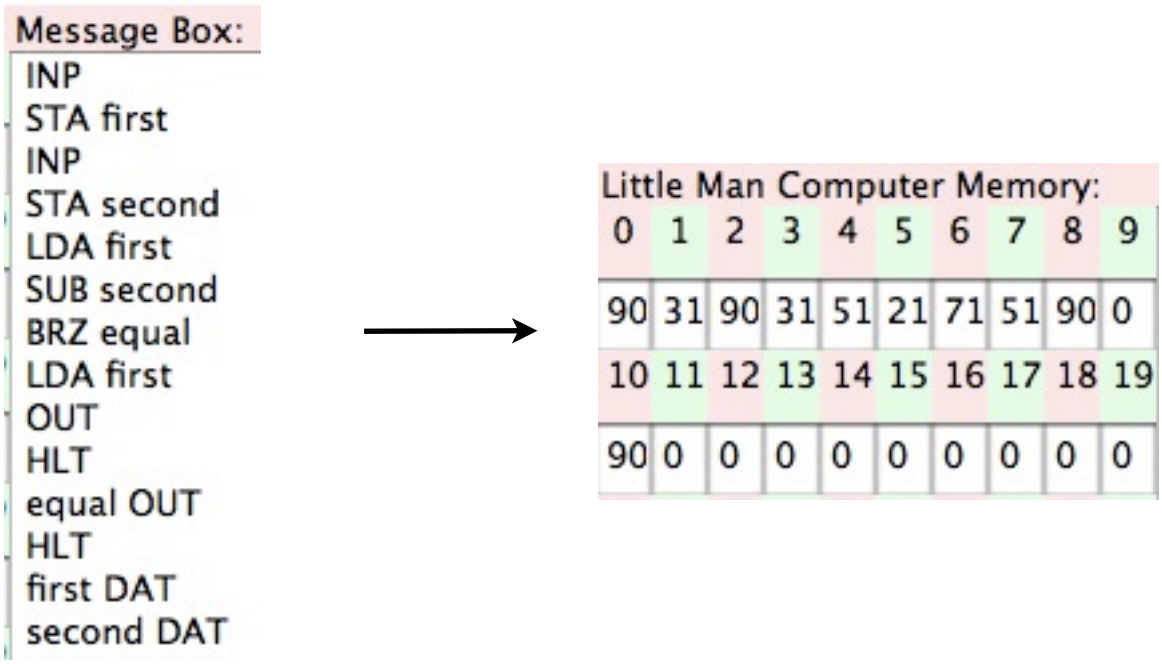
Ask the user for two numbers. Print out the answer to first number minus the second number, followed by the answer to the second number minus the first number.

Test data:	Expected output:
7, 3	4, -4
12, 5	7, -7

Little Man Computer - IF Statements

IF statements work by having a logical test (e.g. is 7 > 3) and then sending the program down one of two routes. In the LMC we use the *branch* command. There are three *branch* commands - Branch If Zero (BRZ), Branch If Zero Or Positive (BRP) and Branch Always (BRA).

In this program we want to find if two numbers are equal. If they are then subtracting them will give the answer 0:



Line By Line

```
INP             <-- Prompt the user for an input [stored in accumulator temporarily]
STA first       <-- Store the answer [currently in accumulator] in a variable called first
INP             <-- Prompt the user for an input [stored in accumulator temporarily]
STA second      <-- Store the answer [currently in accumulator] in second
LDA first       <-- Load the number in variable first back into the accumulator
SUB second      <-- Subtract second from the accumulator
BRZ equal       <-- If the answer is 0 then continue the program from the label equal
LDA first       <-- Load the number in variable first back into the accumulator
OUT             <-- Output the contents of the accumulator [first ]
HLT             <-- End of program
equal OUT       <-- This is equal. Output the contents of the accumulator [0]
HLT             <-- End of program
first DAT       <-- Declare that first is data [i.e. a variable]
second DAT      <-- Declare that second is data [i.e. a variable]
```

The LMC compiler (technically an assembler) converts each *mnemonic* into an *opcode*.

```
INP             <-- 901 [Input]
STA first       <-- 312 [Store in memory address 12]
INP             <-- 901 [Input]
STA second      <-- 313 [Store in memory address 13]
LDA first       <-- 512 [Load the data from memory address 12]
SUB second      <-- 213 [Subtract the number from memory address 13]
BRZ equal       <-- 710 [If the answer is 0, go to instruction 10, otherwise carry on]
LDA first       <-- 512 [Load the data from memory address 12]
OUT             <-- 902 [Output]
HLT             <-- 0 [End of program]
equal OUT       <-- 902 [Output]
HLT             <-- 0 [End of program]
first DAT       <-- [Declare first as a variable]
second DAT      <-- [Declare second as a variable]
```

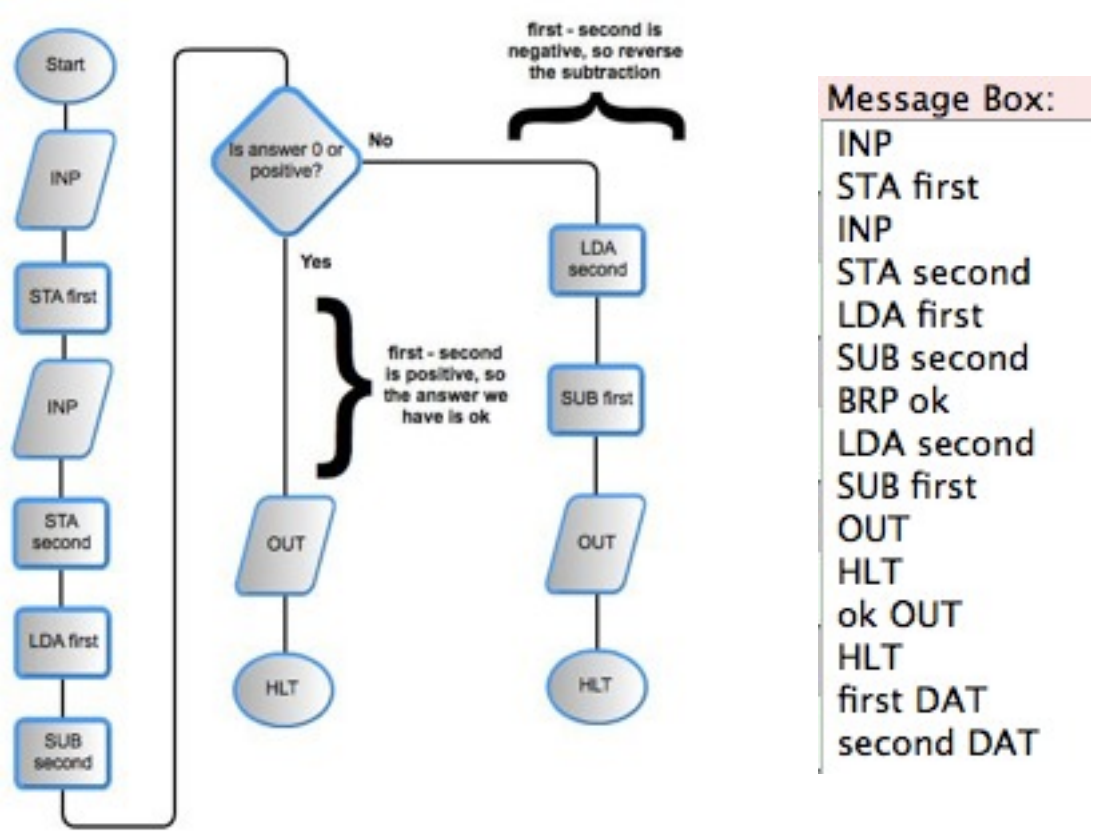
Points to Note

- The word *equal* is used a label, to point where the branch will go.
- In the mnemonics, the label is used twice - in both the *branch* command and also as a label. In the opcodes it is only used once, in the *branch* command. By referring to a specific numbered instruction there is no need to use the label again.
- Both branches require a HLT command.

Little Man Computer - More IF Statements

For this program we want to get two numbers and work out the difference (i.e. the bigger number - the smaller number). But we don't know which is which.

Because this is a bit more complicated, we'll start by planning the program with a flowchart:



Line By Line

```
INP             <-- Prompt the user for an input [stored in accumulator temporarily]
STA first       <-- Store the answer [currently in accumulator] in a variable called first
INP             <-- Prompt the user for an input [stored in accumulator temporarily]
STA second      <-- Store the answer [currently in accumulator] in second
LDA first       <-- Load the number in variable first back into the accumulator
SUB second      <-- Subtract second from the accumulator
BRP ok          <-- If the answer is 0 or positive then the answer is fine - go to ok
LDA second      <-- Load the number in variable second back into the accumulator
SUB first       <-- Subtract first from the accumulator
OUT             <-- Output the contents of the accumulator
HLT             <-- End of program
ok OUT          <-- This is ok. Output the contents of the accumulator
HLT             <-- End of program
first DAT       <-- Declare that first is data [i.e. a variable]
second DAT      <-- Declare that second is data [i.e. a variable]
```

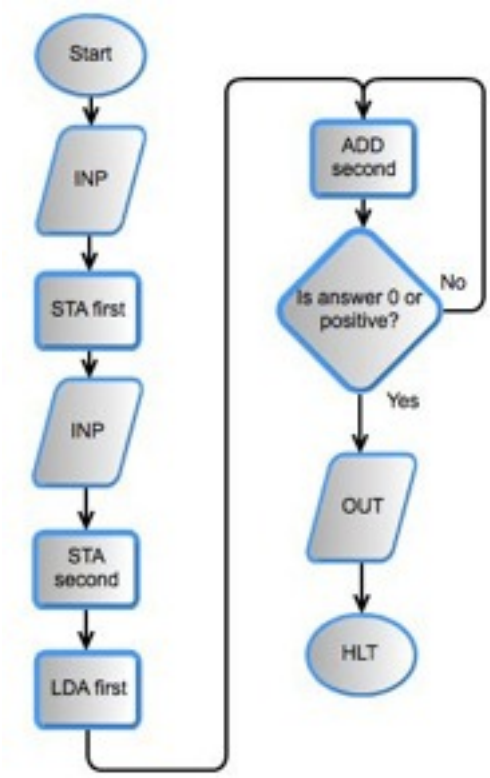
Points to Note

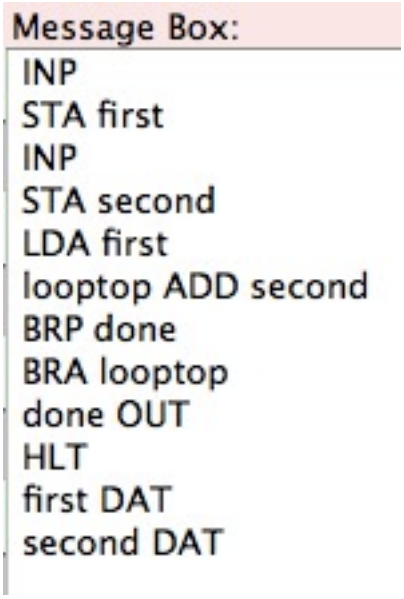
- The word *ok* is used a label, to point where the branch will go.
- It is important to follow the logic. If first - second is 0 or positive then the answer is ok, and the branch that goes to the *ok* label should just print out the answer. If it **isn't** ok then the instructions immediately after the *branch* command should deal with that.
- It is possible to use multiple *branch* statements to create multiple IF/ELSE statements or CASE statements

Little Man Computer - Loops

By combining a BRA (break always) with a BRP or BRZ you can create a loop.

In this program we will take a negative number (e.g. -7) and keep adding a second number (e.g. 2) until it gets to 0 or a positive number:





Line By Line

```
INP             <-- Prompt the user for an input [stored in accumulator temporarily]
STA first       <-- Store the answer [currently in accumulator] in a variable called first
INP             <-- Prompt the user for an input [stored in accumulator temporarily]
STA second      <-- Store the answer [currently in accumulator] in second
LDA first       <-- Load the number in variable first back into the accumulator
looptop ADD second      <-- This is the top of the loop. Add the number in second
BRP done        <-- If the answer is 0 or positive then the answer is fine - go to done
BRA looptop     <-- If the answer is no then go back to looptop
done OUT        <-- This is what happens once we're done. Output the accumulator
HLT             <-- End of program
first DAT       <-- Declare that first is data [i.e. a variable]
second DAT      <-- Declare that second is data [i.e. a variable]
```

Little Man Computer - Intermediate Tasks

Write LMC programs to complete the following tasks. You will need to use the BRZ, BRP and BRA commands here as well. For each task you should submit annotated mnemonics:

Task 1:

Ask the user for two numbers and print out the biggest.

Test data:	Expected output:
7, 9	9
9, 7	9

Task 2:

Ask the user for two numbers. If they are the same then print 0, otherwise add them and print out the answer.

Test data:	Expected output:
3, 9	12
7, 7	0

Task 3:

Ask the user for one larger number and one factor of that number (e.g. 15 and 3, 20 and 5). The program should keep subtracting the smaller number until it reaches 0 - where it should print out 0.

Test data:	Expected output:
16, 4	0
36, 6	0

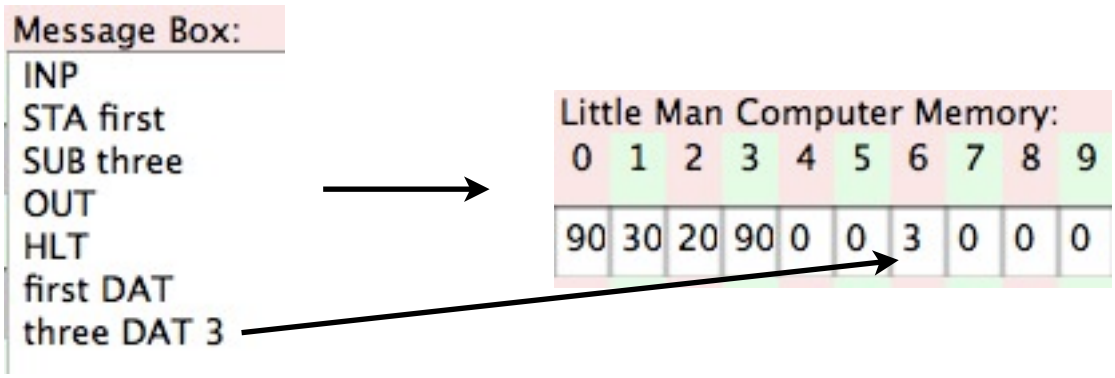
Task 4:

Ask the user for three numbers and print out all three from largest to smallest.

Test data:	Expected output:
3, 16, 5	16, 5, 3
4, -3, 20	20, 4, -3

Little Man Computer - Constants / Initialising Variables

When declaring variables, you can also initialise them with a value:



Line By Line

```
INP             <-- Prompt the user for an input [stored in accumulator temporarily]
STA first       <-- Store the answer [currently in accumulator] in a variable called first
SUB three       <-- Subtract the value of variable three
OUT             <-- Output the accumulator
HLT             <-- End of program
first DAT       <-- Declare that first is data [i.e. a variable]
three DAT 3     <-- Declare that third is data [i.e. a variable] of value 3
```

Points to Note

- Initialising takes place at the same time as declaration.
- Assigning a value to a variable at the start is a good way to make a constant (a number that doesn't change) - but it doesn't have to be a constant. You can still change the number in the program.
- Creating a variable called *one*, with value 1, is a good way to create a counter that goes up or down by one each time you run through a loop.