The Imp Computer

Age group:	10 – adult
Abilities assumed:	Very simple programming
Time:	30-40 minutes
Size of group:	6 upwards

Focus

If statements
Flow of control
Programming

Syllabus Links

This activity is appropriate for any syllabus aim about learning to program that requires an understanding of flow of control.

Summary

You compile simple programs that involve if statements on to a computer made of students roped together. Each student represents an instruction. A baton represents the flow of control. It is passed to the first student who carries out their instruction before passing it on. When it is returned the program has been executed and the appropriate result has appeared on the screen. This makes the program execution both visible and tangible allowing a variety of concepts to be explained and discussed.

Technical Terms

If statement, sequencing, flow of control, program counter, run-time, compile-time.

Materials

Short card or plastic tube
    To act as a baton
Brightly coloured rope (e.g., plastic washing line)
    Cut in to at least 8 meter long pieces
Instruction cards
    Print off the sheets and laminate
A4 box
    To act as a variable
Blank A4 paper
Board or flip chart and pens
    To be the computer screen.
The program
    Program slides for relevant language,
        Alternatively write programs up on board or give as a handout

What to do

The Grab

In the Discworld Science Fantasy books by Terry Pratchett (a must read for any budding computer scientist), technology that isn't magical tends to be run by Imps and other small creatures. For example, the Discworld camera. is a box with a fast painting imp sitting inside with easil. Take too many pictures of sunsets and the imp will run out of pink and start complaining! The book 'Going Postal', is about the Discworld invention of the 'clacks', a shutter semaphore system. Reading the book is actually a great way to learn about how the protocols underlying communication networks work

Explain that in this activity you will use a similar idea to create an 'Imp' computer that can be used to run any program written in any programming language. You are going to compile a program on to the class where you use students rather than imps.

In fact this is exactly what computers started out like. The original OED definition of a computer is: *A person who makes calculations; spec. a person employed for this in an observatory, etc*. Furthermore the earliest computers at Bletchley Park used to crack the German codes in the war were huts full of women.

The activity

Compile time

Display the 'Insulting' program below. Here we use Python syntax, but it can be done with a program written in any language and even in pseudocode as appropriate to the lesson.

```
answer = input ("Can I insult you?")

if answer == "Y":
      print ("You smell!")
else:
      print ("You smell of roses!")

print ("Thank you!")
```

Explain that before you can run a program you have to compile it – translate it into a form that can be executed. Normally programs are compiled into machine code to run on a machine made of silicon, you will compile it onto an imp computer made out of students – which is similar but more squishy.

Compile the program a line at a time, explaining what you are doing at each step.

Instruction 1: The first instruction is an input statement incorporating an assignment. Get a volunteer to the front and give them the 'Instruction 1' card. Explain that their job (when they hold the baton) is to get a message "Can I insult you?", written onto the board, wait for a response and then store the response. Get them to choose another volunteer and give that person a box labelled **answer** to store the response in. They are the variable. [With fewer people available the variable box could just sit separately.] Explain you also need the part of the operating system that controls the

keyboard and screen. Get another volunteer to stand by the board ready to write "to the screen". The first volunteer's job is to tell them to write the sequence of characters C-a-n-SPACE-I etc to the screen, then move the cursor to a new line. [Note the SPACE character is just a space]. They will then have to wait for the rest of the class, acting as the keyboard, to instruct the screen person to write their response, then put that response in to the box. Point out that the program is not being executed yet – we are not following the instructions, just setting them up.

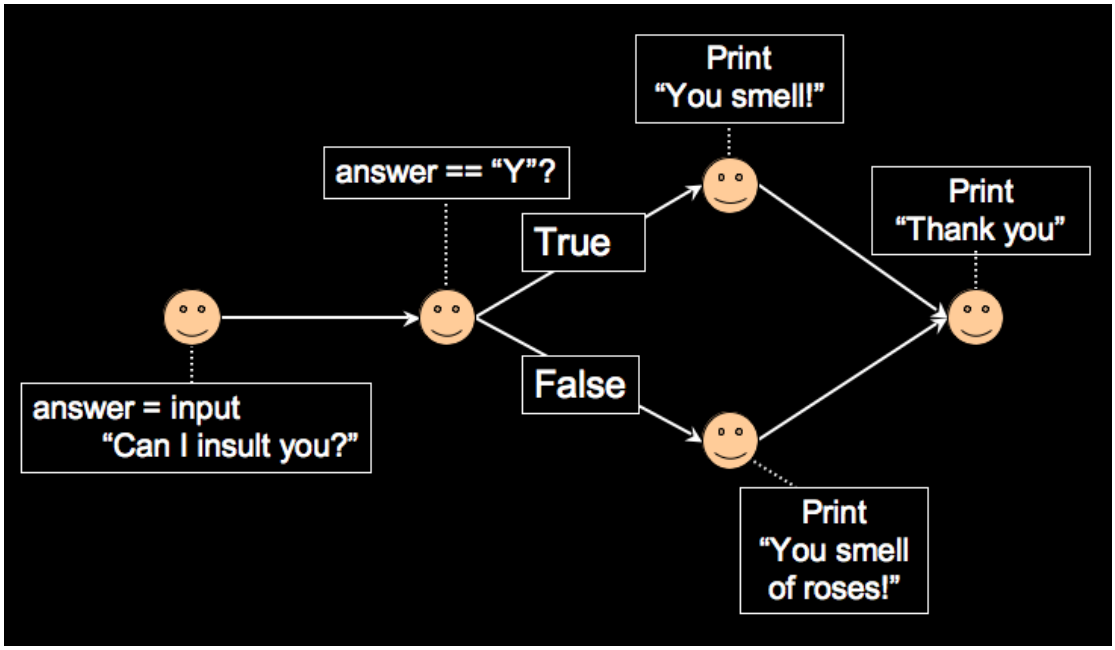
Instruction 2: The next instruction is an if-statement and the first part is the test. Get another volunteer and give them a piece of rope with the other end held by the first instruction volunteer. Explain that the baton can only pass along the rope. They are the test. Their main job is to answer a true/false question. Give them their instruction and explain it. When they get the baton they must find out what is in the box called 'answer'. If what is there is a capital Y then they should pass the baton left. If it is anything else they should pass the baton right. Give them two pieces of rope, one in their left hand and one in their right and pass the other ends to two new volunteers.

Instruction 3: The first of those volunteers is a print statement. Their job is to tell the screen to write up the characters Y-o-u-SPACE-s-m-e-l-l-! They then immediately pass on the baton.

Instruction 4: The second of the new volunteers is also a print statement. Their job is to tell the screen to write up the characters Y-o-u-SPACE-s-m-e-l-l-SPACE-o-f-SPACE-r-o-s-e-s-! They then also immediately pass on the baton.

Instruction 5: The final volunteer is yet another print statement. Their job is to tell the screen to write up the characters T-h-a-n-k-SPACE-y-o-u-! They then also immediately pass on the baton. In their case there is no one to pass the baton to, so they pass it back to you (the operating system). Point out the shape of the structure created from the rope links and how the if statement splits so there are two paths but they meet up to carry on with the rest of the program.

You should have a roped out structure that looks something like the following (you may wish to display this:



Run time
Having compiled the program, we now run it. Explain you are the operating system and you control what programs run and when. When a command is typed, or a button or icon clicked requesting the program to be run, you control the process.

Make it clear at each step which instruction in the original program is being executed.

Instruction 1: Take the baton and pass it to the first volunteer. [You could point out that the baton is acting like a program counter if the class has covered that in e.g., an architecture or machine code lesson.] Talk them through their instructions. Making sure the right thing is put on the board. Get someone in the class to decide what the response is and have that written on the board as it would be if the program was run. Make sure they include pressing return – or the program will wait forever! Now get the Instruction 1 volunteer to copy what was entered on to a piece of paper and put it in to the box labelled 'answer'. They then pass on the baton to the person they are roped to.

Instruction 2: Get the next person to follow their instruction. They look in the box and make a decision. They have to decide whether it is true that the thing in the box is a capital 'Y', then pass the baton the appropriate way according to their instructions.

Instruction 3 and 4: Whichever way it is passed, that person follows their instruction. The other does nothing. A sequence of characters get put on the screen either way. Make sure they get the capitalisation right pointing out that the instructions have to be followed exactly. It should also be printed on a new line.

Instruction 5: Either way the baton is then passed to the final instruction/volunteer. They get their message printed up on the board and take the baton back. The program has terminated.

At this point you could run the program again having the response to the question such that it leads to the other branch being executed.

While the structure is still there ask for questions about anything anyone is unsure of. If no one asks, prompt with questions like what happens if a lower case y is typed in rather than an uppercase Y, then try it out, working through the steps and emphasising what is in the box is not a capital Y – the test is false.

The explanation

We have seen that an if-statement compiles into a structure that splits in two but then joins up again. Even though on the page the instructions are written one after another, the if statement gives them structure – either one branch or the other will be executed on any execution. Either way the branches join back together, so the program then carries on doing the same thing after the if statement. The test is always a true/false question. It has to decide to pass the baton one way or another. Detail like capitalisation matter both in answering the question and in print statements.

In addition to being used to explain the basic concept of an if-statement and how it works, this can be used to explain other concepts related to compilation. We have compiled the program on to people – an Imp computer – but the process is essentially the same as actually happens. We first convert the program as written into an executable form – the program itself written in a programming language isn't

executable itself it is in a form that humans can more easily read and write. Compile time is when we create that executable version (and check we do have an actual program and not some gobbledygook or a program with spelling or other syntax mistakes that make it unclear what program was meant). Run time happens later and is when the instructions are followed. Once we have compiled the program, we run it. As long as we don't delete the compiled version, we can run it over and over without recompiling it. You can also discuss what happens if we change the original program but do not recompile.

You can also point out that each instruction in the program can lead to multiple smaller instructions being followed. We have converted a high level program into a lower level one!

Variations and Extensions

Drawing Imp computers
Having demonstrated the live version of an Imp computer, students can compile other small programs into their own drawings of Imp computers (essentially just a flow chart) and use a coin as the baton to execute the paper version. Having seen the live version it will be much clearer how this relates to the program and bring it much more to life.

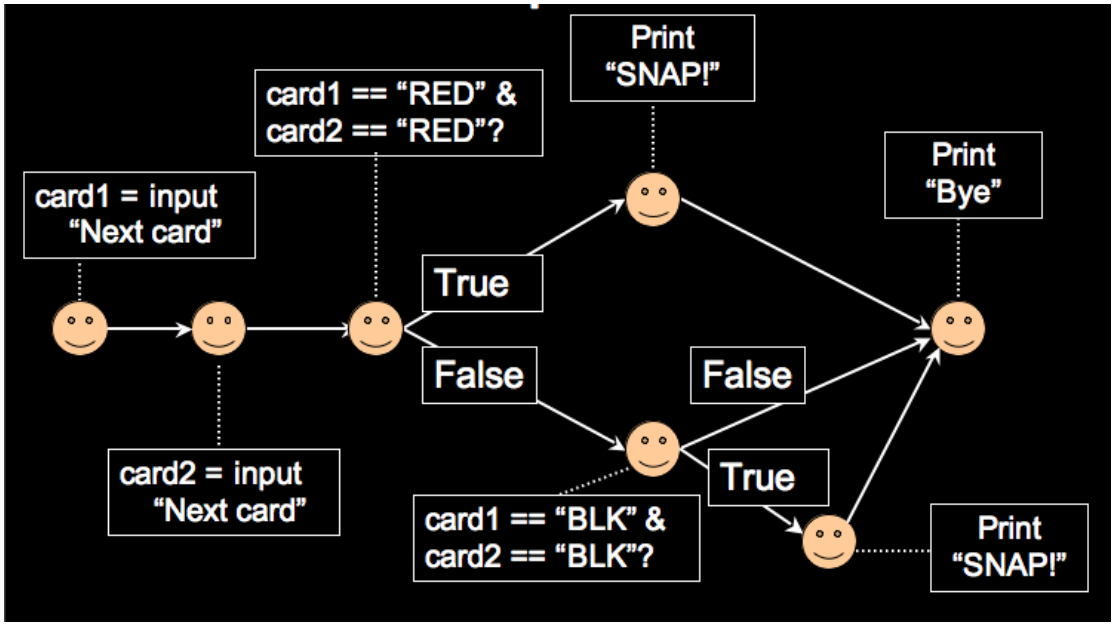
More complex programs: Snap
The same approach (either live or drawn) can be used on more complex programs. For example, the following program that plays RED-BLACK Snap (2 cards of the same colour mean snap). It can be used to illustrate nested if-statements

```
card1 = input ("Next card")
card2 = input ("Next card")

if card1 == "RED" and card2 == "RED":
            print ("SNAP!")
else:
           if card1 == "BLACK" and card2 == "BLACK":
                print ("SNAP!")

Print ("Bye")
```

It compiles to the following Imp computer:

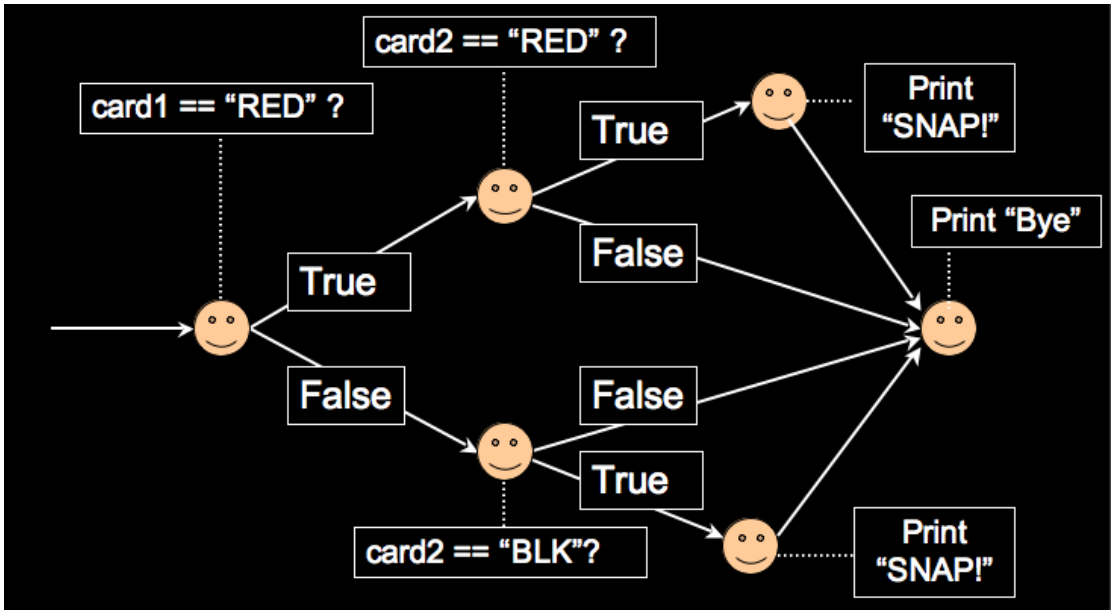


Program variations

It is also possible to explore different programs that do the same thing. For example, the following program does the same thing as the above Snap program, but because it is coded differently it leads to a different structure of program. The two can be compared. (We omit the initial input instructions below):

```
if card1 == "RED":
    if card2 == "RED":
        print ("SNAP!")
else:
     if card2 == BLACK:
         print ("SNAP!")
print ("Bye")
```

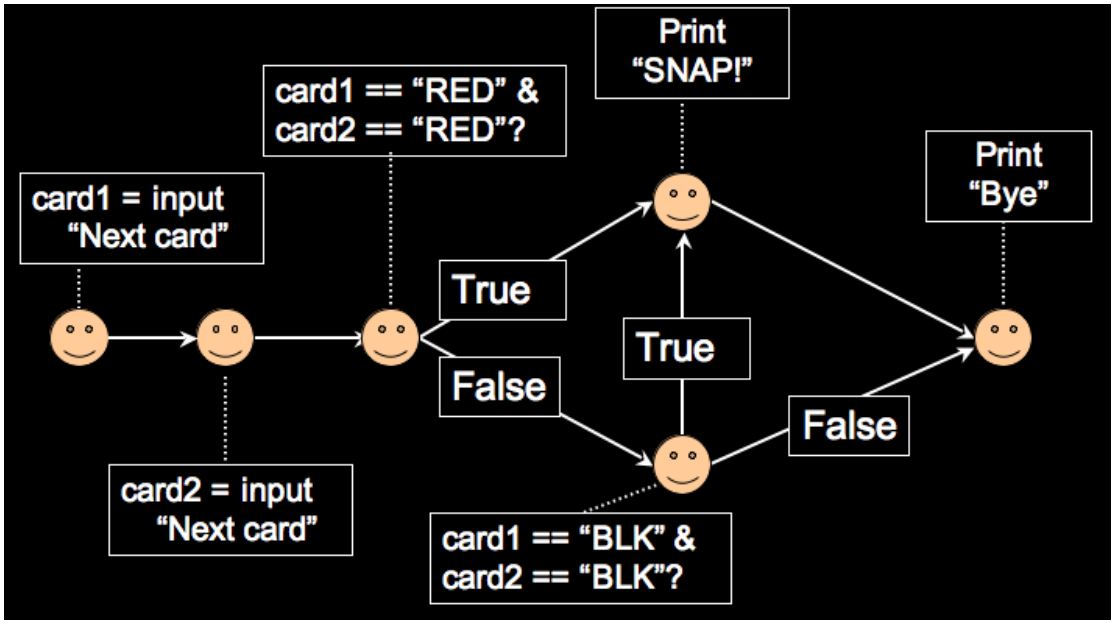
This compiles to the following Imp computer:



Compiler Optimisation

Imp computers can also be used to introduce compiler optimisation. For example get the students to look at the compiled version of the first Snap program. Ask if it could be compiled to a version with fewer Imps – fewer volunteers. That would correspond to a shorter, more compact program that took up less space when compiled. The two statements that print Snap could be combined as they do the same thing and jump to the same place.

By using this optimisation we would get the following simpler version from the same program.



Further Reading

Computing without computers

A free booklet by Paul Curzon on programming, data structures and algorithms explained using links to everyday concepts. Available from http://teachinglondoncomputing.org/resources/

Links to other activities

Box Variables

Execute simple programs that involve variables and assignment by running them on a computer made of students. Students with boxes act as variables as values are copied between them following the instructions of a program. You physically demonstrate the creation of variables, how accessing a variable involves taking a copy of its value, and how storing values in a variable destroys any previous value stored.

Assignment Dry Run

Dry run on paper a series of short fragments of code involving assignment.
This is an important activity to do after explaining variables and assignment. It reinforces understanding and helps identify faulty mental models so they can be fixed. It is a pencil and paper exercise executing code. Being able to do this kind of dry run for any new construct is an important prerequisite to being able to actually write code.

The swap puzzle

Solve a puzzle, coming up with an algorithm that your team can follow faster than anyone else.
This gives a way to introduce the idea of the solution to a problem being a set of instructions that allow others to 'solve' it with no understanding. It also explores how different algorithms can solve the same problem but may not be equally good – some may be faster.

The intelligent piece of paper

Take part in a test of intelligence against an intelligent piece of paper!
This is a good introduction to what an algorithms is and how a computer program is just an algorithm. It can also be used to start a discussion on what it would mean for a computer to be intelligent. It can lead on to an unplugged programming activity creating winning instructions.

Live demonstration of this activity

Teaching London Computing give live sessions for teachers demonstrating this and our other activities. See http://teachinglondoncomputing.org/ for details. Videos of some activities are also available or in preparation.

Department for Education
SUPPORTED BY MAYOR OF LONDON
COMPUTING AT SCHOOL
EDUCATE · ENGAGE · ENCOURAGE

Queen Mary
University of London

Instructions for An Insulting Program

Teaching London Computing / cs4fn: Imp computer
www.teachinglondoncomputing.org
www.cs4fn.org

Queen Mary
University of London

An Insulting Program

Instruction 1

answer = input
("Can I insult you?")

---

1. **Write on the board:**
   "Can I insult you?"

2. **Wait for the user to type something**

3. **Store the characters they type into storage box called <u>answer</u>**

Queen Mary
University of London

An Insulting Program

Instruction 2

if answer == “Y”:

---

1. **Look in storage box <u>answer</u> and check if it holds string “Y”**
2. **If it does then pass the tube LEFT**
3. **Otherwise pass the tube RIGHT**

Queen Mary
University of London

An Insulting Program

Instruction 3

print (“You smell!”)

---

1. Write on the board:
   “You smell!”

Queen Mary
University of London

An Insulting Program

Instruction 4

print (“You smell of roses!”)

---

1. Write on the board:
   “You smell of roses!”

Queen Mary
University of London

An Insulting Program

Instruction 5

print (“Thank you”)

---

1. Write on the board:
   “Thank you”

Queen Mary
University of London

Instructions for A SNAP Program

Teaching London Computing / cs4fn: Imp computer
www.teachinglondoncomputing.org
www.cs4fn.org

Queen Mary
University of London

A SNAP Program

Instruction 1

card1 = input ("Next card")

---

1. **Write on the board: "Next card"**

2. **Wait for the user to type something**

3. **Store the characters they type into storage box called <u>card1</u>**

Queen Mary
University of London

A SNAP Program

Instruction 2

card2 = input ("Next card")

---

1. **Write on the board: "Next card"**

2. **Wait for the user to type something**

3. **Store the characters they type into storage box called <u>card2</u>**

Queen Mary
University of London

A SNAP Program

Instruction 3

```
if card1 == "RED" and
   card2 == "RED":
```

---

1. Look in storage box card1 and check if it holds string "RED"
2. Look in storage box card2 and check if it holds string "RED"
3. If both do then pass the tube LEFT
4. Otherwise pass the tube RIGHT

Queen Mary
University of London

A SNAP Program

Instruction 4

print ("SNAP!")

---

1. Write on the board:
   "SNAP!"

Queen Mary University of London

A SNAP Program

Instruction 5

if card1 == “BLACK” and
card2 == “BLACK”:

---

1. **Look in storage box <u>card1</u> and check if it holds string “BLACK”**
2. **Look in storage box <u>card2</u> and check if it holds string “BLACK”**
3. **If both do then pass the tube RIGHT**
4. **Otherwise pass the tube LEFT**

Queen Mary
University of London

A SNAP Program

Instruction 6

print ("SNAP!")

---

1. Write on the board:
   "SNAP!"

Queen Mary
University of London

A SNAP Program

Instruction 7

print (“Bye”)

---

1. Write on the board:
   “Bye”

Queen Mary
University of London

Instructions for A 2nd SNAP Program

Teaching London Computing / cs4fn: Imp computer
www.teachinglondoncomputing.org
www.cs4fn.org

Queen Mary
University of London

A 2nd SNAP Program

Instruction 1

card1 = input ("Next card")

---

1. **Write on the board: "Next card"**
2. **Wait for the user to type something**
3. **Store the characters they type into storage box called <u>card1</u>**

Queen Mary
University of London

A 2nd SNAP Program

Instruction 2

card2 = input ("Next card")

---

1. **Write on the board: "Next card"**
2. **Wait for the user to type something**
3. **Store the characters they type into storage box called <u>card2</u>**

Queen Mary
University of London

A 2nd SNAP Program

Instruction 3

if card1 == “RED”:

---

1. Look in storage box <u>card1</u> and check if it holds string “RED”
2. If it is then pass the tube LEFT
3. Otherwise pass the tube RIGHT

Queen Mary
University of London

A 2nd SNAP Program

Instruction 4

if card2 == “RED”:

---

1. Look in storage box <u>card2</u> and check if it holds string “RED”
2. If it is then pass the tube LEFT
3. Otherwise pass the tube RIGHT

Queen Mary
University of London

A 2nd SNAP Program

Instruction 5

print (“SNAP!”)

---

1. Write on the board:
   “SNAP!”

Queen Mary
University of London

A 2nd SNAP Program

Instruction 6

if card2 == “BLACK”:

---

1. Look in storage box <u>card2</u> and check if it holds string “BLACK”
2. If it does then pass the tube RIGHT
3. Otherwise pass the tube LEFT

Queen Mary
University of London

A 2nd SNAP Program

Instruction 7

print ("SNAP!")

---

1. **Write on the board:**
   "SNAP!"

Queen Mary
University of London

A 2nd SNAP Program

Instruction 8

print (“Bye”)

---

1. Write on the board:
   “Bye”

Queen Mary
University of London

Name Sheets for program variables

Queen Mary
University of London

Name card

answer

Queen Mary
University of London

Name card

card 1

Queen Mary
University of London

Name card

card2