Theoretical Computers

Fun with finite-state machines

Trainer's Notes

Background

Look in any Computing textbooks aimed at Key Stages 3 and 4 and you will probably find an illustration of a computer in terms of a simple Input / Process / Output model. This is very helpful for abstracting away much of the technical detail of a particular device, and is a useful starting point for children to consider what a computer does. All computers ‘compute’ but for curious children this simple model simply begs new questions. What is a computation? How might it be represented? Computer scientists have developed models that tackle these questions. Finite-state machines are one way to represent the steps involved in a particular process. Many teachers will not be aware of these theoretical models. Finite-state machines (FSM) allow us to model processes and provide another way of expressing a sequence of commands ie a program. They articulate an algorithm through a ‘state diagram’. They are particularly useful in checking string patterns and allow us to introduce some key ideas behind the structure of ‘formal’ languages. Once familiar with the idea of a FSM, they also provide an alternative approach for solving challenges and creating simple programs.

Finite-state machines have limitations too. They are best considered as part of a hierarchy of computational models, spanning from simple data manipulation through Logic Gates to full blown general purpose computers. A FSM connected to some storage facility, provides the basis of a theoretical computer, a ‘Turing machine’. A theoretical model can play a central role in exploring the limits of computation. Just as algorithmic explorations uncover hard (or intractable) problems, through theoretical models computer scientists could reason about what was computable, and what wasn’t. Unlike the common sense idea that computers will, one day, be able to do anything, computer science can demonstrate there are things computers will never be able to do.

The aim of the day

The aim of the unit is summarised in the learning objectives for teachers. The primary aim is to ***educate teachers*** and illustrate the breadth and depth of Computer Science. The specific outcomes ***for teachers*** from this unit, are

- To understand the purpose of state diagrams.
- To appreciate the concept of a finite-state machine and provide examples.
- Understand their place in language theory and regular expressions.
- Be familiar with more advanced models of computation.
- Recognise the importance of a Turing Machine in defining algorithms and identifying the limits of computation.

The purpose of this Tenderfoot session is to equip ***trainers*** with material and ideas to meet these outcomes and broaden the outlook of teachers new to Computing. It hopes to provide a buffet of resources on which teachers can draw, to enrich their Key Stage 3 lessons, at the same time as meeting the key aim: providing greater depth of knowledge for ***teachers*** themselves. Developing ***teachers*** is the focus, not providing activities for pupils.

Throughout the material there are references to famous computer scientists and lots of pointers to other material. The aim is to encourage teachers to delve deeper and take ideas further.

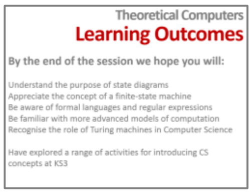
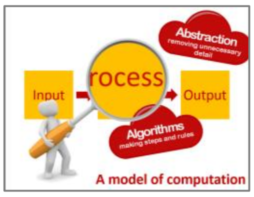
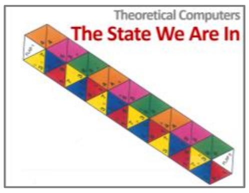
Before delivering the unit, please check you are comfortable with the narrative and references to other material. These trainer’s notes include a summary of each activity. Ensure you rehearse the delivery to familiarise yourself with transitions and animations. The slides include further detailed notes.

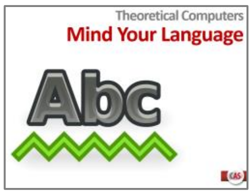
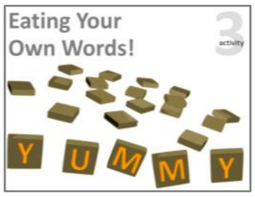
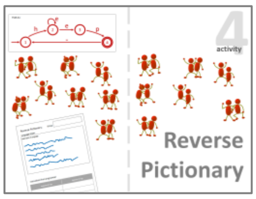
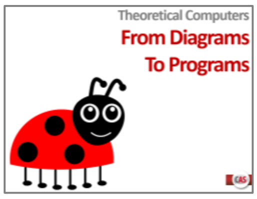
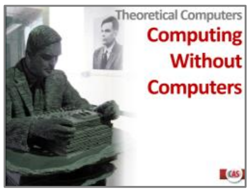
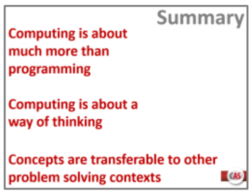
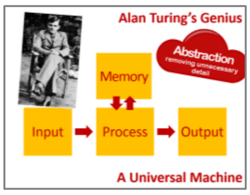
Keep in mind the main purpose of the session – to engage experienced teachers with some of the deeper ideas in Computer Science they may not be familiar with. This sort of background knowledge is broadly 'A level' standard. In time, all Computing teachers should have this grounding, so the aim is to empower the experienced teachers, providing them with material they can deliver and use with less experienced colleagues in shorter training sessions.

There are lots of exercises and supplementary material. The pace should be fast, with the assumption that the audience are experienced teachers, probably already teaching to GCSE level, with some familiarity with the concepts of Computer Science. As such they will not need to work through every activity in full. Sometimes it will be sufficient to part complete an activity so teachers 'get it' and can discuss how it might be used. Judgement is required and the timings below are indicative, to help with planning. Always be flexible and encourage discussion and engagement. Details of each activity are given in the teachers notes. Further guidance on the narrative, slide transitions and animation can be found in the slide notes.

Indicative Timetable

The trainer's presentation is broken down into 5 sections, with several formal theoretical inputs and 6 practical activities outlined below:

 Theoretical Computers **Learning Outcomes** By the end of the session we hope you will: Understand the purpose of state diagrams Appreciate the concept of a finite-state machine Be aware of formal languages and regular expressions Be familiar with more advanced models of computation Recognise the role of Turing machines in Computer Science Have explored a range of activities for introducing CS concepts at KS3  Abstraction removing unnecessary detail Input rocess Output Algorithms making steps and rules A model of computation 15 minutes	Establishes key outcomes from the day for teachers (5 mins). Sets the session in the context of the key educational goal: developing computational thinking (2 mins). Formal theory: Introducing the notion of a model of computation with reference to some core computational thinking concepts (5 mins).
 Theoretical Computers **The State We Are In** Fickle Fruit 2 activity For each group will need: A hat A scarf Several apples and bananas Fruit vendor instruction cards 90 minutes	A lengthy session to introduce the terminology and conventions of state diagrams. A practical exploration of the properties of a hexahexaflexagon (15 mins). Representing the behaviour as a state diagram (15 mins). Formal theory: Key rules for state diagrams (5 mins). Fickle Fruit: group activity observing, reasoning about, predicting and recording behaviour (20 mins). Harold the Happy Robot: developing a state diagram (15 mins) Looking at further suggestions and materials for class / homework activities: Airhead 2020, Thimble Slush and Perfect Pizza (15 mins)

 Theoretical Computers Mind Your Language  Eating Your Own Words! 75 minutes	Formal theory: Distinguishing between formal and natural languages. Introducing language theory, the structure of language, parse trees and derivation (20 mins). Treasure Hunt: explanation of an outdoor activity to introduce FSM (5 mins). Eating Your Own Words: outdoor exercise interpreting state diagrams and detecting patterns in strings. Introducing wild cards and notation for string patterns. Expressing rules for strings as state diagrams (20 mins).  Reverse Pictionary: class exercise encouraging students to design their own notations for regular languages; a motivator for learning a precise notation whilst reinforcing understanding of FSMs. Concludes by drawing out notation for regular expressions (20 mins). Formal theory explaining regular expressions. A look at Mr McDuff's Breakfast and other possible activities for pupils, and tools for teachers to familiarise themselves with regular expressions (10 mins).
 Theoretical Computers From Diagrams To Programs 60 minutes	A consideration of pedagogy: a different way to introduce programming. Expressing algorithms through finite-state machines (5 mins). A practical session introducing Kara, a simple programming environment. Group walk through to establish basics of the interface (20 mins). Practical programming challenges (30 mins).
Class based research DATA HYPOTHESIS Why? COMPUTING AT SCHOOL CAS Research 10 minutes	A short discussion to promote classroom research and encourage reflective practice. Draw out suggestions for potential research areas and mention possible techniques. Ends with a quick promotion of the BCS Certificate in Computer Science Teaching
 Theoretical Computers Computing Without Computers 45 minutes	Formal theory: Introduces Push-down automata and Turing machines. Summarises place of combinational logic, FSM and Turing machines in Chomsky's hierarchy. Emphasises importance of models of computation (10 mins). Chocaholic Turing Machine: A practical classroom exercise working through the stages required in a Turing machine designed to subtract one number from another (20 mins). A quick look at Turing Kara and other resources to take the ideas further (5 mins). Formal theory: Introduces the concept of a Universal Turing machine and the importance of the 'Halting problem' (10 mins).
 Summary Computing is about much more than programming Computing is about a way of thinking Concepts are transferable to other problem solving contexts 5 minutes	 Alan Turing's Genius Abstraction removing unnecessary detail Memory Input Process Output A Universal Machine Concludes by revisiting the starting point of the day, comparing a Universal Turing machine with our original model. Considers the wider impact of Alan Turing and lessons for children. Ends with emphasis on developing computational thinking. Distribute any materials and discuss ways to deliver smaller presentations.

Above all else, remember that the aim is to empower attendees to offer similar sessions to colleagues. It should be inclusive, enjoyable and embody the CAS ethos of collegiality: There is no 'them', only us!

When someone books to attend the training session, send a prompt acknowledgment informing them when final confirmation and further details will be sent. Set a cut-off date, at which point you decide if there are enough bookings to make a viable session.

Once you have enough people booked, contact them again with brief details and suggested prior reading. Although not essential, by suggesting some prior reading you are indicating that this is in depth CPD which requires some commitment on the part of the attendees. It also gives you a chance to establish some dialogue with attendees prior to the event. With a week to go, you could mail a reminder and enquire about the reading and whether it would be useful for teaching. This helps keep the attendees focused on the event.

Prior Reading

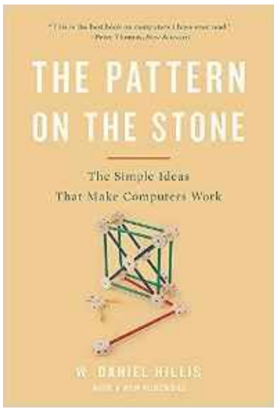
This is one of the more abstract topics covered in the Tenderfoot series and some teachers may struggle to see the relevance of the material if they come to it without some prior groundwork. A very straightforward video (9 minutes) explaining a finite-state machines can be found at youtu.be/vhiiia1_hC4. This is one of the Computerphile videos recommended in the presentation and an excellent introduction to the basics.

Accessible reading about finite-state machines, and their relevance to formal languages can be found in the CS Field Guide being developed at the University of Canterbury, New Zealand. This is still a work in progress. At the time of writing the relevant chapter was incomplete. Nonetheless, it provides an excellent introduction to the ideas that are developed in the session. Indeed, some of the material is used later the presentation. The suggested reading can be found at http://csfieldguide.org.nz/en/chapters/formal-languages.html, Chapter14, sections 14.1 to 14.3.1 inclusive. Later sections are recommended during the presentation.

Further Reading

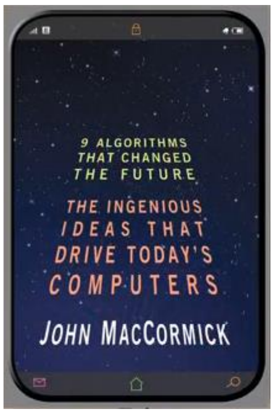
If attendees are inspired to investigate further the Computerphile videos (mentioned in the presentation) are a good starting point. Turing machines are introduced at youtu.be/dNRDvLACg5Q and the Halting Problem at youtu.be/macM_MtS_w4. A good explanation of Chomsky's Hierarchy is explained at youtu.be/224plb3bCog.

Much written material relating to language theory, grammars and finite-state machines is probably too advanced as introductory material. However, Babbage's Bag is a good source of material. An eclectic selection of articles on the I-Programmer website, their introduction to finite-state machines, goo.gl/4RE1Rk would make good reinforcement of the material covered in the session. Turing machines and the halting problem are tackled at goo.gl/uZKx3J, whilst goo.gl/bzF69c explains grammar, languages and Backus Naur Form in more depth.



Two books are also worth highlighting. Although long out of print, The Pattern On The Stone, written by W. Daniel Hillis is an excellent introduction to 'the simple ideas that make computers work'. Chapter 2 sets the idea of finite-state machines in that context.

A more recent book, Nine Algorithms That Changed The Future, by John MacCormick has a very accessible explanation (chapter 10) of the Halting Problem. The chapter offers an exemplar approach to explaining this difficult concept, which is covered at A level.


9 ALGORITHMS THAT CHANGED THE FUTURE
THE INGENIOUS IDEAS THAT DRIVE TODAY'S COMPUTERS
JOHN MACCORMICK

Well before the session is due to take place ensure you have considered computer access. Check whether attendees will be logging on to institution machines or bringing their own laptops. If BYOD, ensure that is made clear in any prior publicity. Check that the venue has a projector and speakers.

Kara (and Turing Kara) which is used in the session is a Java application. Check this will run on the institution computers or, with BYOD, distribute in advance to avoid issues on the day. No installation is required, but computers will require the Java Run-time Environment to allow the application to run.

Ensure there is outside space for the Eating Your Own Words activity, and any state diagrams are drawn out in advance. Pavement chalk is easiest for this, but check the hosts are happy. If not, consider marking out with string and tape.

Ensure you have the following general material:

- Facilities for taking notes (paper and pens)
- A3 Computational Thinking Posters
- CAS Publicity: Copies of SwitchedON, BCS Certificate flyers and any local information

Attendees Materials

Activity	Materials (Per Attendee)	
The Tuckerman Traverse	A3 hexahexaflexagon template, part complete, trimmed and folded	☐
	Hexahexaflexagon Instructions	☐
	Exploring A Hexahexaflexagon sheet	☐
Fickle Fruit	Fickle Fruit Vendor Instructions	☐
	Fickle Fruit Class Exercise and Solutions	☐
	Harold Happy Robot Input Rules	☐
	Designing FSM: Hair Dryer Exercise	☐
	Designing FSM: Vending Machine Exercise	☐
	SwitchedON Reprint: State Diagrams	☐
	The Perfect Pizza Problem	☐
Eating Your Own Words	I Have A Spelling Checker Resources	☐
	Eating Your Own Words Photocopy Masters	☐
	Eating Your Own Words Student Instuctions	☐
Reverse Pictionary	Reverse Pictionary Language Sheet and Diagrams (photocopy masters – per group)	☐
Kara The Ladybird	Kara application (software)	☐
	Kara manual	☐
Chocaholic Turing Machine	Chocaholic Subtraction Machine Rules	☐
	7 dark, 15 white counters, 6 coloured 'lollies'	☐

See overleaf for Trainers materials

Trainers Materials

Activity	Resources	
The Tuckerman Traverse	Large hexahexaflexagon pre constructed (card)	☐
	CS4Fn Hexaflexagon Automata Booklet	☐
	Hexahexaflexagon Instructions	☐
	Exploring A Hexahexaflexagon sheet	☐
	Teacher Notes	☐
Fickle Fruit	Hat and scarf	☐
	Fickle Fruit Vendor Apples and Bananas (laminated)	☐
	Fickle Fruit Vendor Instructions	☐
	Fickle Fruit Class Exercise and Solutions	☐
	Designing FSM: Hair Dryer Exercise	☐
	Designing FSM: Vending Machine Exercise	☐
	SwitchedON Reprint: State Diagrams	☐
	The Perfect Pizza Problem	☐
	Teacher Notes	☐
Eating Your Own Words	Pavement Chalk (or string / tape)	☐
	I Have A Spelling Checker Resources	☐
	CS Unplugged Treasure Hunt Resources	☐
	Eating Your Own Words Photocopy Masters	☐
	Eating Your Own Words Student Instuctions	☐
	Teacher Notes	☐
Reverse Pictionary	Reverse Pictionary Language Sheet and Diagrams (photocopy masters – per group)	☐
	Regular Expressions Further Investigations	☐
	Teacher Notes	☐
Kara The Ladybird	Kara application (software on shareable media)	☐
	Teacher Notes	☐
Chocaholic Turing Machine	Chocaholic Subtraction Machine Rules	☐
	7 dark, 15 white counters, 6 coloured 'lollies'	☐
	Teacher Notes	☐
Reflective Practitioner	BCS Certificate Flyers	☐
	Trainers Notes (laminated / card)	☐

Future Master Teacher Delivery

Trainers Notes to support Tenderfoot Unit 5: Theoretical Computers – Fun with finite-state machines

The unit presentation is designed to support a full one day session, delivered to CAS Master Teachers and other curriculum champions. It will likely be fast paced, delivered to experienced teachers.

It is envisaged that those attendees will take the material and deliver shorter sessions, either as half day, twilight of CAS Hub inputs. It is expected these will take longer to cover each activity as the material will be unfamiliar to teachers new to Computer Science. Please find time to discuss with attendees possible ways to use the material and encourage them to offer further sessions in their locality.

Resources

All supporting material is available for download, corresponding to each session in the Unit.

The material includes

- a full presentation to support all the activities covered in the Unit
- a set of Teachers Notes explaining the material for each session
- separate activity handouts

If you intend to use the material at shorter sessions, simply hide the slides not used.

If you wish to combine material in a different order please consider adding slides to introduce the 'big picture' at the start and to discuss being a reflective practitioner at the end. Please try to stick to the CAS House Style which is outlined on the opening introduction slide.

Half Day / Twilight CPD Sessions

It is suggested the material could be delivered as 4 separate shorter sessions, as indicate by the folders.

- The State We Are In: Introducing the concept of finite-state machines and state diagrams.
- Mind Your Language: Introducing formal languages and regular expressions.
- From Diagrams To Programs: Programming challenges using Kara.
- Computing Without Computers: Introducing Turing machines and Turing Kara challenges.

Each can be topped and tailed with the Introduction and Conclusion slides.

- Introduction
- Spelling Checker
- Treasure Hunt
- Eating Your Own Words
- Reflection / Conclusion

- Introduction
- Fickle Fruit
- Reverse Pictionary
- Regular Expressions Exercises
- Reflection / Conclusion

Of course, Master Teachers and other trainers can combine sessions and activities as they feel best fit the local circumstances.

However, a prerequisite for many of the activities is an understanding of a state diagram. The Tuckerman Traverse, Fickle Fruit, Treasure Hunt or even the Trainsylvania activity in the CS Field Guide all provide possible introductions to the concept of a FSM.

The four suggestions shown are just are a few of the possible combinations which allow time to explore some of the more formal exercises in more depth.

Many activities are short enough to introduce at CAS Hubs or worked through in a school departmental meeting. Whatever ways you choose, the aim is to develop teacher appreciation of CS concepts, not just demonstrate an activity. We hope they are useful.

- Introduction
- Fickle Fruit or Tuckerman Traverse
- Harold: Happy Robot
- Designing FSM: Hair Dryer
- Designing FSM: Vending Machine
- Reflection / Conclusion

- Introduction
- Tuckerman Traverse
- Kara The Ladybird
- Reflection / Conclusion

Theoretical Computers

Fun with finite-state machines

Session Activities Teacher Notes

The Tuckerman Traverse

Teacher Notes to support Tenderfoot Unit 5: Theoretical Computers – Fun with finite-state machines

A practical investigation, exploring the properties of a hexahexaflexagon. Their behaviour is captured in a State Diagram and through this the notion of a Finite-State Machine is introduced.

Preparation required:
Part complete class set of A3 hexahexaflexagons (trimmed and folded in half) with assembly instructions. A Hexahexaflexagon Exploration Sheet per student.

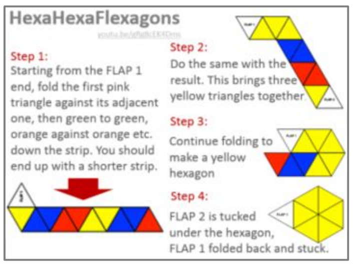
Models Of Computation

Input
Process
Output

Textbooks often explain computers by reference to Input/Process/Output - a general model of how something is computed. Computations can be expressed as algorithms – the steps required to accomplish a particular task. Tenderfoot Unit 5 introduces theoretical models of computation that formalise the notion of an algorithm. This may seem a bit obscure, but finding ways of expressing algorithms leads to some very big questions. Using models of computation, famous computer scientists have shown that there are things computers will never be able to solve, no matter how quick or powerful.

Hexahexaflexagons

Based on material by Paul Curzon (cs4fn), this activity explores the properties of a hexahexaflexagon and uses them to introduce 'finite-state machines'. The supporting booklet is included in the resources and at www.cs4fn.org/hexahexaflexagon/. A hexahexaflexagon is a curious hexagonal shape, made by folding a piece of paper. Their discovery is credited to British mathematician, Arthur H Stone, studying at Princeton University in 1939. Finding his English sized paper didn't fit his American folder, he tore a strip off and folded it up. The Princeton Flexagon Committee was formed with friends, Bryant Tuckerman, Richard Feynman and instructor John Tukey to explore their properties. Some years later (1956), mathematician Martin Gardner popularised them in the magazine Scientific American. In 2012, to celebrate Martin Gardner's birth, on 21 October, Vi Hart produced 3 wonderful videos telling the story of the hexaflexagon. They provide an excellent introduction to the Tuckerman Traverse: youtu.be/VIVIegSt81k. Can children figure out a way to cycle through and display all the faces of a hexahexaflexagon, returning to their starting point?



HexaHexaFlexagons

Step 1:
Starting from the FLAP 1 end, fold the first pink triangle against its adjacent one, then green to green, orange against orange etc. down the strip. You should end up with a shorter strip.

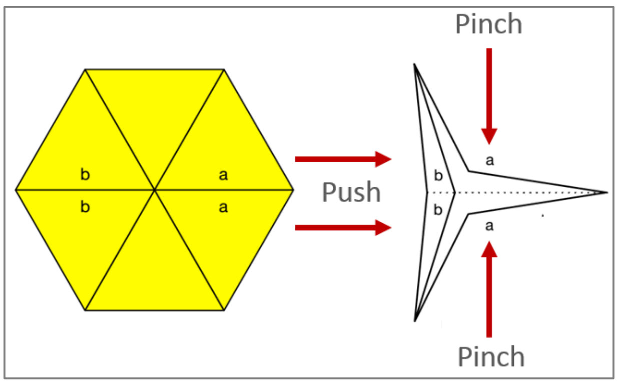
Step 2:
Do the same with the result. This brings three yellow triangles together

Step 3:
Continue folding to make a yellow hexagon

Step 4:
FLAP 2 is tucked under the hexagon, FLAP 1 folded back and stuck.

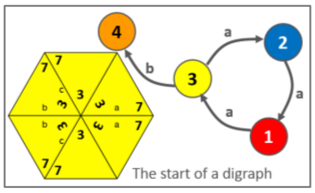
Resources include a template to make hexahexaflexagons. It should be enlarged by 141% to fit A3 paper. Small hexahexaflexagons are hard to manipulate. The shape is best provided cut out, folded along its length and stuck back to back, as accurate folding and sticking is essential. Students can work individually or in groups depending on the number available. The 4 steps on the handout show how to complete it. If students struggle, encourage them to use the video link on the handout.

Flexing is a matter of pinching and flattening the opposite side. The letters and numbers on the flexagon help the student's exploration. Start with the yellow side facing you. If assembled correctly, the 3's should be in a central ring and the lower case a's and b's together. *Always keep the flexagon facing the same way up*. By pinching and pushing, it turns inside out. The pinch points are indicated by adjacent pairs of lower case letters (a, b or c). Encourage initial exploration to see what can be discovered.



The Tuckerman Traverse

After a few minutes exploration, introduce the Tuckerman Traverse. Discussion prompts are given in the slide notes. How many faces does it have (9)? Some faces have the same colour, but can be seen to be different by the positioning of the digits. *With the flexagon kept the same way up*, each face is identified by the central ring of digits. How can we be sure we have explored them all? A diagram is an example of abstraction – removing details that obscure understanding. Which details are important? The 3's in the middle identify the face, so we can draw a node. When we pinch at 'a', we move to face 2. From face 2 we can't pinch anywhere and move back so the arrow is one way. The faces are indicated by nodes, and the transitions indicated by edges linking different faces. Diagrams like these are known as directed graphs (digraphs). The graph represents the transitions between faces. The edges have a letter assigned, indicating where to pinch, and the arrow indicates a direction.



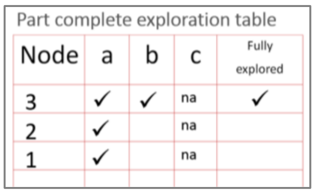
The start of a digraph

Exploring a HexaHexaFlexagon

Move from face to face on your hexahexaflexagon. There are many faces, some the same colour. Different faces are identified by the numbers in the centre. Draw a digraph of your findings below, noting the face number, and the letter you pinch to reveal another face. Keep track of your explorations by completing the table as you progress. This lets you check you've tried every possibility.

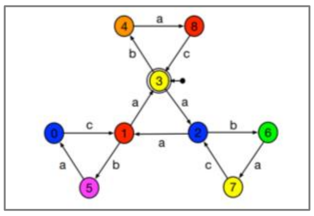
Node	a	b	c	Fully explored

Working in small groups, students complete the table to record results. Exploration needs to be systematic. The presentation records the first few moves together. Read the slide notes carefully to ensure you understand the animation. Make sure students complete the table and add the transition to a digraph in the box above.


Part complete exploration table

Node	a	b	c	Fully explored
3	✓	✓	na	✓
2	✓		na	
1	✓		na	

The completed graph models the hexahexaflexagon. It is now easy to plan a 'Tuckerman Traverse'. By abstracting away unnecessary detail the problem is easier to solve. Indicating where to start (with an arrow), students can trace the inputs required to move between different states. A double ring denotes an end state. A graph indicating a set of states and inputs required for each transition is known as a State Diagram. It expresses the behaviour of a 'finite-state machine'.



The graph denotes a finite number of states (nodes). It also indicates the actions (or inputs) required to move from one state to another (the letter to pinch). All possible actions (in this case the letters a, b and c) are known as the machines alphabet – the only acceptable inputs. State Diagrams can also output things. To keep it simple, the only output in this case is displaying the colour and number of the face. State Diagrams are extremely useful. They are a visual representation of a potential sequence of inputs/actions and can therefore be used to model many computational processes.

There is a more difficult extension. Turning the flexagon over reveals a face with no numbers in the middle. There are many more faces to explore flexing it this way up. Is a full traversal of all sides possible? To answer that, we need to decompose the problem into smaller explorations. Decomposing a complex task into smaller parts is a key concept in computational thinking. Suggesting what to investigate first makes a good discussion. Slide notes provide prompts but it is left as an open ended extension.

A simple utility for children to create their own hexahexaflexagon, offers a practical element to end the activity. The result is a vector graphic which can be scaled to maximise the print area available.

State Diagrams, as we have seen, can visually represent the behaviour of something that responds to input, has a set (finite) number of states, and can (if needed) output something too. Artefacts displaying such behaviour are all around us. In Computer Science we call them finite-state machines. Finite-state machines are extremely useful. They can be used to model many (but not all) computational processes. The presentation considers the behaviour of a ball point pen and a combination lock. Traffic lights are a more complex example. A homework exercise might be to draw a state diagram for a pelican crossing, or identify other household objects that can be modelled through state diagrams. Lots of simple electrical or mechanical devices will fit this description.

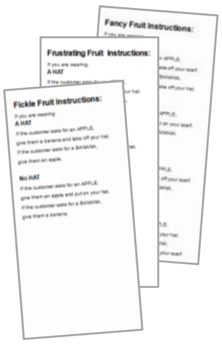
Fickle Fruit: Finite-State Machines

Teacher Notes to support Tenderfoot Unit 5: Theoretical Computers – Fun with finite-state machines

A finite-state machine is not really a mechanical entity, but an abstract set of instructions which a computer can be programmed to follow precisely. The activity, from the MathManiaCS project encourages children to reason about what they observe. Finding ways to record observations naturally leads to State Diagrams and the notion of a Finite-State Machine. More formal exercises are included to develop techniques illustrated in the activity.

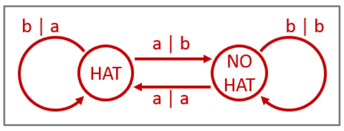
Preparation required:
For each group: A hat and scarf, Fickle Fruit Vendor Instructions, 6 apples and bananas (or pictures)
For each pupil: Fickle Fruit Class Exercise and Solutions, Happy Robot Input Rules, rough paper
Follow up exercises: Designing Finite-State Machines - Hair Dryer and Vending Machine exercises
Extension: The Perfect Pizza story (all can be provided electronically)

Fickle, Frustrating and Fancy Fruit



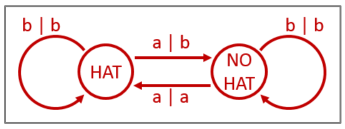
Start the exercise as a class, with one volunteer at the front as a fruit vendor. The vendor should wear a hat. Give the volunteer the Fickle Fruit Instruction card. No-one else should see it. Instruct students to request (in an orderly fashion) either a banana or apple. The vendor responds according to the instructions on the card. Encourage students to keep some record of their observations. When familiar with the routine, split into groups with one student as fruit vendor. They can take it in turns as the activity progresses.

In groups, with the fruit vendor in possession of the Fickle Fruit Instruction card only (and starting with a hat on), challenge students to place orders for *three consecutive apples*. If they get the answer quickly (banana, banana, banana), repeat the exercise with the vendor starting without a hat. Do they fully understand the vendor's behaviour? Try ordering *two bananas then an apple*, starting with the vendor in their current attire. Then try *two apples then a banana*. Bring to a close by asking groups to predict the behaviour of the vendor and share any methods they have used to record the behaviour. Has anyone drawn a state diagram? Swap vendors and use the Frustrating Fruit instructions. They can start with or without the hat. Issue the same challenges. Eventually, someone will realise that you cannot place an order for 3 consecutive apples. Stop the exercise and ask if they are sure?

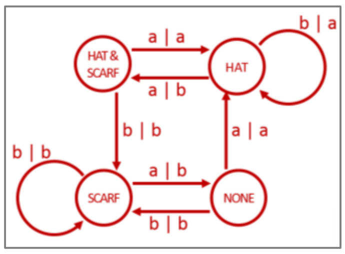


At this stage, we need an organised way of characterising the vendor's behaviour. A slide demonstrates the creation of a state diagram for the **Fickle Fruit** instructions. There are two states for the vendor: Hat and No Hat. We can represent the transition with an arrow. We'll use 'a' and 'b' to represent apple and banana. We can write the input and output on the transition, using a pipe to separate them. So a|b means we asked for an apple and received a banana. The presentation allows this to be completed as a class through questions and answers.

The Fickle Fruit Class exercise asks students to draw the diagram for the Frustrating Fruit vendor. Can the students explain why a three apple order is impossible? The answer is in the presentation. Note that a state diagram is completely deterministic. Every state must have a transition for every possible input. In this case, there are 2 states and 2 possible inputs, so four transitions.

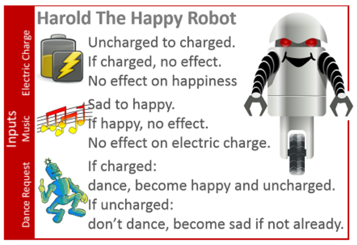


The final exercise is more challenging. The Fancy Fruit vendor has both a hat and a scarf. Challenge students to complete the state diagram on the exercise sheet. As they don't know the possible states in advance, complete a rough draft first. The solution (on a slide) illustrates a further issue. The transitions outgoing from the Hat & Scarf state are identical to those from the None state. This is an example of a redundant state. The diagram can therefore be simplified, collapsing the two states into one.



Harold The Happy Robot

Once familiar with state diagrams, as a group design a FSM from a given specification - the behaviour of "Harold, the happy robot". At any time, Harold is either **happy** or **sad**. He is also either **charged** or **not-charged** with electricity. There are three types of **inputs** that Harold might get: electricity, music or a 'dance' request. The rules that describe Harold's behaviour are fairly natural. They are reproduced on Happy Robot Card, so each student can have a copy.


Harold The Happy Robot

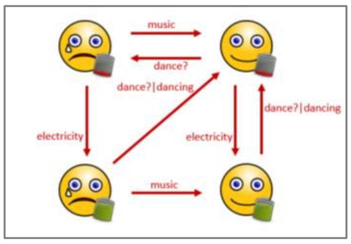
Inputs

Electric Charge
Uncharged to charged.
If charged, no effect.
No effect on happiness

Music
Sad to happy.
If happy, no effect.
No effect on electric charge.

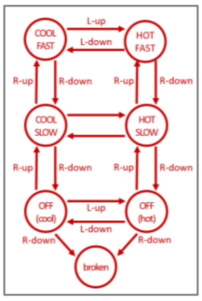
Dance Request
If charged:
dance, become happy and uncharged.
If uncharged:
don't dance, become sad if not already.

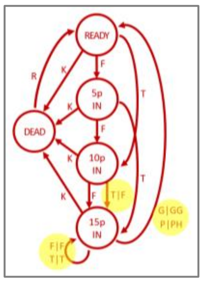
Deciding the states is the key starting point for state diagrams. Is dancing part of a state or is it an output? We'll make it an output (the only output), in response to a dance request input. So any output is associated with the transition, like the Fruit diagrams. Guide discussion towards discovering the states, with 'dancing' not part of any in this model: Sad / Uncharged, Happy / Uncharged, Happy / Charged and Sad / Charged. Discuss transitions from each in turn. The diagram builds in stages, question prompts given in the slide notes.



Designing Finite-State Machines

Once students have grasped the principles they could investigate everyday appliances. There are two exercises, a hair dryer and vending machine. In the case of the hair dryer, point out that there are no outputs on the transitions. The transitions represent the input (L-up/down or R-up/down). As the output is constant, it makes sense for the state to represent the output e.g. blowing cool air fast. There are two conventions for representing FSM's. Representing *output on the state*, as we did with the hexahexaflexagon, and are now doing with the hair dryer, is known as a Moore machine. Representing *output on the transition*, as we did with the Fickle Fruit machines is known as a Mealy machine. Whilst this detail is not needed at KS3, it does feature at A level. A state diagram will use one or other representation, the choice usually dictated by the task it represents. It is possible to express any Moore machine as a Mealy machine, and vice-versa.





The vending machine is more challenging. Discussion about representing output is helpful. The output is a discrete drink, before returning to a ready state so best represented on a transition – a Mealy machine. Deciding on states and types of machine can be difficult, often requiring intuition that comes from experience. Be prepared with some good hints. The solution to the Slush Dispenser is shown. To check understanding the slide animation highlights transitions showing output to use as discussion prompts.

FSM's can be very useful for modelling the design of interfaces. Homework might develop diagrams for setting the time or alarm on a digital watch. By modelling an interface, design errors can quickly be identified. If the FSM that describes a device is complicated it is a warning that the interface will be difficult to navigate. This is a huge area in Computer Science: HCI or Human Computer Interaction. Cs4fn host a video about setting microwaves (goo.gl/Bv4LZU), pointing out implications for interfaces in serious settings, such as hospitals. A reprint from SwitchedON, the CAS magazine, included in the resources looks at how software interfaces can be expressed as a FSM. It highlights the importance for software developers in separating the processing requirements from the interface behaviour and includes suggestions for further exercises.

One final, challenging exercise is also included in the resources. From the now defunct MegaMath website, the Perfect Pizza is a story in which the behaviour of the Babuie, the pizza maker can be explained by a finite -state machine. Slide notes include discussion prompts. A last suggestion, Manufactoria, (goo.gl/B28pKZ) is a very challenging game based on finite-state machines - probably one for teachers, rather than students!

Short exercises and an outdoor activity explore the use of finite-state machines to define language checkers. Introduces an awareness of language structure and the notion of context free grammars and language translation.

Preparation required:
'I Have A Spelling Checker', Eating Your Own Words instructions for all students.
FSA state diagram drawn in playground, paper extension exercise if required.
Treasure Hunt materials if undertaking that activity.

Eye Halve a Spelling Chequer

Eye halve a spelling chequer
It came with my pea sea
It plainly marques four my revue
Miss steaks eye kin knot sea.

Eye strike a quay and type a word
And weight four it two say
Weather eye am wrong oar write
It shows me strait a weigh.

As soon as a mist ache is maid
It nose bee fore two long
And eye can put the error rite
Its really ever wrong.

Eye have run this poem threw it
I am shore your pleased two no
Its letter perfect in it's weigh
My chequer tolled me sew.

(Sauce unknown)

The Structure Of Language

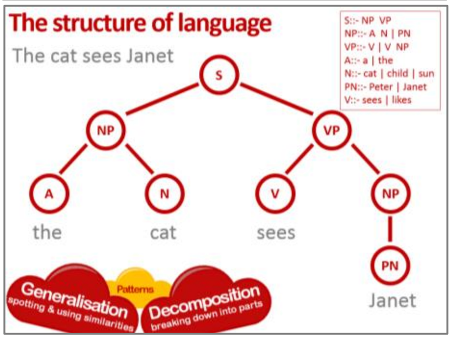
Have you ever wondered how a spell check works … and why it sometimes doesn't! The poem written to highlight their limitations was written by Professor Jerrold H. Zar, at Northern Illinois University, in 1992. The original and a simpler version (shown) are included. Deriving the correct meaning is a good introductory exercise. Homophones (words that sound the same, but are spelt differently) highlight the problem. Natural languages, such as English are full of ambiguity and there is more to them than just spelling and grammatical rules. Meaning is often dependent on context, which is why online translation services are notorious.

Natural languages evolve. Formal languages, by contrast are designed. Programming languages adhere to strictly defined rules. This makes it possible to accurately translate them into the machine code. If the syntax and grammar aren't perfect, the translator rejects them but there is no contextual ambiguity. Linguistics is an important area as computer science strives, not only to define more intuitive programming languages, but translation and grammar checking services too. CS4Fn have a child friendly introduction: goo.gl/B47YMv.

In the 1950's, Noam Chomsky laid the foundations for analysing languages. He defined a language as a sequence of 'atoms'. Taking English as an example, a sentence is made up of a sequence of words. Words are made up of a sequence or string of letters. The letters are the languages' alphabet – from which more complex sequences can be built. Not all possible sequences are valid words.

Sentence S	→	a Noun Phrase (NP) followed by a Verb Phrase (VP)
Noun Phrase NP	→	an Article (A) followed by a Noun (N) OR Proper Noun (PN)
Verb Phrase VP	→	a Verb (V) OR a Verb followed by Noun Phrase
Article A	→	'a' OR 'the'
Noun N	→	'cat' OR 'child' OR 'sun'
Proper Noun PN	→	'Peter' OR 'Janet'
Verb V	→	'sees' OR 'likes'

Grammatical rules define how the atoms can be combined and structured. In this example a sentence is defined as a combination of a Noun Phrase 'symbol' followed by a Verb Phrase 'symbol'. All subsequent symbols are defined until we get down to actual words in the language. The box showing the symbolic representation (note the pipe to represent OR) is known as Backus-Naur Form. Developed by John Backus and Peter Naur to define Algol, and is now the main technique for defining formal languages. To evaluate whether an expression is valid, a Parse Tree can break it down into defined symbols. This is known as a derivation. Further detail and examples are given in the slides / notes. Essentially it is an exercise in decomposition and pattern recognition.



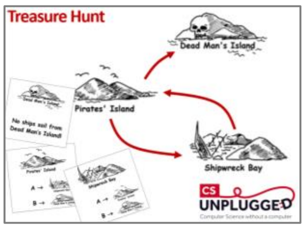
S::- NP VP
NP::- A N | PN
VP::- V | V NP
A::- a | the
N::- cat | child | sun
PN::- Peter | Janet
V::- sees | likes

Computer Science Field Guide
An online interactive resource for high school students learning about computer science.

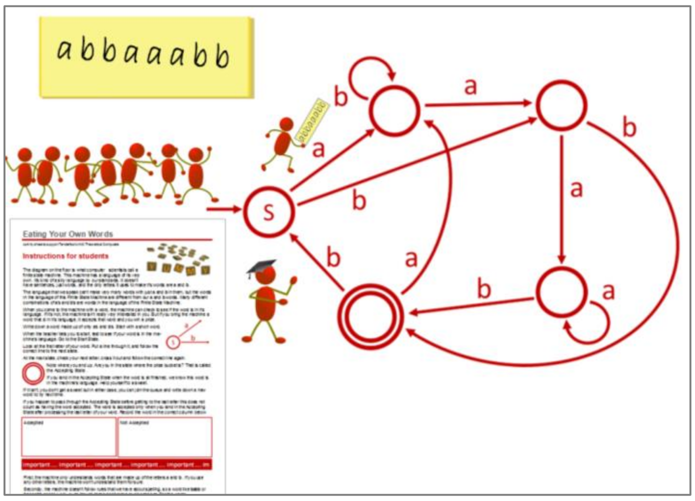
Languages can be very complex, illustrated by the Java example in the presentation. It has hundreds of rules but compilers use them to dissect statements into symbols, which decompose into defined atoms. There is a good introductory article at goo.gl/B8kR0p . The Computer Science Field Guide (csfieldguide.org.nz) chapter on Formal Languages is well worth reading to appreciate the depth of this area.

Finite-State Machines

Computer Science will be unfamiliar territory for many ICT teachers. Computerphile, a YouTube channel, has 250+ short videos on all aspects of computing topics. youtu.be/vhiiia1_hC4 is an excellent (8 min) explanation of link between finite-state machines, languages and computer science. There are several related videos. If you haven't familiarised students with the idea of a finite-state machine, the CSUnplugged Treasure Hunt activity described in the presentation is good introduction. As the main activity is outdoors, this makes a good starter. A 2 minute video: csunplugged.org/videos/#Finite_State_Automata explains it well. Such machines are known as acceptors or finite-state automata (FSA) since the output is indicated by where you finish.



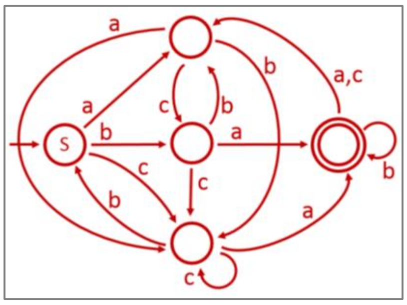
Eating Your Own Words



Students act out the operation of a FSA drawn in the playground. They create 'tickets' – a string of letters that serve as inputs. If they end up in an accepting state they receive a sweet - their ticket was recognized by the machine as a word in its language. An easier example is supplied but the one illustrated is a good starting point for KS3 students. The states should be large enough to stand in. Place a container with lots of small sweets in the accepting state (the double circle). Ensure all children can see what is happening.

Give out the instructions. The machine understands its own language, not ours. Made up of just two letters, a and b, it may seem a bit silly at first. There are lots of words in its language, but not all combinations of 'a' and 'b' are words it recognises. Start with short words (less than 8 letters). Give students the first few as a walk through. The teacher is best positioned between the Start and Accepting states. The task is to write a word made of 'a's and 'b's. If they <u>end up</u> finishing in the accepting state, they get a sweet. They don't if they are <u>passing through</u> the accepting state before the word is complete. Having finished, rejoin the end of the queue for another go. For those who catch on quickly, try words over 14 letters long.

Back in the classroom encourage children to articulate what makes acceptable words. Through pattern recognition, children generalise the rules of the machine, which they apply when choosing new words. What is the longest word in this machines language? Clearly we can't write a word with an infinite number of letters, so we need a shortcut for writing 'any number of' 'a's or 'b's. Children may be familiar with the use of wild cards, but if not, now is the time to introduce them. An asterix represents 'zero or more' of the previous character. To reinforce understanding, you could extend the FSA exercise by moving the Accepting State, adding a second Accepting State or ask students to choose words that involve the use of a wild card.



The exercise could also be repeated with the more complex machine included. This FSA introduces a 3 letter alphabet, and offers more challenge. It can be approached as a paper exercise, and is included in the photocopiable resources. Generalising from specific instances to broader rules takes practice. The presentation ends with three simple FSAs. Challenge the students to express the rules for acceptable words and ask if any can be expressed using wildcards? It's not easy. We really need tools for expressing some other rules. There are recognised shortcuts used to write general or 'regular expressions' ('regex' for short) and any regular expression can be expressed as a finite state machine. The next activity introduces regular expressions.

Investigating Regular Expressions

Teacher Notes to support Tenderfoot Unit 5: Theoretical Computers – Fun with finite-state machines

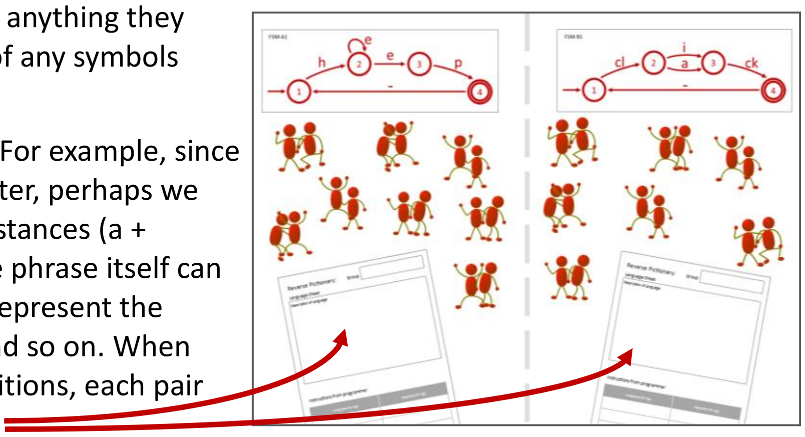
A practical activity to develop familiarity with finite-state automata (FSA) and a fun classroom demonstration. They encourage students to design their own notations for regular languages, and provide motivation for learning precise notation.

Preparation required:
Reverse Pictionary sheets for students.
A variety of props for Mr McDuff's breakfast.

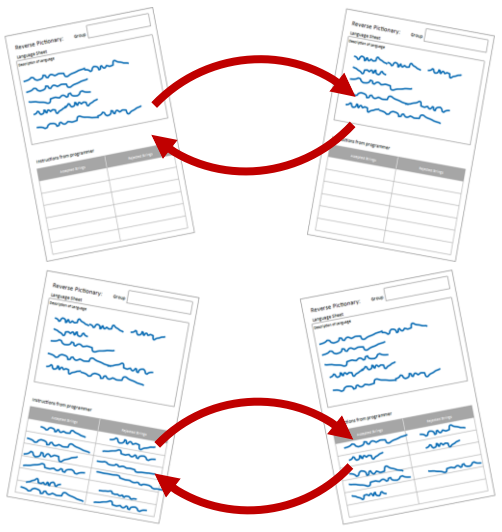
Reverse Pictionary

This exercise, developed by Linda Pettigrew, comes from the CS Field Guide produced in New Zealand. Split the class in half and ask them to pair up. Give each pair in one half a copy of the FSM-A1 diagram, and those in the other half a copy FSM-B1. Give each pair a Language Sheet as well. They will be writing in the top half only. Each half should not see the FSM the other half have been given.

Give the pairs five minutes to come up with a description of the words their FSM will accept. Encourage them to think about ways to describe repeating sequences. They may be familiar with some symbols used in regular expressions, such as *, but they can use anything they decide. But they also need to list the meaning of any symbols they use so others can interpret them.



You may need to help get the students started. For example, since * represents zero or more instances of a character, perhaps we should have a symbol to mean 'one or more' instances (a + perhaps). We could then write he+p. The whole phrase itself can be repeated, with a hyphen, so how might we represent the repetition of a string rather than a character, and so on. When they are happy with their description and definitions, each pair completes the Language Description box.



Pairs then swap the language description with a pair in the other half.

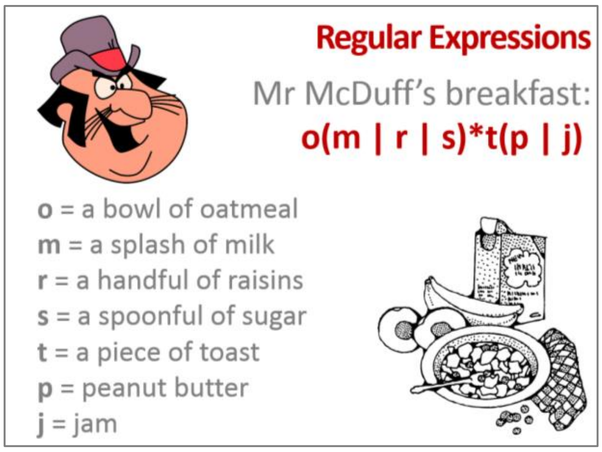
The receiving pair now complete the bottom half of the sheet. Having read the description they enter six strings they think would be accepted, and six they think would be rejected.

Finally, the sheets are gathered in and redistributed back to the original half. They do not have to return to the original pair. Each pair now acts as a 'computer' taking each input string provided at the bottom and confirming it conforms to the FSM. If a string is accepted / rejected incorrectly, the pair have to work out where the error arose.

Once familiar with the activity, a second set of FSM can be shared and the activity repeated.

Follow up discussion can investigate whether some descriptions were longer than needed, or confusing, and whether the language of the FSM was captured in the description. This provides a constructivist approach to introducing the notation of regular expressions whilst emphasising the point that any language expressed by a FSM can be represented as a regular expression.

Mister McDuff's Breakfast



Regular Expressions

Mr McDuff's breakfast:

o(m | r | s)*t(p | j)

o = a bowl of oatmeal
m = a splash of milk
r = a handful of raisins
s = a spoonful of sugar
t = a piece of toast
p = peanut butter
j = jam

An idea from the MegaMath project, Mr McDuff's Breakfast is a child friendly activity to introduce regex notation. This can be acted out to a class. Mr McDuff likes his breakfast. His choices are summarised in the items listed. Mr McDuff likes variety. Students should try to summarise the rules of his eating habits.

Make several breakfasts in front of the class until someone can explain that Mr McDuff always has a bowl of oatmeal and a piece of toast.

He sometimes has milk or raisins or sugar on his oatmeal. He may have it plain. If he has sugar, he may have several spoons, similarly with handfuls of raisins or splashes of milk. His toast will always have either jam or peanut butter. We can summarise all his breakfasts in the regular expression. Note the 'pipe' symbol (|) here signifies OR, not the distinction between input and output, as in a finite-state machine.

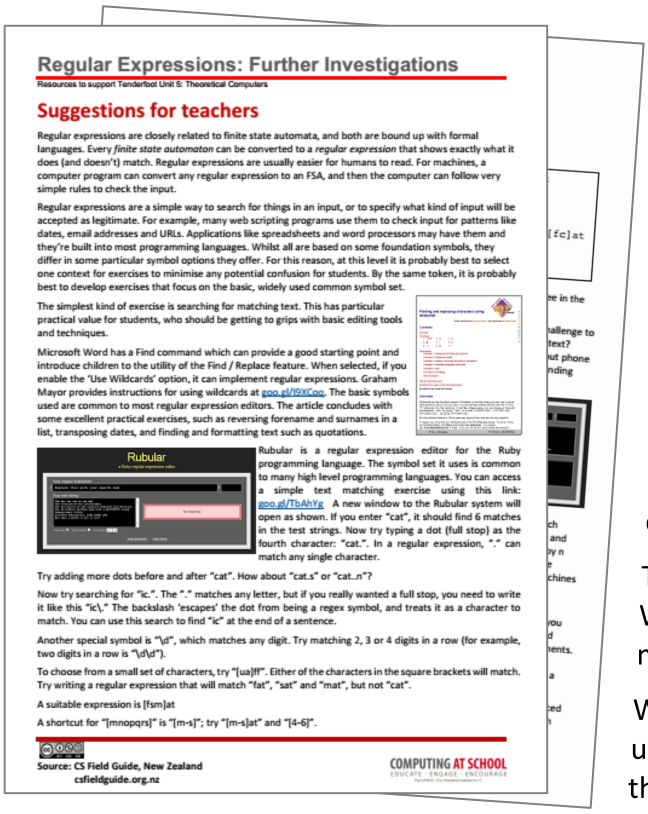
Three key symbols to emphasise:

- * represents zero or more of the preceding element
- | represents alternatives (or)
- ( ) enclose multiple items to which a symbol applies

Armed with this knowledge, can the children come up with a regular expression for their own breakfast? They should include anything they drink as well as eat.

Taking It Further

CS4Fn has a very good article aimed at secondary aged students. It introduces regular expressions by looking at knitting patterns. You can find the article via this link: goo.gl/h3lzvF. It introduces all the key notation for writing regular expressions. A good supporting homework.



Regular Expressions: Further Investigations

Resources to support Tenderfoot Unit 5: Theoretical Computers

Suggestions for teachers

Regular expressions are closely related to finite state automata, and both are bound up with formal languages. Every *finite state automaton* can be converted to a *regular expression* that shows exactly what it does (and doesn't) match. Regular expressions are usually easier for humans to read. For machines, a computer program can convert any regular expression to an FSA, and then the computer can follow very simple rules to check the input.

Regular expressions are a simple way to search for things in an input, or to specify what kind of input will be accepted as legitimate. For example, many web scripting programs use them to check input for patterns like dates, email addresses and URLs. Applications like spreadsheets and word processors may have them and they're built into most programming languages. Whilst all are based on some foundation symbols, they differ in some particular symbol options they offer. For this reason, at this level it is probably best to select one context for exercises to minimise any potential confusion for students. By the same token, it is probably best to develop exercises that focus on the basic, widely used common symbol set.

The simplest kind of exercise is searching for matching text. This has particular practical value for students, who should be getting to grips with basic editing tools and techniques.

Microsoft Word has a Find command which can provide a good starting point and introduce children to the utility of the Find / Replace feature. When selected, if you enable the 'Use Wildcards' option, it can implement regular expressions. Graham Mayor provides instructions for using wildcards at goo.gl/I9XCoq. The basic symbols used are common to most regular expression editors. The article concludes with some excellent practical exercises, such as reversing forename and surnames in a list, transposing dates, and finding and formatting text such as quotations.

Rubular is a regular expression editor for the Ruby programming language. The symbol set it uses is common to many high level programming languages. You can access a simple text matching exercise using this link: goo.gl/TbAhYg A new window to the Rubular system will open as shown. If you enter "cat", it should find 6 matches in the test strings. Now try typing a dot (full stop) as the fourth character: "cat.". In a regular expression, "." can match any single character.

Try adding more dots before and after "cat". How about "cat.s" or "cat..n"?

Now try searching for "ic.". The "." matches any letter, but if you really wanted a full stop, you need to write it like this "ic\." The backslash 'escapes' the dot from being a regex symbol, and treats it as a character to match. You can use this search to find "ic" at the end of a sentence.

Another special symbol is "\d", which matches any digit. Try matching 2, 3 or 4 digits in a row (for example, two digits in a row is "\d\d").

To choose from a small set of characters, try "[ua]ff". Either of the characters in the square brackets will match. Try writing a regular expression that will match "fat", "sat" and "mat", but not "cat".

A suitable expression is [fsm]at

A shortcut for "[mnopqrs]" is "[m-s]"; try "[m-s]at" and "[4-6]".

Source: CS Field Guide, New Zealand
csfieldguide.org.nz

COMPUTING AT SCHOOL
EDUCATE · ENGAGE · ENCOURAGE

RegexOne (regexone.com) is a good website for teachers to learn the basics of writing Regular Expressions. It's a little dry, and not recommended for children. However, students would benefit from some experience of writing regular expressions. In themselves they are good pattern matching exercises, an essential element in computational thinking. But practical exercises can also illustrate the sort of 'real world' tasks in which Regular Expressions (and Finite State Machines) might be used.

Regular Expressions: Further Investigations is a supplementary document which provides further suggestions and detailed exercises teachers can use in class.

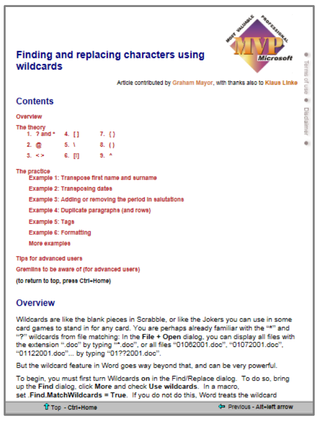
These start with practical exercises using the Find feature in Word to setting your own exercises using Rubular, one of many regex checking utilities.

Whether or not these are used with students, teachers unfamiliar with this area would benefit from working through the suggestions.

Suggestions for teachers

Regular expressions are closely related to finite state automata, and both are bound up with formal languages. Every *finite state automaton* can be converted to a *regular expression* that shows exactly what it does (and doesn't) match. Regular expressions are usually easier for humans to read. For machines, a computer program can convert any regular expression to an FSA, and then the computer can follow very simple rules to check the input.

Regular expressions are a simple way to search for things in an input, or to specify what kind of input will be accepted as legitimate. For example, many web scripting programs use them to check input for patterns like dates, email addresses and URLs. Applications like spreadsheets and word processors may have them and they're built into most programming languages. Whilst all are based on some foundation symbols, they differ in some particular symbol options they offer. For this reason, at this level it is probably best to select one context for exercises to minimise any potential confusion for students. By the same token, it is probably best to develop exercises that focus on the basic, widely used common symbol set.


Finding and replacing characters using wildcards

Article contributed by Graham Mayor, with thanks also to Klaus Linke

Contents

Overview

The theory

1. ? and * 4. [ ] 7. { }
2. @ 5. \ 8. ( )
3. < > 6. [!] 9. ^

The practice

Example 1: Transpose first name and surname
Example 2: Transposing dates
Example 3: Adding or removing the period in salutations
Example 4: Duplicate paragraphs (and rows)
Example 5: Tags
Example 6: Formatting
More examples

Tips for advanced users

Gremlins to be aware of (for advanced users)

(to return to top, press Ctrl+Home)

Overview

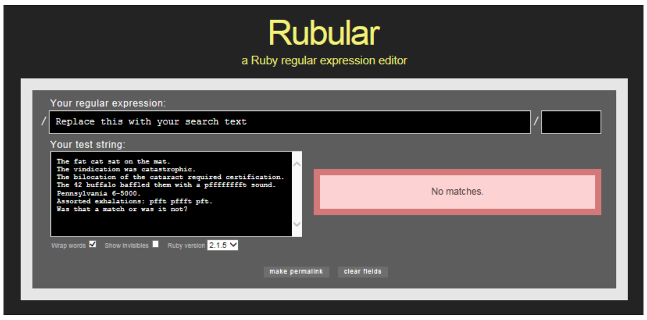
Wildcards are like the blank pieces in Scrabble, or like the Jokers you can use in some card games to stand in for any card. You are perhaps already familiar with the "*" and "?" wildcards from file matching: In the **File + Open** dialog, you can display all files with the extension ".doc" by typing "*.doc", or all files "01062001.doc", "01072001.doc", "01122001.doc"... by typing "01??2001.doc".

But the wildcard feature in Word goes way beyond that, and can be very powerful.

To begin, you must first turn Wildcards **on** in the Find/Replace dialog. To do so, bring up the **Find** dialog, click **More** and check **Use wildcards**. In a macro, set **.Find.MatchWildcards = True**. If you do not do this, Word treats the wildcard

The simplest kind of exercise is searching for matching text. This has particular practical value for students, who should be getting to grips with basic editing tools and techniques.

Microsoft Word has a Find command which can provide a good starting point and introduce children to the utility of the Find / Replace feature. When selected, if you enable the 'Use Wildcards' option, it can implement regular expressions. Graham Mayor provides instructions for using wildcards at goo.gl/I9XCoq. The basic symbols used are common to most regular expression editors. The article concludes with some excellent practical exercises, such as reversing forename and surnames in a list, transposing dates, and finding and formatting text such as quotations.


Your regular expression:

/ Replace this with your search text /

Your test string:

```
The fat cat sat on the mat.
The vindication was catastrophic.
The bilocation of the cataract required certification.
The 42 buffalo baffled them with a pfffffffft sound.
Pennsylvania 6-5000.
Assorted exhalations: pfft pffft pft.
Was that a match or was it not?
```

No matches.

Wrap words ✓ Show invisibles ☐ Ruby version 2.1.5

Rubular is a regular expression editor for the Ruby programming language. The symbol set it uses is common to many high level programming languages. You can access a simple text matching exercise using this link: goo.gl/TbAhYg A new window to the Rubular system will open as shown. If you enter "cat", it should find 6 matches in the test strings. Now try typing a dot (full stop) as the fourth character: "cat.". In a regular expression, "." can match any single character.

Try adding more dots before and after "cat". How about "cat.s" or "cat..n"?

Now try searching for "ic.". The "." matches any letter, but if you really wanted a full stop, you need to write it like this "ic\." The backslash 'escapes' the dot from being a regex symbol, and treats it as a character to match. You can use this search to find "ic" at the end of a sentence.

Another special symbol is "\d", which matches any digit. Try matching 2, 3 or 4 digits in a row (for example, two digits in a row is "\d\d").

To choose from a small set of characters, try "[ua]ff". Either of the characters in the square brackets will match. Try writing a regular expression that will match "fat", "sat" and "mat", but not "cat".

A suitable expression is [fsm]at

A shortcut for "[mnopqrs]" is "[m-s]"; try "[m-s]at" and "[4-6]".

Source: CS Field Guide, New Zealand

Another useful shortcut is being able to match repeated letters. There are four common rules:

- a* matches 0 or more repetitions of a
- a+ matches 1 or more repetitions of a
- a? matches 0 or 1 occurrences of a (that is, a is optional)
- a{5} matches "aaaaa" (that is, a repeated 5 times)

```
f+
pf*t
af*
f*t
f{5}
.{5}n
```

Try experimenting with the examples in the box left.

If you want to choose between options, the vertical bar is useful. Try the expressions in the box rightt and work out what they match. You can type extra text into the test string area in Rubular if you want to experiment.

Notice the use of brackets to group parts of the regular expression. It's useful if you want the "+" or "*" to apply to more than one character.

```
was|that|hat
was|t?hat
th(at|e) cat
[Tt]h(at|e) [fc]at
(ff)+
f(ff)+
```

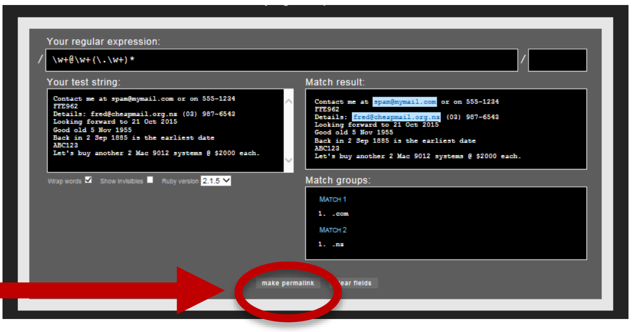
Once familiar with these try to write a regex that matches the first two words, but not the last three in the following link: goo.gl/9ZniUj.

Of course, regular expressions are mainly used for more serious purposes. Click on the following challenge to get some new text to search: goo.gl/MWIYCY. Can you write an expression to find the dates in the text? Here's one option, but it's not perfect: \d [A-Z][a-z][a-z] \d\d\d\d Can you improve on it? What about phone numbers? You'll need to think about what variations of phone numbers are common! How about finding email addresses?

Some of these are difficult challenges. A great feature of Rubular is that you can develop your own challenges with sets of test strings for students to use.

Note the 'make permalink' option in the screenshot.

This allows you to prepare your own test strings and share a link (like the examples above) with your students. If you are struggling with the e-mail challenge, the screenshot shows an example solution.



Your regular expression:

```
/ \w+@\w+(\.\w+)* /
```

Your test string:

```
Contact me at spam@mymail.com or on 555-1234
FFE962
Details: fred@cheapmail.org.nz (03) 987-6543
Looking forward to 21 Oct 2015
Good old 5 Nov 1955
Back in 2 Sep 1885 is the earliest date
ABC123
Let's buy another 2 Mac 9012 systems @ $2000 each.
```

Wrap words ☑ Show invisibles ☐ Ruby version 2.1.5

Match result:

```
Contact me at spam@mymail.com or on 555-1234
FFE962
Details: fred@cheapmail.org.nz (03) 987-6543
Looking forward to 21 Oct 2015
Good old 5 Nov 1955
Back in 2 Sep 1885 is the earliest date
ABC123
Let's buy another 2 Mac 9012 systems @ $2000 each.
```

Match groups:

```
MATCH 1
1. .com
MATCH 2
1. .nz
```

Regular expressions do have their limits — for example, you won't be able to create one that can match palindromes (words and phrases that are the same backwards as forwards, such as "kayak", "rotator" and "hannah"), and you can't use one to detect strings that consist of n repeats of the letter "a" followed by n repeats of the letter "b". There are other systems for doing that and these are an important part of the theory of Formal Languages. For computer scientists, an important part of the value of Finite State Machines and other theoretical models is in exploring the limitations of what they can do. Nevertheless, regular expressions are very useful for a lot of common pattern matching requirements.

As noted previously, any regular expression has a corresponding Finite State Automata. At KS4 or KS5, you might want to consider challenging students to code a Finite State Automata. It's a fairly straightforward challenge. Each state can be defined as a function, with the transitions expressed in a series of IF statements. The input string needs to be sliced, the first character examined and passed to the start state function.

Subsequent characters can be retrieved by putting the string slicing within a counter controlled loop. The current state can be tracked with a variable.

Much of the material for this section comes from the CS Field Guide produced in New Zealand. The student guide can be found at csfieldguide.org.nz, with supplementary material in the teacher guide at goo.gl/nMpwPZ.

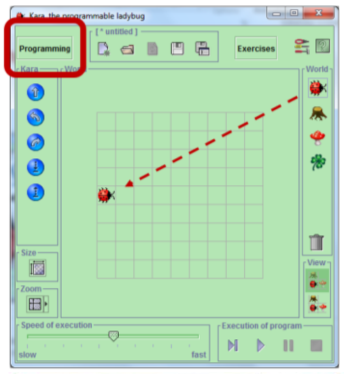
Kara is a programmable ladybird. This activity uses Kara as a vehicle for introducing programming and alternative ways of thinking about algorithms.

Preparation required:
Kara available on class computers. Check to ensure security settings allow Java based applications.
Kara manual for each student as reference.

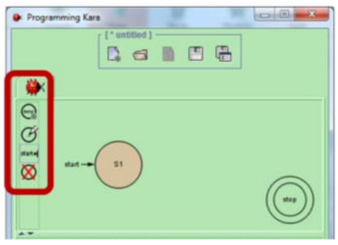
Finite-State Machines

This is a practical session looking at introducing programming through Finite State Machines. The application we'll use, Kara, was born in the research done by Raimond Reichert at ETH, Zurich (a leading STEM University) in the years 1999-2003. He was looking at ways that theoretical computers could be used to introduce the fundamental concepts of Computer Science and was an early advocate of making computer science a part of general school education. He wrote, "Programming practiced as an educational exercise … is best learned in a toy environment, designed to illustrate selected concepts in the simplest possible setting. The fundamental concepts of programming may be intellectually demanding, but they are not complex in the sense of requiring mastery of lots of details. Instead of using a programming language, we use a simpler model … finite-state machines." This approach has certain advantages. Reichert goes on to say, "It is easy to represent them in a graphical manner. Paths of execution are defined statically as paths in a directed graph; no other control structures are needed." A program can be defined by creating a state diagram, and children can quickly produce diagrams that can do fairly powerful things.

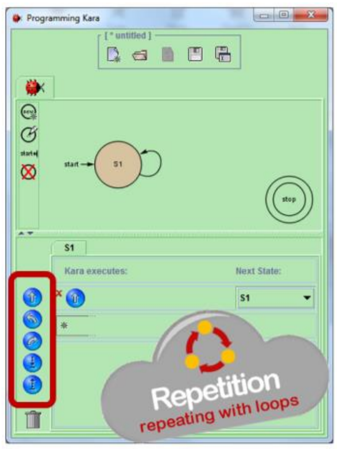
Introducing Kara



Kara, a Java application is free to download (goo.gl/OPZKor) and requires no installation. It comes with a handy 7 page manual which you can distribute to children. Both are included in the resources. The presentation is a practical introduction designed to be followed by the class. When you run the application, Kara's world opens. It is a very simple world, consisting of a grid of squares. Each square can have leaves, mushrooms or trees in them, but we start by just placing Kara in her world (by dragging).



The Programming button opens a second window where pupils build their Finite State Machine. It already has an end state in the diagram. Down the left hand side are controls to let you create, edit and delete states. Create a new state, named S1. The new state appears in your diagram. If you hover the mouse over the state, with the MIDDLE highlighted, you can move it into the centre of the diagram and make it the Start State, by selecting that option on the left.



With the mouse over the EDGE of the state, we can draw our transitions. Initially, draw a line, back to itself, as shown. In the lower half of the window, the transition has been recorded. Note the tab, which indicates which state we have selected. Notice also the 'Next State', which indicates we remain in S1.

Look at the options on the left. These allow Kara to move forward, turn left and right, pick up or put down an object. Add a move forward command to the transition definition (simply drag it across). When executed Kara will move forward one square, return to S1, move forward again, return to S1 and so on, moving forward continually. We've written our first program, expressed as a Finite State Machine, which implements a continuous loop.

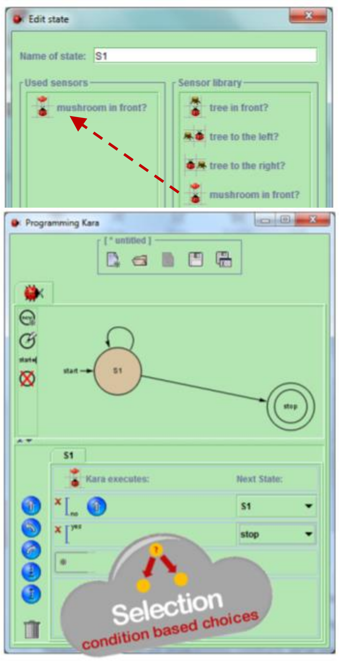
There are only 3 constructs in programming: Sequence, Selection and Repetition.

We can make Kara's world more exciting by adding a mushroom! Kara should continue moving forward until she bumps into a mushroom. Then she should stop. How might we express this in the FSM?

The discussion should draw out the need for a transition to the Stop state, but also the need to test for a mushroom at S1. We need to add a sensor, the input from which will determine which transition to take. With S1 selected, activate the Edit State option and add a mushroom sensor.

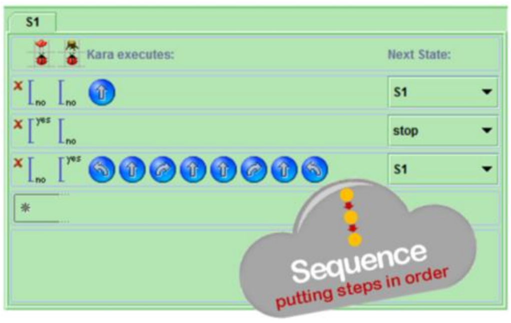
When you return to the Programming window, notice how the sensor has been added to the state definition. We can now set different actions in answer to the condition by selecting the Yes or No option, and adding a second statement moving Kara to the Stop state.

Thinking again about our programming constructs, we've now implemented an IF statement, or selection. The state diagram will update and when the program is executed Kara should stop at the mushroom.



The final part of the presentation adds a tree into Kara's path. When Kara detects a tree, we want her to go round it, before continuing on her way. Check the students can recount the steps involved:

- Edit S1
- Add a Tree sensor
- Assign the correct actions to each condition



With two sensors there are four possible combinations. In this case, we don't need to consider what would happen if Kara sensed both a tree and a mushroom in front because only one element can be on one square. However, students will need to consider all combinations when trees are detected to the side, whilst mushrooms are in front.

There are similar considerations with leaves, which are detected when they are underneath Kara. These are encountered in the tasks, and can be linked to work with students developing truth tables. Notice here how we have a sequence of actions triggered by a condition, the third of our key programming constructs.

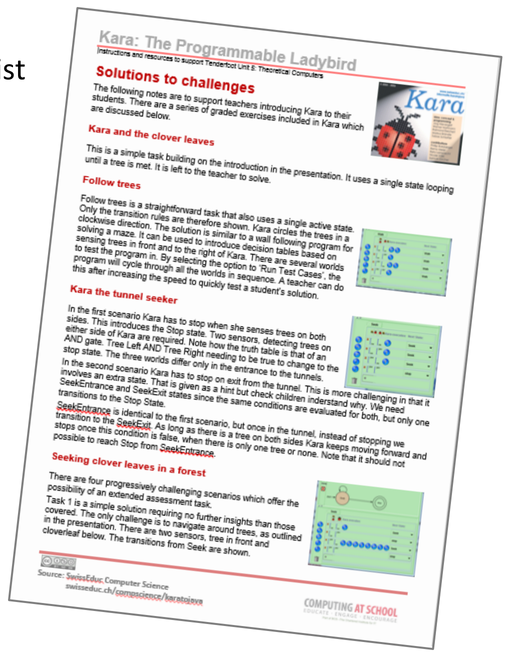
Using the usual icon saves a finite-state machine as a file with a .kara extension. Note that it will not save the world you have configured. To do that, you will need to use the same save icon in Kara's world. These are saved as files with a .world extension

Kara comes with a series of graded exercises built in. The drop down list in Kara's world reveals what is available.

The easy challenges should be accessible to KS3 children, whilst the harder challenges provide plenty of scope for differentiation and extension work.

As well as the explanation of the task, each challenge comes with several preconfigured worlds to try your solution. Notice that a world can be constrained to just a few tiles, rather than the whole grid.

Solutions and hints to the easy challenges are provided in the resources but it is worth teachers trying to solve as many as possible for themselves so they can appreciate any issues students will encounter.



Kara: The Programmable Ladybird

Instructions and resources to support Tenderfoot Unit 5: Theoretical Computers

Solutions to challenges

The following notes are to support teachers introducing Kara to their students. There are a series of graded exercises included in Kara which are discussed below.

Kara and the clover leaves

This is a simple task building on the introduction in the presentation. It uses a single state looping until a tree is met. It is left to the teacher to solve.

Follow trees

Follow trees is a straightforward task that also uses a single active state. Only the transition rules are therefore shown. Kara circles the trees in a clockwise direction. The solution is similar to a wall following program for solving a maze. It can be used to introduce decision tables based on sensing trees in front and to the right of Kara. There are several worlds to test the program in. By selecting the option to 'Run Test Cases', the program will cycle through all the worlds in sequence. A teacher can do this after increasing the speed to quickly test a student's solution.

Kara the tunnel seeker

In the first scenario Kara has to stop when she senses trees on both sides. This introduces the Stop state. Two sensors, detecting trees on either side of Kara are required. Note how the truth table is that of an AND gate. Tree Left AND Tree Right needing to be true to change to the stop state. The three worlds differ only in the entrance to the tunnels.

In the second scenario Kara has to stop on exit from the tunnel. This is more challenging in that it involves an extra state. That is given as a hint but check children inderstand why. We need SeekEntrance and SeekExit states since the same conditions are evaluated for both, but only one transitions to the Stop State.

SeekEntrance is identical to the first scenario, but once in the tunnel, instead of stopping we transition to the SeekExit. As long as there is a tree on both sides Kara keeps moving forward and stops once this condition is false, when there is only one tree or none. Note that it should not possible to reach Stop from SeekEntrance.

Seeking clover leaves in a forest

There are four progressively challenging scenarios which offer the possibility of an extended assessment task.

Task 1 is a simple solution requiring no further insights than those covered. The only challenge is to navigate around trees, as outlined in the presentation. There are two sensors, tree in front and cloverleaf below. The transitions from Seek are shown.

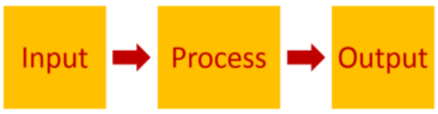
Source: SwissEduc Computer Science
swisseduc.ch/compscience/karatojava

COMPUTING AT SCHOOL
EDUCATE · ENGAGE · ENCOURAGE

A paired or group activity that demonstrates the behaviour of a simple Turing machine, in a child friendly fashion.

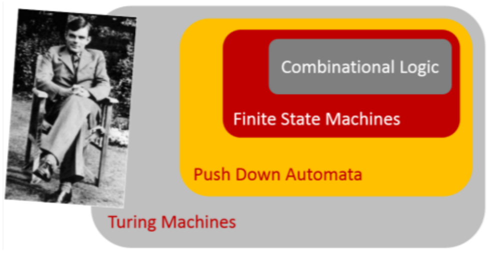
Preparation required:
Chocolate buttons (or counters). Minimum 7 dark, 15 white and 6 coloured lollies (or similar) per group
Chocaholic Turing Machine (cs4fn article) per child
Chocaholic Subtraction Machine Rules per group

Models of Computation



What is a computer, and what does it means to compute? Our starting point is a general model of how something is computed: taking some input, doing something to it before the results of that process are output. We have seen how finite-state automata, are mechanisms for expressing that computational process, and can describe simple programs as state transition diagrams. In computer science, we can think of finite-state automata as abstract models that helps convey a computational process.

Wikipedia has a good article on Automata Theory, with a diagram that puts combinational logic and finite-state machines at the heart of the theory (goo.gl/RAe5yz). We can demonstrate many simple processes in terms of collections of logical constructs (AND, OR and NOT). We see that when looking at the behaviour of specific circuits, for example an adding circuit, within a computer. We call that combinational logic.

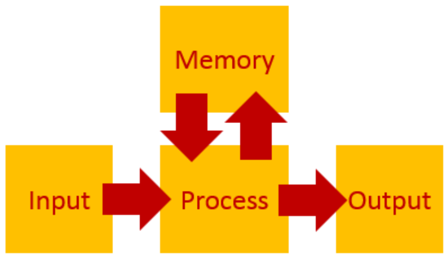


Finite-state machines are a more powerful abstraction – the next layer up in our models of computation. But there are things they cannot do. You can't devise a finite-state machine (FSM) to check if a phrase using brackets always has a closing bracket to match every opening bracket. Similarly no FSM can check if a word is a palindrome (spelt the same backwards). These sort of problems rely on pairing up. We can write FSMs that pair up specific numbers of items, but we can't write a general machine for any number of symbols. The presentation considers a FSM that tries to pair up cups and saucers. Notes for the animation are provided with the slide. Can you see the problem?

The name gives it away really. A FINITE state machine cannot have an INFINITE number of states to handle an indeterminate number of inputs. It can't 'keep count'. It has no memory of what has gone before, it only knows what state it is in. For this reason, more flexible models, known as Push Down Automata were conceived. In essence these are finite-state machines connected to a memory stack. Now we can just have two states for our Cups and Saucers machine, recognising a surplus of either. Each time a cup comes in, we simply remove a saucer from the top of the stack. When the stack is empty, we return to the Ready state.

Think back to the original model of a computer – the Input, Process, Output diagram. A Push Down Automaton can be thought of as this, with some form of memory attached to allow it to 'keep count'. As we've just seen, that memory is a 'stack' which limits what it can do. It can only retrieve things from the top of the stack.

A more general model, which allows access to any element of memory is known as a Turing Machine … and it has a remarkable history. In computer science, the different levels of the diagram above are known as the 'Chomsky hierarchy'. There's more to it than we have considered here, but if you'd like to delve deeper, Computerphile is the place to start (youtu.be/224plb3bCog). If you are likely to be teaching A level at any point all the related Computerphile videos are worth watching.

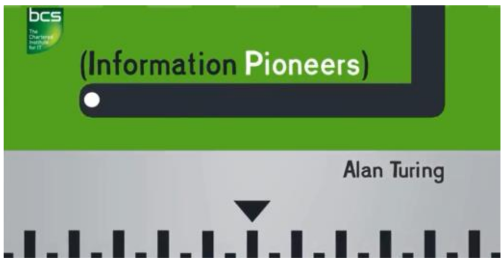


Turing Machines

The Turing Machine was conceived in the 1930's by Alan Turing, a mathematical genius best known for his role in breaking the Enigma codes in the Second World War. Whilst grappling with whether it was possible to determine in advance if a mathematical problem was solvable, various mathematicians tried to develop a formal definition of an algorithm. Several ideas were developed independently

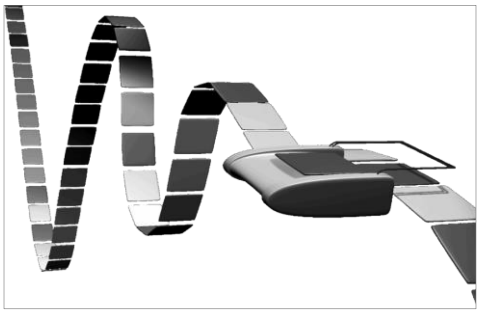
Despite the differences, there was nothing to choose between them – they were all shown to be equivalent to each other – an incredible coming together of ideas. Alan Turing conceived the idea of a computing machine, which was the most intuitive. He based it on the idea of a human 'computer' – someone who did calculations in a series of steps, using a scratch pad to jot down parts of the working out.

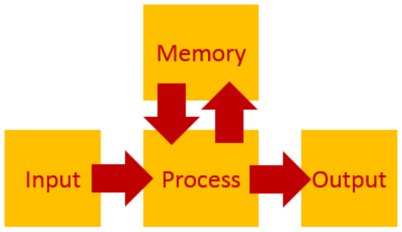
It is remarkable for two reasons. First, he had the idea long before any electronic computers had ever been built. Secondly, his model has been shown to be as powerful as any computer built since! By powerful, we don't mean fast, we mean it can compute anything any modern computer can compute. This is the heart of Computer Science. Turing machines can describe general computations. Having a model of computation allows us to reason about what sort of problems are computable, and what are not. Put another way, if something can be computed, a Turing machine can be designed to do it.



So what did Turing envisage? Actually, it is pretty simple. It consists of just two things: a very long tape that can move in either direction, and a mechanism in the middle that can look at the bit of tape beneath it, read what is on it, erase and write other symbols on if needed. A short video, ideal for students, explains the place of the Turing machine in the development of Alan Turing's ideas (youtu.be/gROuKl82BTk).

We can simplify a Turing Machine to a schematic: a tape and a read/write head. Superficially, it seems very different from our model of input-process-output, but consider the parts. The tape can have symbols written on it, either before the machine starts, or when it is running. So we have some means of inputting information. The contents of the tape, whilst it is running, are a store, just like memory in a modern computer. When the machine stops what is left on the tape can be considered the output.





'What it does whilst it is running is determined by the behaviour of the Read / Write head. The behaviour can be described by a finite-state machine. It acts like the computer processor. In fact we can think of modern computer processors as finite-state machines, albeit very complicated ones. They contain billions of logic gates which combine to put a machine in a particular state on each clock cycle.

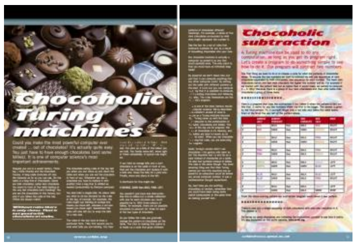
It may seem incredible to children that such a simple machine can do anything a powerful modern computer can, so it is worth stressing two important caveats:

- It has as much memory as it needs (the tape goes on for ever)
- It has a much running time as it needs

The second of these caveats is important to emphasise. Turing machines are slow – very slow!

At KS3, the implications of a Turing Machine for investigating the limits of computation aren't really important. What is, is to make the point that in order to understand how computers compute, we can abstract away the specific details of modern computers to focus on the essential process. We can make this point by demonstrating how a machine like this can do some simple computation. To do that in a fun way, we'll fall back on our traditional prop of bribing children with sweets.

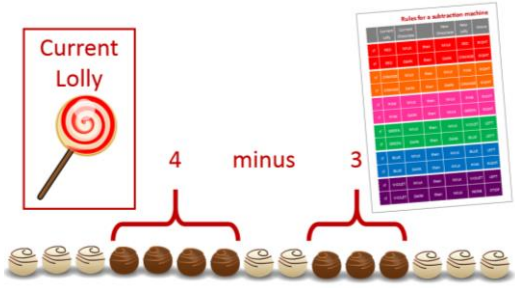
A Chocaholic Turing Machine



This activity comes from cs4fn, and was first outlined in their special issue to celebrate the centenary of Alan Turing's birth. The whole magazine (issue 14) is well worth reading (goo.gl/wYMZAM) if you wish to know more about his life and work. A copy is included in the resources.

Split the class into small groups or pairs. You need a large supply of chocolate buttons, both white and dark. Alternatively, use counters (two colours), but it isn't as much fun. For the example, a minimum of 7 dark and 15 white are needed for each group, more if other numbers are tried. You also need six different coloured lollies per group. These do not have to be edible and could be made of card. Our example uses Red, Orange, Violet, Blue, Green and Pink. The activity, with an introductory article can be found on pages 10 & 11 of cs4fn (above). Encourage students to read the article first.

The chocolates represent the starting tape of the Turing Machine. Arrange the opening chocolates in a line on the table. The presentation provides a walk-through of the opening moves. We are going to implement a subtracting machine. The dark chocolates represent the two numbers we have input, so we are going to perform the calculation 4 minus 3. Representing numbers in this way is known as the unary number system.

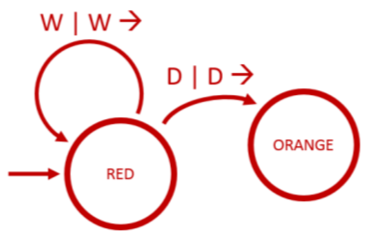


Our tape distinguishes between the two numbers by having some white chocolates between them. Each group also needs the rules for the Subtraction Machine. One person should be assigned the job of holding the Current Lolly. They start by holding the Red Lolly. Starting from the leftmost chocolate, students implement the rules outlined on the handout. The presentation includes prompts in the slide notes so the animation can proceed in response to answers from the children.

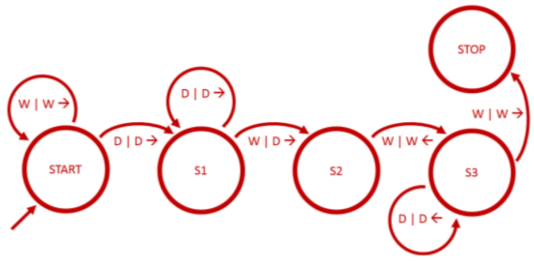
For the first few cycles the Current Lolly is RED, and the Current Chocolate WHITE so the Head moves right one place on each cycle. Eventually the Current Chocolate is DARK. It will remain DARK but the Lolly changes to ORANGE and the Head moves right. Because the lolly has changed, we now need to look at the rules for an ORANGE lolly. You might wonder if anything is ever going to change, but it will.

When a chocolate changes colour, remove the one from the tape and EAT IT! Replace it with the correct colour from the supplies. Children can take turns to move the head so they get an equal chance of eating a chocolate. Challenge them to work out the pattern of chocolates when the whole process is finished.

A further animation allows demonstration of the final two steps. The final output is displayed on the tape. One dark chocolate represents the result of the calculation (4 – 3). Ask students to set up their own subtraction using different numbers. It will work so long as the first number is larger than the second.

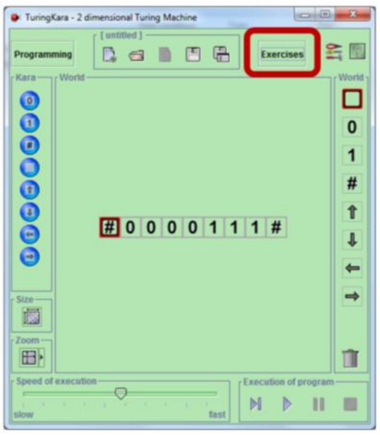


Ask students to express the rules as a finite-state machine. The presentation develops the first two states and notation for the transitions. The transition has to represent the initial read (input). It also indicates the tape contents after the action and the direction of travel of the head (output). Having modelled the first rule, written in the correct way, see if students can add the second rule. The subsequent slide gives the complete FSM.



A final challenge, for able students might be to devise a set of rules (or FSM) to increment a unary number by one. They can use the same conventions, dark chocolates bounded by white chocolates for the starting number. The solution (shown) can be worked through as a class example. To check understanding, one more transition is needed to trap a faulty layout – can anyone spot it? If students are stuck consider the transitions from S2.

Taking It Further



There is a version of Kara designed to implement Turing machines. Turing Kara is rather more challenging than Kara, the programmable ladybird, explored in a previous session. However, for very able students, it may provide good extension challenges. Like Kara, it comes with a set of exercises and the first two are accessible to able students at this age. Each challenge comes with its own worlds. They restrict the 'tape' to a short sequence which avoids a lot of potential confusion (in later challenges the tape can be represented as a grid). It also introduces 'bounding symbols (#), and uses 3 symbols on the tape, a 1 and 0, but also a blank square.

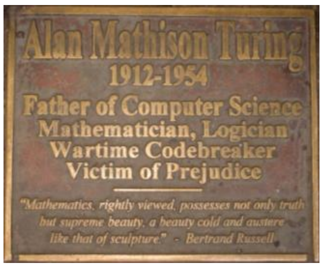
Finally, two more pointers, specifically for teachers. First, Rob Mullins from the Cambridge Computer Lab provides detailed instructions to use a Raspberry Pi to build a Turing machine that uses LED's and an interface developed in Python (goo.gl/ouz9Du). Second, on the centenary of Alan Turing's birth, Google had a wonderful doodle (goo.gl/bvFpbO) with 12 brain teasing puzzles. If you can't figure out what to do, goo.gl/DRCEM8 is an excellent explanatory article.

In Conclusion

Alan Turing demonstrated that anything that could be computed, could be computed by a Turing machine. Through this computer scientists could explore what was computable, and what wasn't. Unlike the common sense idea that computers will, one day, be able to do anything, computer science can demonstrate there are things computers will never be able to do. This is the significance of 'the halting problem', something you may have heard about, which is introduced at A Level. If a program is running and seems to be stuck, how do we know whether it is just taking a long time to perform a calculation or is stuck in an infinite loop? One will eventually halt, having performed the calculation, the other will go on for ever. Turing proved computers can't inspect every program and say, conclusively if they will run to completion. It is an example of an uncomputable problem, proving there are things that cannot be computed.

If this seems a little obscure, Turing also offered a more fundamental insight. In a Turing machine, the data is put on the tape. The purpose of the machine is captured in the finite-state machine that processes that data, like our subtracting machine. But Turing went on to show how the instructions for the finite-state machine could also be encoded as symbols on the tape. By running up and down the tape, the machine could read an instruction, then find the data and do something to it, before going to read the next instruction, and so on. He called this the Universal Machine.

We can conclude where we started. Turing envisaged a general purpose machine that could be programmed by putting different instructions in memory. He envisaged software before any software had been written. But since any Turing machine could have the instructions of any computational sequence put on its tape, all computers were equivalent in what they could, and could not do. Any machine could 'emulate' any other. In a sense, it is the ultimate 'computational abstraction'. A theoretical model of a computer that describes all modern computers … developed before any computer had ever been built.


Alan Mathison Turing
1912-1954
Father of Computer Science
Mathematician, Logician
Wartime Codebreaker
Victim of Prejudice
"Mathematics, rightly viewed, possesses not only truth but supreme beauty, a beauty cold and austere, like that of sculpture" - Bertrand Russell

Following his code breaking exploits in the war Turing went on to play a key role in designing the early computers. He laid the foundations for the field of Artificial Intelligence. At the time of his death he was exploring computational patterns occurring in nature, a field that is really just developing now. Quite simply, he was ahead of his time. He died young, a result of being hounded because he was gay. Turing was the father of Computer Science, but his life and death provide many other lessons for children today.

A new subject offers lots of potential for research on the part of teachers. Throughout the activities in this unit there is encouragement to consider issues, reflect on classroom practice and engage in action research. Not only does it contribute to practitioner's professional development, but can provide a key part of the evidence required for teacher accreditation via the BCS Certificate in Computer Science Teaching.

Preparation required:
Publicity for the BCS Certificate available for all attendees.
Familiarity with the questions to be posed and consideration of how to facilitate discussions.

Points To Ponder

Scattered throughout the trainer's presentation are 'Points To Ponder' slides. Usually at the end of an activity, they have a dual purpose, both practical and professional. From a practical point of view, by posing a question for short discussion, they provide the presenter with a few minutes to prepare for the next activity and gather their thoughts. Moreover, they encourage attendees to converse with each other. The aim is not to engage in a lengthy discussion, but to plant ideas that could be pursued as classroom research. Near the end of the presentation, we hope you will raise the value of such action research.

In this unit, the questions posed at the end of an activity are:

The Tuckerman Traverse: The National Curriculum makes no mention of finite-state machines. What is the value in considering the National Curriculum when planning our teaching?

Fickle Fruit: How many examples might a key stage 3 pupil need to be able to generalise the notion of a 'finite-state machine' (FSM)

Reverse Pictionary: What cross-curriculum benefits might accrue from studying finite-state machines and regular expressions? Could it impact on literacy?

Kara, The Programmable Ladybird: Does the use of state diagrams and finite-state machines offer an alternative approach for introducing programming?

Class Based Research

Why? Before embarking on the last activity, it is worth pausing to prompt a discussion about the value of classroom research both to support continuing professional development and to inform the wider teaching community. Use the terms "teacher enquiry", "action research", "reflective practitioner" and "practitioner research". Teaching computer programming is a relatively new activity. Many teachers are experiencing it for the first time. Those with experience can offer valuable support, advice and information to other colleagues. There are lots of opportunities for colleagues, old and new to contribute to an emerging pedagogy.

Inform attendees of the CAS Teacher Inquiry in Computing Education project. It provides a forum and focus for research in the field of computing. It can be accessed from the home page of the CAS website, under the link to projects: http://www.computingatschool.org.uk/

Research an area in which teachers have ready access to the data but be aware of researcher bias and local ethical considerations. Don't try to prove something you believe – always think that you are "bringing a better understanding" of the situation – it helps avoid bias.

What? Emphasise the key words such as “find out”, “evaluate” and “compare” to spark ideas. Each of the following give examples for a possible focus for research:

Find out from colleagues in English which aspects of this topic would support their pupils’ development.

Evaluate the use of finite-state machines as a vehicle to introduce programming. Focus your research on something you are trying out. Start by creating a question. Develop sub-questions that bring more detail or further focus your research.

Compare school data of the pupil’s performance in English and how well they engage with work on language structure and regular expressions.

Design, implement and evaluate a single lesson taught by different teachers. Explore how individual approaches impact on the central learning goals.

How? Briefly tell the audience of different methods. Choose one of the examples from the suggestions above, or those raised in ‘Points To Ponder’ and discuss which methods might be used:

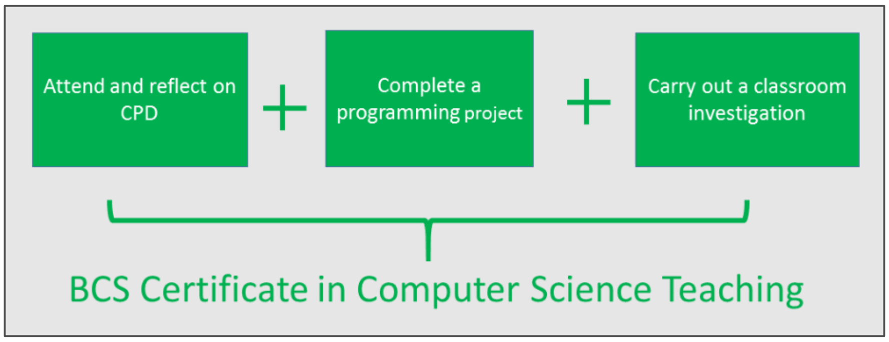
Questionnaires (including online) and surveys, interviews, focus groups (and online forums, wikis), scrutiny of documents including pupils’ work and blogs, scrutiny of numerical data and observation are all common techniques.

Less common but very effective might be analysis of professional conversations (in meetings, by email or via forums), analysis of pupil interactions/engagement in VLEs and forums, discourse analysis and content analysis.

Remember – all research has ethical considerations. Remind attendees of a school’s requirements and permissions regarding information about pupils and parents, in particular the idea of informed consent and freedom to withdraw their data from the research process. Remind them that if they embark on a University course then they will be governed by their ethics policy. Similarly, if completing the BCS Certificate then there is an explicit ethical requirement.

BCS Certificate in Computer Science Teaching

End with a reminder of the BCS Certificate in Computer Science Teaching – ensure hand-outs are available. One part of the evidence for the award is classroom research undertaken by the teacher. The other two areas involve attending CPD – like today, and completing a manageable programming project.


Attend and reflect on CPD
+
Complete a programming project
+
Carry out a classroom investigation
BCS Certificate in Computer Science Teaching

If that sounds a little daunting, remind teachers they can opt for either an independent or guided route. The latter means they have support from a mentor who can help guide them through the requirements. It is a valued award, giving professional recognition, accredited by the BCS, The Chartered Institute for IT. More importantly, it’s designed to help teachers in the work they are developing in school.

Encourage teachers to seek support from their schools to gain accreditation. Ensure they raise it as part of the performance management cycle.

More information is available at: http://www.computingatschool.org.uk/certificate.