From Problems to Programs with Computational Thinking

Greg Michaelson

Contents

0 Preface 5
1. Introduction 6
1.1. Overview 6
1.2. Computational thinking and information 7
1.3. Information structures 8
1.4. Finding information structures 9
1.5. Generalising information structures 10
1.6. From information to computation 11
1.7. Conclusions 12
1.8. Exercise 13
2. Pedagogy 15
2.1. Overview 15
2.2. Paulo Freire 15
2.3. Alan Turing 16
2.4. Modern movement 17
2.5. Functional programming 18
2.5. Exercises 20
3. Simple lists 21
3.1. Overview 21
3.2. Introduction 21
3.3. Shopping list 1 21
3.4. Shopping list 2 22
3.5. Shopping list 3 22
3.6. Implementing the shopping list in BYOB 22
3.7. Simplifying the algorithm? 23
3.8. Shopping basket 24
3.9. Shopping bill 24
3.10. Discussion 1 25
3.11. Procedural abstraction 25
3.12. Implementing the procedure in BYOB 26
3.13. Checking the lists 26
3.14. Discussion 2 27
3.15. Implementing the check in BYOB 27
3.16. Exercises 28
4. Dinner party 30
4.1. Introduction 30

4.2. Where to start? 30
4.3. Guest list..30
4.3. Seating..32
4.4. Mapping guests to seats..34
4.5. Reflection..35
4.6. Exercises 36
5. Pattern identification and abstraction..38
5.1. Introduction 38
5.1. Patterns, structures and abstraction 38
5.2. Functional abstraction..39
5.3. Array and iteration abstractions 40
5.4. Reflection..44
5.5. Exercises 45
6. Recursion..46
6.1. Introduction 46
6.2. Recursive function pattern and abstraction..46
6.3. Recursive procedure pattern and abstraction 49
6.4. Recursive songs 52
6.5. Non-terminating recursion..56
6.6. Reflection..57
6.6. Exercises 57
7. Sequences and patterns..59
7.1. Introduction 59
7.2. Powers of 2..60
7.3. Squares 61
7.4. Generation from sequence indices 63
7.5. Filtering from generated sequences..64
7.6. Reflection..64
7.7. Exercises 65
8. Imperative and object oriented programming..66
8.1. Introduction 66
8.2. The stack 66
8.3. Two stacks..67
8.4. Patterns and abstraction 68
8.5. Class = encapsulate(record + sub-programs)..70
8.6. Madness in the methods..72
8.7. Overloading constructors 73

8.8. Summary 74
9. Basic algorithms and data structures 77
9.1. Linear array 77
9.1.1. Linear Search 77
9.1.2. Binary search – ascending order - iterative 77
9.1.3. Binary search – ascending order – recursive 77
9.1.4. Swap 78
9.1.5. Bubble sort – ascending order 78
9.1.6. Bubble sort – ascending order - with success check 79
9.1.7. Quicksort – ascending order 79
9.1.8. Insert – ascending order 80
9.1.9. Delete – ascending order 80
9.2. Stack 81
9.2.1. Push 81
9.2.2. Pop 81
9.3. Queue 81
9.3.1. Join 82
9.3.2. Leave 82
9.4. Linked list – iterative/update 82
9.4.1. Show 82
9.4.2. Insert – ascending order 83
9.4.3. Delete- ascending order 83
9.5. Linked list – recursive/copy 84
9.5.1. Show 84
9.5.2. Insert – ascending order 84
9.5.3. Delete – ascending order 84
9.5.4. Sort – ascending order 85
9.6. Exercises 85
9.6.1. Linear array 85
9.6.2. Stack 86
9.6.3. Queue 86
9.6.4. Linked list – iterative/update 86
9.6.5. Linked list – recursive/copy 86
Appendix A. Haggis Pseudocode 88

0 Preface

This book is an on-going attempt to pull together my somewhat variegated ideas about how to teach programming.

I’d like to thank:

- Quintin Cutts of the University of Glasgow, for encouraging me to think about this peculiar stuff in the first place;
- my long suffering students and colleagues;
- Ian King of Kelso High School and his Scottish Borders teacher colleagues, with whom I first explored Chapter 1;
- participants of my workshop at the 2013 Computing At School Conference in Birmingham, with whom I further refined Chapter 1;
- Paul Rayner of St Brendan's College, Yeppoon, Queensland, Australia with whom I developed Chapter 3.
- participants at the SICSA Education workshop in 2015 at the University of Strathclyde, with whom I explored Chapters 7 and 8.

This material is very much work in progress. Comments and suggestions are always welcome.

Greg Michaelson
Heriot-Watt University
Scotland

G.Michaelson@hw.ac.uk

1. Introduction

1.1. Overview

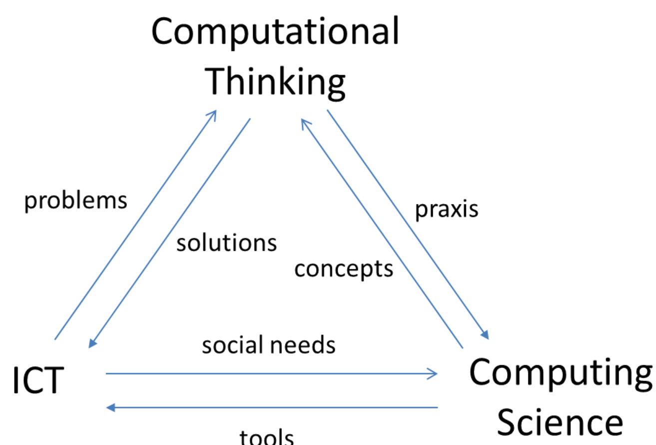
Computing involves making models of the world, to understand it and change it in predictable ways. We can then write programs that map the abstractions of a conceptual model onto the concrete behaviours of a physical computer so the behaviour of the program on the computer tells us about the world works.

We make models by solving real world problems. And we realise models as programs by programming real computers. So should we separate out problem solving, model making and programming, or are they inextricably linked? We also need languages and notations to solve problems, make models and write programs. Should these be the same or different? And isn't this all just Computing Science?

I think that it's important to distinguish *Computing* from Computing Science. Computing Science is an academic discipline which underpins all ICT, especially model making and tool making. Computing Science is concerned with the theory and practice of computations, which involves making models of reality from information structures and algorithms, and then animating the models on computers. That is, programming bridges models and computers.

It's been argued that everyone needs to learn how to program (e.g. Observer new Review, 1/4/12) and that somehow programming is the "new Latin". I think we need to teach everyone how to think, in particular how to characterise a problem before realising a solution in some given hardware and software technology.

I think we should view ICT, Computing Science and Computational Thinking, of which much more soon, as forming a Computing triangle:



where:

- Computing Science provides concepts for Computational Thinking in search of a praxis – that is synergy between theory and practise;
- ICT offers problems to Computational Thinking in search of solutions
- Computing Science responds to social needs from ICT with tools.

In homage to Niklaus Wirth's *Algorithms* + *Data Structures* = *Programs*[1] we might now say that:

information + computations = solutions

Thus, to solve problems, we can ask:
- how do we know when the problem is solved?
- what information is relevant to solving the problem?
- how must the information change for the problem to be solved?
- what computation(s) should we perform on the information to reach the solution?

The hardest part of problem solving is characterising the problem. Short of asking someone else, perhaps the best approach is to look for a similar problem we already know how to solve. And from our slogan, a similar problem will have similar information and similar computations: that's why it's likely to offer similar solutions.

1.2. Computational thinking and information

Recently, Computational Thinking (CT) has become seen as a key approach to problem solving. There are several different formulation of CT. My take, following Kao et al[2], is that it has four core techniques:
- decomposition;
- pattern identification[3];
- pattern generalisation/abstraction;
- algorithm design.

Decomposition is based on teasing out the basic building blocks of problems and involves:
- identifying the information needed to solve the problem;
- breaking the problem up into smaller sub-problems;
- identifying the sub-information needed to solve sub-problems.

Pattern identification is based on finding differences and involves:
- looking for patterns amongst problems. This requires us to think about whether we've seen a problem like this before, and, if so, how the new problem is different?
- looking for patterns in the information. This requires us to consider how the information is structured, whether there are useful relationships within the information, whether we've seen information organised like this before, and, if so, how the new information is different.

---

[1] Prentice Hall, 1973.
[2] E. Kao, Exploring Computational Thinking at Google, CSTA Voice, May, 2011, https://csta.acm.org/Communications/sub/CSTAVoice_Files/csta_voice_05_2011.pdf
[3] I used to call this "pattern recognition" until Roger Boyle pointed out that this is an image processing term.

Pattern generalisation and abstraction is the inverse of pattern identification. It involves using these differences to:

- find the general cases for our problem. Here we think about what does and doesn't change in how the sub-problems are organised.
- clarify the general organisation of the information? Here we consider what information does and doesn't change in the overarching information structure?

That is, we find common templates for the things that don't change with slots for the things that do change.

Finally, for algorithm design we think about:

- the sequence of steps from the initial information to the problem being solved;
- how the sub problems are connected;
- how the information changes between steps.

Now, it's easy to write down these stages but harder to see how they apply in practical problem solving. It's really not clear where to begin. And we already have tried and tested techniques for programming so why can't we retrofit CT to what we do already?

Here, options include:

- programming language oriented: full strength language, pedagogic, full strength subset, functional, logic, object oriented, reference language.
- simple to complex: stepwise refinement; structured programming; iterative prototyping.
- component: modular programming; algorithms; data structures; types; classes; libraries.
- design: flowcharts; data flow; entity relationship; state machines; UML.

Really, all we can say is that fashions change. There are too many possibilities and none work easily beyond simple cases. More to the point, they all have a strong focus on the final program. Programming isn't hard when you know how to solve the problem. It then becomes a matter of battling with the vagaries of language-specific syntax, semantics and tools. For people new to programming, this language specific detail can become overwhelming, leading to a plethora of tiny, low level concerns at the expense of understanding how to solve an original problem.

1.3. Information structures

Our familiar approaches are certainly useful at the end of problem solving when we come to implement a solution, but at the start it's vital to focus on the problem without regard for the implementation. the key is to work out how to characterise the information and computations. And it's best to start with the information, not the computation.

We can characterise information as:

information = base elements + structure

Note that we are not concerned with type or class, which are programming concepts to do with representing and implementing problem information.

Here, the base elements are the names of real-world things. At simplest, these are represented as sequences of characters to form meaningful unitary entities like words and numbers. Our things may also be built out of sub-elements, that is things themselves may be structured.

Thus, we find:
- sequences: things before and after each other, either unordered or ordered on some property;
- tables: things arranged in rows of columns;
- arrays: things accessed by indices;
- records: things accessed by field name;
- lists: things arranged in chains, accessed by heads and tails;
- trees: things arranged in branching structures, accessed by branches;
- graphs: things arranged as nodes linked by arcs.

Note again that we are not concerned with data structures which are programming concepts to do with representing and implementing problem information structures.

Note also that structures have equivalences. Thus, a table is an array or records of row/column contents; an array is a list of index/value records; a list ia an array of records of heads and tails. That is there are different, equally valid ways of characterising an information structure. Ultimately, the choice of structure is pragmatic: what gives the best characterisation of the problem in terms of, say, comprehension or abstraction.

1.4. Finding information structures

Alas, it's still no clearer where we should begin. It is tempting to go back to good old fashioned Computer Science. But there's no need. Rather, we should think about how real-world things are organised, and look at lots of concrete examples. Once we adopt an informational thinking approach, we can see information structures everywhere, often in how physical things are arranged but more conventionally in how data is organised.

For example, try thinking about how we interact with:
- parked cars;
- numbered houses;
- supermarket queues;
- English v Scottish bank queues;
- shopping lists;
- receipts;
- bills;
- account statements;
- itineraries;
- diaries;
- calendars;
- invitation lists;
- address books;
- seating diagrams;
- shop catalogues;
- library catalogues;

- family trees;
- cladistic trees;
- decision trees;
- underground maps;
- road maps;
- lottery tickets;
- betting slips;
- sports league tables;
- mobile contacts;
- browser favourites;
- social media friends;
- digital photo albums.

To characterise the information structure we need to interrogate the information. We need to ask how the information is organised, that is if it is:
- simple or composite;
- linear or grid or branching or cyclical;
- unordered or ordered;
- fixed or changeable in content or size or shape.

We can approach this by thinking about how to *access* the elements of the information structure by asking:
- why do we want to access the elements?
- which elements do we want to access?
- how do why/which affect access?
- where do we start the access?
- how do we continue access?
- how do we know if we've been successful or unsuccessful?
- what do we do if we're unsuccessful?

In the same way, we can ask how, if at all, we can *add* or *delete* or *modify* elements.

1.5. Generalising information structures

We now have some vague inkling about to identify a concrete information structure that fits some specific real-world problem scenario. Applying computational thinking, we can now try to generalise by comparing apparently disparate information structures. Thus we can ask what stays the same and what changes. We can also look for similarities in concrete detail and in gross structure, ignoring the elements. In particular, are there commonalities in how to access, update, add and delete elements. Ultimately, we can use these comparisons to draw out the abstract structures we identified above.

For example, what do: a car park and tables in a café, a bank statement and a utility bill, an allocated seat in an aircraft and in a cinema, a road map and an underground map, have in common? What are their differences? What common abstract information structure is suitable for both of them?

Finally, we can start to formalise the idea of an information structure as an *abstract data type*, with ways to:
- make a new structure;
- add information;
- check if information is present;
- find information;

- change information;
- remove information.

Not all information structures need have all these capabilities. For example, in many real world information structures, the information cannot easily be changed.

1.6. From information to computation

We can view these operations on abstract data types as being like the verbs of a language. We then make sentences by apply the verbs to nouns. That is, computation strature operations on information structures into algorithms. The key here is to make the structure of the computation follow the structure of the information. Note that we're making algorithms; we're not yet programming.

Once again, we should be driven by the problem scenario. We know that a computation solves a problem by turning old information into new information. So in the problem scenario we need to explore the sequences of information change.

Typically, we want to traverse the information structure, often visiting each element once, and stopping when some condition is met. On the way, we might do something to each element and accumulate some intermediate information.

For information arranged in fixed sequences, we typically want to traverse from some first element to some last element. So we need a notion of the next or current element. This is termed *bounded iteration*.

In other cases, we might want to continue traversal until some more general property is satisfied, and this might involve repeatedly visiting the same information or the same locations in the structure. This is termed *unbounded iteration*.

In both cases, we need to know how to start, continue and end the iteration, and this will be intimately associated with how we access and modify the information structure.

An alternative to iteration is *recursion*. Here, if we've got to the end of the structure, we stop traversal and maybe return a final value. Otherwise, we do something with the current element and then traverse the rest of the structure. Here we can distinguish the *base case*, where we stop, from the *recursion case* where we do the same thing to the rest of the structure.

As with iteration, we need to know how to start, continue and end the recursion, again closely following the information structure's properties.

Recursion and iteration are equivalent in expressive power. Alas, recursion is often seen as scary and advanced. In fact, in my experience, if you introduce

recursion before iteration, students find it natural. One way is through children's counting songs like:
- Ten green bottles;
- On man went to mow;
- Twelve days of Christmas.

We may well come back to this in another chapter..

During information structure traversal, we often need to keep track of intermediate stages. We may wish to remember different positions in a structure, for example where we last found something, or partial results, for example the value of the last element we found satisfying some property.

We can now introduce the idea of a variable as a general name/value association, where we can change the value associated with the name from a computation, often using the previous value.

I think that introducing variables should be delayed until they are needed to manage the stages of information structure traversal.

Coming back to accessing or changing an element in an information structure, we need to know how we can uniquely identify an element. Typically, we use what we might think of as a compound variable, for example the name of the structure qualified by a named field identifier or a numeric index.

Sometimes, we want to change an element regardless of its properties. But we may only want to change it if it satisfies some criteria. So now we can introduce notions of condition and choice. Similarly, we may want to change what we do next depending on properties of elements. That is, we can use conditions and choice to manage the stages of traversal.

Finally, we can stand back and think about whether the computation is necessarily:
- iterative, or could it be recursive?
- sequential, or could it be concurrent?
- linear, or could it be backtracking?
- deterministic, could it be non-deterministic?
- bounded, or could it be unbounded?

1.7. Conclusions

I have quite deliberately talked about problem solving in very general terms, without using any notation. But we do need to describe all these aspects of solving a specific problem in a concrete, consistent manner, so why don't we just use a programming language? After all, programming and seeing things working at an early stage are both highly motivating.

One problem is that the choice of language affects what can be described. Furthermore, the fine detail of a specific language can get in the way of

understanding fundamental concepts. Finally, the techniques used to knock up quick hit, small programs[4] usually don't scale well to larger problems and this can prove frustrating and demotivating.

I think that, for problem solving, we should use a neutral notation like a reference language. In the following material, I will use the Haggis reference language developed with Quintin Cutts.

To conclude, I think that we should focus on problem solving not programming. We've seen that:

computational thinking = decomposition+ abstraction + patterns + algorithms

Now, CT is a framework not a recipe. In CT, the components overlap and interact. It just isn't possible to separate out definite stages of decomposition, abstraction, patterns and algorithms. Finding these is itself a creative, iterative activity.

I think that classic CT overemphasises computation over information. For me:

solution= information + computation

I think that we should let the information structure the computation, and we should start with concrete instances of our problem scenario. We should then use CT to ask good questions, to tease out well known information structures and to guide computation design.

1.8. Exercise

a) Choose two very different scenarios from the list in 1.4.:

[4] "hacking"

- parked cars;
- numbered houses;
- supermarket queues;
- English v Scottish bank queues;
- shopping lists;
- receipts;
- bills;
- account statements;
- itineraries;
- diaries;
- calendars;
- invitation lists;
- address books;
- seating diagrams;
- shop catalogues;
- library catalogues;
- family trees;
- cladistic trees;
- decision trees;
- underground maps;
- road maps;
- lottery tickets;
- betting slips;
- sports league tables;
- mobile contacts;
- browser favourites;
- social media friends;
- digital photo albums.

b) Choose one information structure from the list in 1.3.:

- sequences: things before and after each other, either ordered on some property or unordered;
- tables: things arranged in rows of columns;
- arrays: things accessed by indices;
- records; things accessed by field name;
- lists; things arranged in chains, accessed by heads and tails;
- trees: things arranged in branching structures, accessed by branches;
- graphs: things arranged as nodes linked by arcs.

c) Explain how to represent both scenarios from 1.4.using the structure from 1.3.

This is well suited as an activity for pairs of people.

2. Pedagogy

2.1. Overview

Of course I'm making this all up as it goes along, but it is maybe useful to try and tease out what I think are the influences on the approach I'm trying to elaborate. Off the top of the head, I think there are four principle sources of ideas:

- Paulo Freire
- Alan Turing
- the modern movement;
- functional programming.

Let's briefly survey each of these in turn.

2.2. Paulo Freire

Paulo Freire (1921-1997) was a Brazilian educator who is justly celebrated for his critique of traditional education in reproducing often repressive social relations. In Freire's best known book, *The Pedagogy of the Oppressed*[5], he articulates an approach to teaching literacy through working with students to construct, and then critically reflect on, accounts of their day to day experiences. In so doing, his aim was to enable his students to be active and equal subjects of education, equipped to analyse and change society, rather than just passive objects who consume pre-defined knowledge.

I first read Freire in the late 1970's, coincidentally when I first started teaching Computing in what was then a Scottish Central Institution, now a "post-92" University. Much of what Freire said resonated with my own experiences at school, at University and then as a Lecturer. Longer term, I think Freire has informed and reinforced wish to make education an egalitarian and liberating activity for both learners and teachers. Here, however, I want to focus on Freire's pedagogy rather than his politics.

In Freire's approach, there are pleasing correspondences with the decomposition, pattern identification and generalisation stages in computational thinking. Alas, it is just not possible to do justice to these here: I strongly recommend Chapter 3 of *Pedagogy of the Oppressed*, which is a good read, despite both the unfamiliar terminology and the sometimes lumpy translation.

Rather, Freire makes three striking observations which I think are strongly relevant to all teaching but particularly pertinent to Computing.

First of all, education should be based on scenarios which are highly familiar to learners. For Computing this seems obvious: we should make our programming examples as motivating as possible by relating them to our students' day to day experiences. Often, though, our conceptions of our

---

[5] Penguin, 1972.

students' experiences do not correspond to their own. For example, scenarios drawn from sports or books or films or music often reflect our misconceptions about what interest our students, and the specific content dates quickly. The scenarios in the first chapter above have been chosen to be familiar to both students and teachers, and to be gender and age neutral.

Secondly, scenarios should be neither "*too explicit or too enigmatic*". If scenarios are too explicit then analysing them offers no challenge. For example, scenarios based on, say, collections of tunes or films or books scream table-of-rows-of-fields-in-some-alphabetic-order as this is precisely how familiar apps organise them. Conversely, if scenarios are too enigmatic, then there is less opportunity for students to make sense of them without teacher intervention and guidance. Thus, above I've tried to choose examples that do not have "obvious" pre-given structures but where there is enough implicit structure to enable straightforward analysis .

Thirdly, as suggested by Freire's collaborator Bode, one should explore alternative characterisations of scenarios rather than quickly homing in a single "obvious" solution, as such exploration both reinforces and enhances development of that solution. Thus, above, I've suggested that it's worth considering whether different information structures, however unlikely, are appropriate for a given scenario.

2.3. Alan Turing

Alan Turing (1912-1954) is now renown as one of the founders of contemporary Computing and Artificial Intelligence. Turing was originally a mathematical logician, working on characterising the limits to mathematics. The title of his seminal 1936 paper, *On computable numbers, with an application to the Entscheidungsproblem*[6], makes it sound like it's going to be impossibly hard to comprehend: indeed, most of it is highly technical, using notation and terminology which has long been superseded.

However, at the heart of the paper is Turing's elegant discussion of the plausibility of his model of Computing, based on what are now called Turing machines (p249-252). Here, Turing explores the general form of his machines by a painstaking low level analysis of practical paper and pencil computing. He starts by considering "*...writing certain symbols on paper. We may suppose this paper is divided into squares like a child's arithmetic book.*"

He then systematically explores whether or not:

- the squares need to be arranged in a grid – they don't;
- arbitrary many different symbols are needed – they aren't;
- arbitrary many different "states of mind" are required to manipulate the symbols – they aren't;
- whether it's necessary to change more than one symbol at a time – it isn't;
- and so on.

[6] Proceedings of the London Mathematical Society, Series 2, 42 (1936-7), pp 230–265.

In effect, Turing carries out a thorough analysis of Computing in general, in terms of information representations and of operations, systematically abstracting away from concrete detail to simplify his model as far as he can. Again, his approach has strong resonances with computational thinking.

For me, several things are germane here. First of all, Turing strongly roots his justification for his theoretical work in concrete human practice[7]. This well complements Freire's requirement that scenarios should be familiar.

Secondly, while Turing explicitly separates out information from its processing, he refines both aspects side by side. I think that far too much Computing teaching focuses exclusively on algorithms, often taking information and its organisation as given.

Next, Turing is keen to derive really simple ways to represent arbitrary information and characterise information processing, regardless of efficiency. Again, I think that too much Computing teaching homes in on using rich information representations, which may well be efficient and give rise to elegant algorithms, but which are really hard for beginners to grasp. Thus, I think it is far better to start with simple, if sub-optimal, information representations and algorithms, and then explore their application and limitations, to motivate more sophisticated approaches. I've tried to do this in the worked example of shopping in what is currently Chapter 3.

Finally, in exploring what he terms an "*intuitive argument*" to justify the properties he claims for his machines, Turing's style is succinct yet approachable, avoiding formal ideas and notations. In teaching Computing, it is tempting to dive into a programming language as soon as possible, to quickly animate a model on a computer to provide motivation and reinforcement. In contrast, while a working program is always the ultimate goal, I think that learning and understanding how to get there from a problem scenario is of crucial importance. That is, following Turing, I think we should try to use clear natural language to explore scenarios systematically in fine detail, rather than presenting formalised, pre-digested solutions. Arguably, I haven' altogether succeeded yet...

2.4. Modern movement

The modern movement was an international trend in architecture and industrial design that started in the late 19th century. Its roots lay in the new steel and concrete technologies that enabled the construction of buildings whose sizes, shapes and spans were no longer constrained by the limitations of stone. Now associated with stark tower blocks and sub-standard housing schemes, in its day, modern architecture led to the elegant and graceful

---

[7] Every numerate person can do sums by hand; a skill that pocket calculators, intelligent sales tills, and mobile apps seem to be steadily eroding.

skyscrapers of the early 20th century that typify the classic views of the Manhattan and Chicago skylines.

One of the first modernist architects, Louis Sullivan[8] coined the phrase:

> *form ever follows function*;

that is, the principle that what something looks like should reflect its purpose. When taken to extremes, this can lead to a rejection of all ornamentation resulting in a severe functionalism. However, this was a liberating philosophy for Sullivan and his contemporaries as they were no longer constrained to traditional designs by traditional technologies.

I think that this idea is significant for computational thinking in three respects. First of all, it suggests that the information and computational structures of systems should reflect the requirements of the problem. Of course this sounds obvious. But it is easy to become locked into design choices early on because they somehow feel right rather than because they actually reflect the needs of the problem. CT is precisely about teasing out structures latent in problems, without the preconceptions of “traditional” approaches.

Secondly, “form follows function”[9] suggests that a system should not do more than solve the problem at hand. It is often tempting to over design systems such that they do far more than the original requirement, perhaps in an attempt to somehow “future proof” them. However, over engineered functionality often gets in the way of meeting core needs. I think that systems should be no more complex than required for the job as specified. For example, it’s all too easy to reach for a sort when there’s so little data that a search will suffice. Similarly, it’s often tempting to design an object or record structure which contains information which is only going to be relevant if the requirements change.

Thirdly, “form follows function” implies that users should be able to quickly tell how to use a system because its design corresponds to their operational needs. This is particularly important for GUIs, where highly subjective notions of “user friendliness” can lead to unnecessarily confusing interfaces. Thus, interfaces should reflect upfront the primary uses for which the system is intended. One way to achieve this is to aim for simplicity in design.

2.5. Functional programming

Functional programming comes out of the same 1930s world of mathematical logic as Turing machines. While nobody today would dream of programming in Turing machine style, Turing machines are commonly used as a base line measure of the complexity of algorithms. In contrast, functional programming has a significant if minority following for contemporary practical Computing, in particular in Haskell and Standard ML. And ideas from functional

[8] "The Tall Office Building Artistically Considered". Lippincott's Magazine (March 1896): 403–409
[9] in the popular shorter version.

programming are now incorporated cleanly in imperative languages, for example anonymous and higher order functions in Ruby and Python.

Alas, functional programming has been wildly over-hyped. Honestly:

- functional programming is not the solution to the software crisis;
- functional programming does not ease program proof;
- it is not easy to exploit useful parallelism in functional programs;
- functional programs can be just as opaque as they are succinct.

In reality, functional programming is just another programming style with its own balance of theoretical and practical strengths and weaknesses.

On the other hand, functional programming has an unfortunate reputation as a hard discipline, based on squiggly notations and really difficult ideas like:

- recursion;
- recursive data structures;
- polymorphic types;
- anonymous functions;
- higher order functions;
- not to mention monads, monoids and other mono-manias...

In reality, functional programming is no harder than any other programming style, once you know how to do it.

For me, the key functional programming idea is:

> *make the structure of the program correspond to the structure of the data.*

That is, the structure of the information should come first, and strongly guide subsequent choices about computation structures. This fits neatly with “form follows function”.

Functional programming encourages this approach because there is a close correspondence between how information and computations are represented. In particular, in functional languages, recursive data structures, like linked lists and binary trees, have direct equivalences in the ways that recursive functions to process them are defined.

Typically, a recursive structure is either empty or has components which are themselves combinations of atomic values and recursive structures, all glued together by operators called constructors. Then, a function to process a recursive structure typically has a base case for an empty structure, and recursion cases for non-empty structures, which use patterns of constants, variables and constructors to match and pick up the structure’s components.

The beauty of functional programming is that this close correspondence generalises to arbitrary information structures. That is, we can identify:

- a general information pattern for representing an empty or non-empty structure, using variables to name the significant components and operators to show how they fit together;
- a corresponding computation abstraction as a function with cases for the empty and non-empty forms of the information pattern, operating on the variables for the components.

In short, if we have a way of representing instances of an information structure, we can read off the structure of the computation from the structure of the information.

So, if functional programming is so well aligned with CT, why isn't this book based on it? Well, we'll look at this in the forthcoming chapter on choice of programming language for teaching...

2.5. Exercises

a) Discuss the view that you can't treat students as equals because it makes you look weak, so they play up and you can't ever regain control. Instead you have to start hard and establish who's boss.
b) Discuss the view that ideas from mathematics and logic are of little relevance to teaching contemporary Computing.
c) Discuss the view that if you don't make an app look attractive then no one will want to use it.
d) Every iteration has an equivalent recursive form and vice versa, so what are the advantages and disadvantages of teaching repetition through recursion before iteration?

3. Simple lists

3.1. Overview

In this activity, we're going to think about simple lists. Here, we don't mean linked lists with heads and tails, as found in declarative languages; rather the sort of list one might jot down on the back of an envelope.

To motivate this, we're going to consider the use of simple lists in shopping, to represent three specific *information structures*: the original shopping list, the shopping bill and, indeed, the shopping basket.

It is important to focus on the *information* needed for this problem, and how it is *organised* and *accessed*, and not be so concerned with particular representations. Nonetheless, we will look at how we can systematically use reference language to describe, and BYOB to implement, these information structures.

3.2. Introduction

A simple list is an unordered sequence of items which is accessed by moving through the sequence from the first to the last item. We may start a new list, and then add information to the list, item by item, one after another. We may also check to see if an item is already present in the list.

Of course, with a real list one can also cross out or change items, but we won't worry about these quite yet.

So our simple list operations are:

- *create* empty list;
- *add* item to list;
- *inspect* list for presence of item;
- get *next* item from list in turn;

3.3. Shopping list 1

To make a shopping list we:

```
create a new shopping list
WHILE we can still think of items DO
 IF the item isn't already in the list THEN
  add item to list
 END IF
END WHILE
```

We can see that making this information structure involves creating an empty structure and then adding items to it.

3.4. Shopping list 2

Suppose we were making the shopping list with someone else. We might repeatedly ask them if they can think of another item, and if so to tell us what the item is. That is, we need to receive two pieces of information from the other person:

```
create a new shopping list
SEND “Any more?” TO DISPLAY
RECEIVE more FROM KEYBOARD
WHILE more DO
 SEND “Next item?” TO DISPLAY
 RECEIVE item FROM KEYBOARD
 add item to shopping list
 SEND “Any more” TO DISPLAY
 RECEIVE more FROM KEYBOARD
END WHILE
```

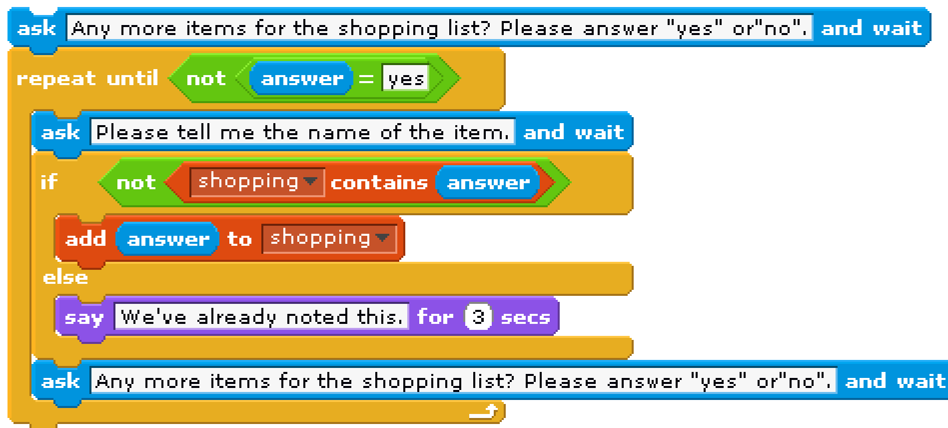
3.5. Shopping list 3

Sometimes when making up a shopping list, we think of the same thing twice, so every time we come up with a new item we should check to see if it’s on the list already:

```
create a new shopping list
SEND “Any more items for the shopping?” TO DISPLAY
RECEIVE more FROM KEYBOARD
WHILE more DO
 SEND “Next item?” TO DISPLAY
 RECEIVE item FROM KEYBOARD
 IF item not in shopping list THEN
  add item to shopping list
 ELSE
  SEND “We’ve already noted this” TO DISPLAY
 END IF
 SEND “Any more items for the shopping?” TO DISPLAY
 RECEIVE more FROM KEYBOARD
END WHILE
```

3.6. Implementing the shopping list in BYOB

Here’s the algorithm for the shopping list in BYOB:



The key decision is to implement our simple list as a BYOB list. Note that the BYOB list is much more complicated than our simple list as it also allows indexed access to insert, delete and replace entries. Here we are only using the “contains” and “add” constructs, as well as creating a new list called “shopping” outside of the script.

Note also that the BYOB is much smaller than the reference language. BYOB effectively combines requesting and receiving input into a single “ask” construct and always puts the input into “answer”. Also, BYOB uses nested constructs where we can see graphically how nesting is bounded, rather than explicit end markers.

3.7. Simplifying the algorithm?

We could also conflate these two separate pieces of information using the standard computing trick of terminating a sequence of items of unknown length with a last item:

```
create a new shopping list
SEND “Next item?” TO DISPLAY
RECEIVE item FROM KEYBOARD
WHILE item != “last” DO
 IF item not in shopping list THEN
  add item to shopping list
 ELSE
  SEND “We’ve already noted this” TO DISPLAY
 END IF
 SEND “Next item?” TO DISPLAY
 RECEIVE item FROM KEYBOARD
END WHILE
```

People who have studied programming will be familiar with this technique. However, I think that from a pedagogical perspective it is a false economy to start here as it does not correspond so immediately to a new learner’s intuitions.

And from a practical perspective, the conflation can itself be the source of problems because we are using the same representation for two different

types. Here, we have to choose an end marker item name which is very unlikely to correspond to a real item. To be fanciful, suppose we were cobblers trying to buy a new last. This is more marked when inputting sequences of numbers, as the choice of a specific value as an end marker then constrains the ranges of values which can be processed.

3.8. Shopping basket

Suppose we've got to the shop and we've forgotten the shopping list. We might walk round the shop, asking the person who's with us what we need, finding it on the shelves and putting it into the basket. Right now, we will only select one of each item. We won't worry about exactly how we find the things we need quite yet.

Here, we can reuse the algorithm for making the shopping list, by changing the name of the list and the message when we try to get something we've got already:

```
create a new basket
SEND “Any more items for the basket?” TO DISPLAY
RECEIVE more FROM KEYBOARD
WHILE more DO
 SEND “Next item?” TO DISPLAY
 RECEIVE item FROM KEYBOARD
 IF item not in basket THEN
  add item to basket
 ELSE
  SEND “We've already got this” TO DISPLAY
 END IF
 SEND “Any more items for the basket” TO DISPLAY
 RECEIVE more FROM KEYBOARD
END WHILE
```

3.9. Shopping bill

Finally, suppose that we get to the checkout and the local youth hockey team are collecting money for equipment by accepting donations in return unpacking shopping baskets. At each till, the shop assistant asks the helper for the name of each item and adds it to the bill. Unfortunately, the helpers are easily distracted and sometimes they call out the same item more than once. We won't worry about the prices of items or how shop assistants find them right now.

Here, we can again reuse the algorithm with changes to the list name and messages:

```
create a new bill
SEND “Any more items for the bill?” TO DISPLAY
RECEIVE more FROM KEYBOARD
WHILE more DO
```

```
 SEND “Next item?” TO DISPLAY
 RECEIVE item FROM KEYBOARD
 IF item not in bill THEN
  add item to bill
 ELSE
  SEND “We’ve already  charged for this” TO DISPLAY
 END IF
 SEND “Any more items for the bill” TO DISPLAY
 RECEIVE more FROM KEYBOARD
END WHILE
```

3.10. Discussion 1

We can now draw out a number of computational thinking concepts from this scenario. First of all, we haven’t used a specific programming language until the final stage of implementation. Using a reference language enabled us to successively *refine* our initial vague shopping list algorithm without making any concrete commitments to a specific language’s constructs. This has made our algorithm a bit wordier than the BYOB implementation but it should be straightforward to realise it in an arbitrary language once we’ve decided how to represent a simple list.

Secondly, because we’ve identified a common *pattern* of information structure manipulation, we’ve been able to *reuse* an algorithm that captures the manipulation. Here, reuse has been based on going in and changing algorithm text. It would more flexible to *abstract* from the algorithm as a procedure/method/function with parameters for the list and message.

3.11. Procedural abstraction

By comparing the three algorithms for constructing lists from keyboard input, we can see that they have three points of difference:

- the *list* variable to which the next item is added;
- the *request* message;
- the *error* message.

We can abstract over the algorithms by introducing a *procedure* with *formal parameters* which are used at these points of difference:

```
PROCEDURE makeSimple (list, request, error)
BEGIN PROCEDURE
 SEND [“Any more items for the”,request,”?”] TO DISPLAY
 RECEIVE more FROM KEYBOARD
 WHILE more DO
  SEND “Next item?” TO DISPLAY
  RECEIVE item FROM KEYBOARD
  IF item not in list THEN
   add item to list
  ELSE
   SEND error TO DISPLAY
```

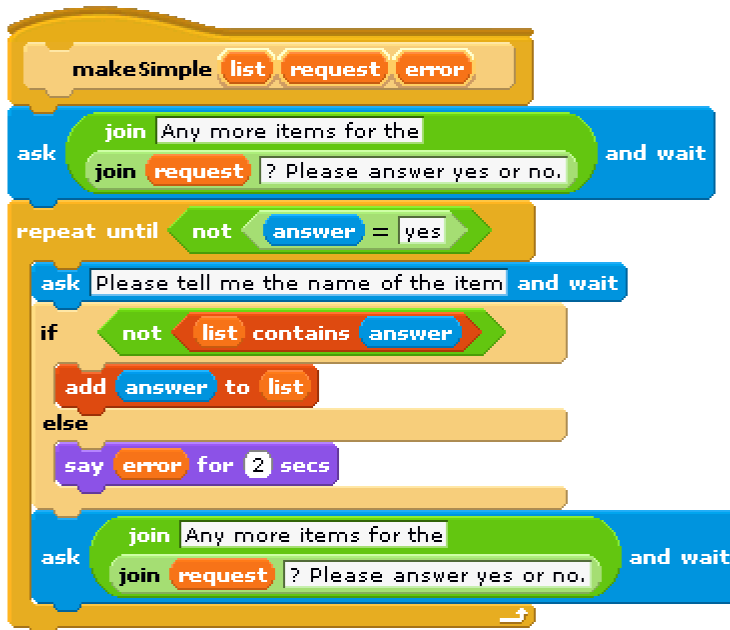
```
  END IF
  SEND ["Any more items for the",request,"?"] TO DISPLAY
  RECEIVE more FROM KEYBOARD
 END WHILE
END PROCEDURE
```

We can now use the procedure by *calling it* with *actual parameters* in place of the formal parameters:

```
create new shopping list
makeSimple
 (shopping list,"shopping?","We've noted this already")
create new basket list
makeSimple(basket list,"basket","We've got this already")
create new bill list
makeSimple
 (bill list,"bill","We've already charged for this")
```

Notice that we now create the list outside of the procedure because we want it to still exist once we've left the procedure.

3.12. Implementing the procedure in BYOB



3.13. Checking the lists

There are checks we might want to perform at each stage of shopping, to make sure that we and the shop assistant haven't made any mistakes:

a) is every item on the shopping list in the basket; that is, have we bought everything we needed?

```
FOREACH item in the shopping list DO
 IF item not in the basket THEN
  SEND [item,”missing”] TO DISPLAY
END FOREACH
```

b) is every item in the basket on the shopping list; that is, have we bought anything we don’t need?

```
FOREACH item in the basket DO
 IF item not in the shopping list THEN
  SEND [item,”not needed”] TO DISPLAY
END FOREACH
```

c) is every item in the basket on the bill; that is, have we been charged for everything we bought?

```
FOREACH item in the basket DO
 IF item not in the bill THEN
  SEND [item,”not charged”] TO DISPLAY
END FOREACH
```

d) is every item on the bill from the basket; that is, have we been charged for anything we didn’t buy?

```
FOREACH item in the bill DO
 IF item not in the basket THEN
  SEND [item,”not wanted”] TO DISPLAY
END FOREACH
```

3.14. Discussion 2

Note that to get the effect of a next operation we’ve used the pseudo-code FOREACH to implicitly change the item of consideration on each iteration. *next* is actually quite a complex operation, involving the maintenance in the background of state information about where we are in a structure. We could define *contains* in terms of FOREACH and *next*.

As with making the shopping list, basket and bill, these algorithms are all pretty much the same: check if something in one list is in another list, and if it isn’t then take some action, in our case by sending a warning message. Once again, this would be best abstracted as a procedure with parameters for the list and message.

3.15. Implementing the check in BYOB

The algorithm to see if everything in the shopping list is in the basket is, in BYOB:

BYOB does not have a FOREACH so we need to explicitly select each item from the shopping list in turn. We introduce a variable "next" to keep track of where we are in the shopping list, which we need to initialise, test for termination and increment. In a different language we might use a for loop directly.

Here, the BYOB is larger than the pseudo-code.

3.16. Exercises

1. a) write an algorithm to display a simple list on the display;
   b) implement the algorithm in BYOB for some simple list of your choice;
   c) abstract the algorithm as a procedure;
   d) implement the procedure in BYOB

2. a) change algorithm 10. a), that checks if every item in the shopping list is in the basket, to write items from the shopping list that aren't in the basket to a new simple list instead of displaying them as they are found;
   b) implement the algorithm in BYOB followed by code to display the final new list;
   c) abstract the algorithm as a procedure;
   d) implement the procedure in BYOB.

3. a) abstract over the algorithms in 10. by introducing a procedure;
   b) write down how the procedure would be called for each of the checks;
   c) implement the procedure in BYOB

Suppose we have a new operation to *drop* an item from a list.

4. a) write an algorithm to check that everything in the shopping list is in the basket and vice versa by dropping each item in the shopping list from the basket. At the end of the algorithm, we are left with a shopping list of items we didn't put in the basket and a basket of items that weren't in the shopping list.
   b) implement the algorithm in BYOB followed by code to display the final lists;
   c) abstract over the algorithm by introducing a procedure;
   d) implement the procedure in BYOB.

5. a) write an algorithm to check if a simple list contains an item, using FOREACH.

b) implement the algorithm in BYOB;
c) abstract over the algorithm by introducing a procedure;
d) implement the procedure in BYOB.

4. Dinner party

4.1. Introduction

In this chapter we're going to explore a substantial concrete problem scenario in a lot more detail, to tease out the CT stages from Chapter 1 in a bit more detail and explore their interdependencies.

We're not even going to attempt to apply CT step by step because CT doesn't work like that; rather, like all problem solving methodologies, it involves continually moving to and from stages, making and revising decisions, considering several stages at the same time, endlessly iterating in search of a fixed point.

At the end, though, we will try to rationally reconstruct what we've done in CT terms.

4.2. Where to start?

Ours scenario is a dinner party. And, we're going to invite guests and they're going to be seated at tables. So we have a first decomposition with two potential information structures:

- guest list;
- seating layout;

and some sub-problems of how to:

- construct the guest list
- characterise the seating layout;
- map guests to seats.

We have also identified guests and seats as loci of decisions, and will need to work out how they may best be characterised.

We can immediately see that all these are closely intertwined. Who we invite will be based on criteria that will also be used in part to determine where they sit. And the information in the guest list will be central to mapping guests to seats.

4.3. Guest list

Making a guest list generally involves a partially structured brain dump. We might well choose people en-masse by family or friendship group or work group or activity group.

These feel like patterns which also start to suggest information we might record about guests.

We will often invite people based on considerations such as:

- we always invite X;
- we must invite Y;
- how about inviting all the people who Z;

- if we invite A we must invite B.

And we have constraints on who we can invite. There are general constraints like the number of seating places, simple constraints like:
- C can't come on that date;

and more complicated ones like:
- if D comes then E won't come.

Let's now have a first stab at characterising guests. For each guest, we need to know:
- their name;
- why we're inviting them;
- whether there are any constraints on inviting them.

To return to the guest list, how about starting with a list of all the people who *might* be invited and then decide who we *will* and *won't* invite. Now the guest list has become a *potential* guest list, and we have decomposed again to find two more information structures:
- invited list;
- not invited list,

and another sub-problem:
- mapping potential guests to invited and not-invited guests.

The not invited list might also record:
- why someone wasn't invited.

It might seem that we don't need the not invited list as we could just delete people from the potential list. But it might be useful to be able to reconcile our invitation decisions against the original list to ensure that we haven't forgotten to consider anyone.

Also, if subsequently we find that we can invite more people, say because someone's dropped out, we might want to revisit the not invited list. We then need selection criteria which might be found through further decomposition of the guest invitation criteria, and an algorithm to select guests who will now be invited[10].

It is tempting to consider rank ordering the not invited list to ease selection. Or we might rank order the potential guest list and then select as many for the invited list as we have seats. But perhaps this is really an algorithmic consideration which we should return to later.

This is getting a bit complicated. Maybe we're being over ambitious in trying to codify all aspects of the problem, which is rapidly turning into an AI challenge. If we're really intent on building a computerised system for organising dinner parties, people might well balk at having to enter so much information.

---

[10] Uninviting people, is rare as well as unpleasant.

Suppose we don't worry about detailed guest choice criteria and decide up-front ourselves who we:
- must invite;
- might invite;
- won't invite.

So now we have an alternative decomposition. For each potential guest, we record their:
- name;
- invitation status.

Thus, we have three invitation lists:
- must;
- might;
- won't,

and a new, trivial sub-problem of:
- mapping potential guests to the status lists.

In principle, we don't even need to record potential guests' invitation statuses as that's implicit in the list their in. But it might be useful if we're shuffling guests bnetween lists to have some record of our first idea of how much we wanted to invite them:

Arguably, though, simplifying things here means that later we'll have to revisit more complex criteria for allocating guests to seats.

4.3. Seating

Let's now think about the seating arrangements. We assume that we already know the venue we're going to use. In particular, we know how many people it can seat, how many tables there are and how many people can sit at each table.

How about simply numbering all the seats, without even thinking about tables, and then systematically allocating people from the invited list, one after another. So we would have a:
- seat list

and a trivial sub-problem of:
- mapping from the invited guest list to seats.

Of course this is an awful solution which takes no account of who's sitting next to or near to who.

A better second decomposition might be to individually identify seats at tables. However, this is rather more complicated than at first sight.

Let's start with a map of the venue, showing each table with its seats in relation to other features of the venue such as the positions of the entrances

and the toilets and the bar and the buffet and the dance floor. Indeed, if we're hiring the venue, the owners might well supply us with such a floor plan.

We next need to have some way of identifying tables. We could just number them. Of course, when we're actually discussing who sits where, we will talk in terms of the physical venue layout. So we might use names like "top table" or "nearest the entrance" or "next to the buffet" for specific tables. As we know roughly who's coming and why, we might well pre-allocate tables to groups of people and then refer to the table by the group name.

We now need a way of identifying seats relative to identified tables, so we need a:

- table list

and for each table we need a:

- seat list.

Of course, for a hired venue, the owners will have some standard way of identifying seats at tables, to help them plan the service.

Anyway, we will characterise each seat by:

- table;
- seat at table[11].

We haven't yet considered the shapes of the tables. If they're circular we could number the seats going widdershins[12] from some agreed starting point. If they're rectangular then we could identify sides and then number seats starting from each corner. How then do we account for being next to or near to or opposite someone? At least let's assume that all the tables are the same size and shape.

We may well need to worry about where tables are in relation to each other, and hence whether seats at different tables are actually next to or near to each other. That is we may need some relative way of identifying seats at tables as well as an absolute way.

We're now considering patterns amongst tables and seats that capture proximity notions of:

- next to;
- opposite;
- near to.

For each seat, we could list all the seats it's next to and opposite and near to.

Or we could list all the relations between tables and tables, and seats and tables, and seats and seats, using some notation like Prolog facts or SQL relation, but that feels a bit concrete.

---

[11] Stop thinking about 2D arrays: that's an implementation decision.
[12] Scots for anti-clockwise. Of course we could go clockwise...

Or we could represent the venue as a hierarchical connected graph, with:
- nodes for tables and seats;
- top level arcs between tables which are next to each other;
- table level arcs between seats which are next to or opposite each other;
- inter-table arcs between seats which are near each other.

Or we could identify tables as coordinate positions in the venue so that can use arithmetic to determine notions of proximity. For example, if tables are in an M*N grid, then a table in coordinate X/Y is next to tables at X-1/Y, X+1/Y, X,Y-1, X, Y+1 with special cases when X /Y are 0 or M/N.

We could work out an even more elaborate scheme for numbering seats at tables so that we can calculate whether seats are opposite or next to each other. If we're really cunning then maybe we could use a numbering scheme that even lets us to calculate whether chairs at different tables are back to back. Again, though, this feels like an implementation consideration

In all three cases, we're trying to choose an information structure that captures patterns amongst relationships between seats and tables. Perhaps we should just stick with the floor plan, and a pencil and eraser...?

4.4. Mapping guests to seats

Finally, let's think about how to map guests to seats. All through the above discussion of guests and seats, we've kept coming back to criteria for inviting and seating guests, and we have tried to choose characterisations that will help us capture these criteria. That is, we've constantly considered what the end solution will look like in deciding how to analyse the problem.

We have decomposed a list of potential guests into lists of invited and not invited guests, with information about why we invited them. In so doing, we may also have captured information that is helpful for deciding how they're to be seated. Oh, maybe we started with a list of potential guests and started to explore an algorithm based on patterns of properties to partition it into invited and uninvited guests.

We have also decomposed the venue into, at simplest, a list of tables each with a list of seats. We also have some way of recording all those fiddly proximity patterns amongst tables and seats.

In discussing how to characterise the seating, we suggested and rejected just allocating people to seats willy nilly. Rules for seating include:
- put all X at same table;
- put A and B at same table (because...);
- put C and D next to each other (because...);
- put E and F opposite each other (because...);
- don't put G and H at same table (because...)
- don't put I and J next to each other (because...);
- don't put K and L opposite each other (because...);

- M doesn't mind where they sit.

These rules are not necessarily absolute. For example, "put A and B next to each other" may be more important to some A and B than others: maybe they're happy to be opposite each other or just at the same table. Similarly, "don't put C and D" at same table may not matter so much provided we "don't put C and D next to/opposite each other".

So, how to proceed? We want the people at the same table to feel comfortable in each other's company. If we have more seats than guests then it's a wee bit easier as we can always temporarily park guests at a spare table while we decide where to seat them. And we can end up with less guests than seats at individual tables.

We need to systematically work through the rules, maybe starting with those that seat a maximum number of people at once like "put all X at same table" or "put A and B at same table".

We can also use connections between rules like:

- A is an X and put A and B at same table (even if B isn't an X) so put B at the X table;

and:

- put A and B at same table and put B and C at same table so put A and C at same table.

At the same time we need to think about "don't put G and H at same table" and the connected forms like:

- put A and B at same table but don't put A and C at same table so don't put B and C at same table

and:

- A is an X and put A and B at same table but don't put B and C at same table so either don't put C on table X or don't put A and B on table X

We go on applying the rules to find initial seats for guests and then move them between tables. Eventually we'll reach a more or less satisfactory fixed point where all the guests have seats and we've minimised putting incompatible guests near to each other[13].

4.5. Reflection

Organising a dinner party is a really involved business. Working out the guest lists is fairly straightforward but allocating guests to seats can be really fiddly. Indeed, this is an example of a constraint satisfaction problem which gets more and more complicated as the numbers of guests and seating rules increase.

It looks like this isn't a very good problem for teaching Computing because it doesn't end up with a neat algorithmic solution for which we can cut code.

[13] Or we'll decide to cancel the dinner party.

However, I think that's why it's a really good basis for thinking about how CT works.

And, yes, school students don't hold dinner parties. Teachers do though and this material is aimed at teachers. Anyway, all the above applies equally well to a birthday party. Or a wedding reception. Or a retirement dinner.

Let's think about what we've done in CT terms. We've used:

- decomposition to identify information structures and sub-problems, and by implication broad algorithmic requirements;
- abstraction to identify information in structures and rules for solving sub-problems, and by implication finer algorithmic detail;
- patterns in information to explore alternative information structures and algorithms;

We've iterated amongst these stages in a messy sort of way. What's above really is a structured brain dump: I've tried not to go back and polish the argument to present a perfect solution. I think it's important for you to see that I'm not entirely sure what I'm doing when I'm trying to solve a brand new problem, just as I think it's important for your students to understand this about you.

4.6. Exercises

Suppose we have a list of potential guests P:

```
[[Ann,1],
 [Bob,2],
 [Carol,3],
 [Dave,4],
 [Ena,2],
 [Fred,1],
 [Gail,5]]
```

so:

P[i] ➔ ith guest
P[i][1] ➔ name of ith guest
P[i][2] ➔ priority of ith guest – 1 == high & 5 == low

1. Write pseudocode functions to:
    a. find the priority for a named guest;
    b. change the priority for a named guest;
    c. check if one named guest has higher priority than another named guest;

2. Write a pseudocode algorithm to allocate each guest to one of the lists I(nvite), M(aybe invite) and D(on't invite). You'll need to decide how to map priorities into decisions e.g. 1 == invite; 2-3 == maybe; 4-5 == don't.

Suppose we have S seats.

3. Write a pseudocode algorithm which moves guests from the maybe invite list to the invite list until there are as many invited guests as seats. If when all the maybe guests have been invited there are still seats free, then move guests from the don’t invite list. You will need to decide how to prioritise movements e.g. with the above example priorities, move 2 before 3 and 4 before 5.

4. In what circumstances might this priority system be unfair?

5. Pattern identification and abstraction

5.1. Introduction

We started by observing that computational thinking is based on:

- decomposition;
- pattern identification ;
- pattern generalisation/abstraction;
- algorithm design.

In the last chapter, we saw how every time we make a decomposition choice, we are, by implication, identifying sub-problems which will require associated algorithms to solve them. That is, decomposition and algorithm design are very strongly connected. As we'll see in this chapter, pattern identification and generalisation/ abstraction are also strongly linked, that is, we use pattern identification to find opportunities for abstractions.

Let's just remind ourselves why we like abstractions. First of all, they can make algorithms and hence programs more *succinc*t. It's important to note that succinct is not necessarily the same as *efficient* in computer terms: typically, many abstractions make little or no difference to the final code efficiency because compilers use cunning tricks like unfolding sub-program calls and unrolling loops in search for optimisations. Thus abstractions help make programming *more efficient for humans* by:

- encouraging the construction and use of general reusable artefacts;
- making programs smaller, more elegant and hence easier to read and understand, though this is a somewhat subjective judgement.

5.1. Patterns, structures and abstraction

It is usual to think about pattern identification as focusing on looking for patterns in data. However, as we're about to explore, pattern identification is a key technique for generalising computational structures as well as information structures.

I think that pattern identification is about finding *structural similarities and differences between instances* and abstraction is about *generalising through naming.* There are several ways we can do this.

First of all, if we compare instances of constructs and find structural similarities, then we can abstract over the common structure by introducing a name to stand for it. We can then use the name instead of the shared structure.

Secondly, if there are structural similarities then there must also be *structural differences*[14]. Then, we can abstract over the structural differences by *generalising* them, again by *introducing names.*

[14] unless the instances we're comparing are identical...

Now, if we've named an abstracted structure in which we've found differences, then we can *parameterise* it using the names introduced to generalise the structural differences. Thus, we can invoke the named structure with values for the parameters to create concrete instances.

Finally, if we find repeated instances of constructs with structural differences, we can again abstract by introducing names. If we can then identify some principle behind the repetition in terms of the name, then we can potentially introduce some form of iteration to generate values for the abstracted names to recapitulate the repeated instances.

So, this all sounds suspiciously like how to introduce sub-programs. And indeed it is. But it's also about how to introduce data structures and objects.

5.2. Functional abstraction

Let's start by looking at the process of finding similarities and differences in more detail. We'll do so by thinking about how to abstract functions from expressions, by naming both common structures and differences.

Suppose we want to add 1 to 3, and 1 to 127:

```
SET a TO 3+1;
SET a TO 127+1
```

These are have the common structure: ?+1. The point of difference is marked with a "?".

So let's:

- name an abstraction from this common sub-structure *inc*;
- replace the point of difference with a variable *n*: n+1;
- and parameterise the named sub structure:

```
FUNCTION inc(n)
RETURN n+1;
END FUNCTION
```

We can then instantiate the abstraction with concrete values:

```
SET a TO inc(3);
SET b TO inc(127)
```

Suppose we want to join "Happy" to "Birthday" with a space in between, and "Happy" to "New Year" with a space in between:

```
SET greeting1 TO "Happy"&" "&"Birthday";
SET greeting2 TO "Happy"&" "&"New Year"
```

These have the common sub-structure: "Happy"&" "&?. So let's:

- name an abstraction for the common substructure *happy*;
- replace the point of difference with a variable *event*: “Happy”&” “&event
- and parameterise the named substructure:

```
FUNCTION happy(event)
RETURN “Happy”&” “&event;
END FUNCTION
```

We can now instantiate this abstraction with concrete values:

```
SET greeting1 TO happy(“Birthday”);
SET greeting2 TO happy(“New Year”);
```

Suppose we’ve also made an abstraction for generalising messages like “Good Morning” and “Good Evening”:

```
FUNCTION good(time)
RETURN “Good”&” “&time;
END FUNCTION
```

Now, let’s compare these abstractions. First of all, they have different names, so any common abstraction will need a new one. They both put a space in between two strings so let’s call the new abstraction *space*.

Both abstractions have single parameters so the new common abstraction will also need a new one. This parameter is used as the second string so let’s call it *string2*.

So, using this new parameter in both original abstractions:

```
FUNCTION happy(string2)
RETURN “Happy”&” “&string2;
END FUNCTION

FUNCTION good(string2)
RETURN “Good”&” “&string2;
END FUNCTION
```

we can see that they have common structure: ?&” “&string2. So let’s:
- replace the point of difference with a parameter *string1*: string1&” “&string2
- and further parameterise the new named abstraction:

```
FUNCTION space(string1,string2)
RETURN string1&” “&string2;
END FUNCTION
```

5.3. Array and iteration abstractions

Let’s now look at introducing arrays as an abstraction from variables. We’ll also see how to introduce iteration as an abstraction from:

- sequences of assignment to array elements;
- arithmetic on sequences of array elements.

Suppose we've run a questionnaire on whether people like programming, with simple Yes/No options. We want to count how many people chose each option, and find the total number of responses and the percentages for each response.

We'll start both counts at 0. Then, so long as there are more values to input, we'll increment the appropriate count. We will assume that inputs are always either yes or no:

```
SET yes TO 0;
SET no TO 0;
WHILE more inputs DO
 IF next input is yes THEN
   SET yes TO yes+1
 ELSE
   SET no TO no+1
 END IF
END WHILE;
SET total TO yes+no;
SET yesP TO yes*100/total;
set noP TO 100-yesP
```

Now, let's generalise thus a wee bit by asking for a rating for how much people like programming on a Lickert scale:
0 – not at all
1 – not much
2 – OK
3 – much
4 – very much

First of all we need five counts and we need to set them all to 0:

```
SET notAtAll TO 0;
SET notMuch TO 0;
SET OK TO 0;
SET much TO 0;
SET veryMuch TO 0;
```

Suppose we want to abstract over this. We can see that there's a pattern:

```
SET ? TO 0;
```

So we could introduce a parameterised named procedural abstraction:

```
PROCEDURE zero(i)
SET i To 0;
END PROCEDURE
```

and then write:

```
zero(notAtAll);
zero(notMuch);
...
zero(veryMuch);
```

This is a very poor choice of abstraction which, if anything, is more obscure than the original. We've spotted the wrong difference: the pattern here is not in the computation but in the same treatment of a group of variables.

Suppose, we rename the variables with a common name and qualifier:

```
SET L0 TO 0;
SET L1 TO 0;
SET L2 TO 0;
SET L3 TO 0;
SET L4 TO 0
```

We've lost clarity as the names no longer tell us what the variables represent. Nonetheless, we can now see that there's a pattern:

```
SET L? TO 0
```

inviting us to abstract over the name qualifier.

While most programming languages won't let us do this, pretty well all have the notion of an array as an indexed sequence of variables with a common name. So let's introduce an array variable *L* with five elements, and use number indices on this common variable identifier instead of numbered qualifiers on a common name:

```
SET L[0] TO 0;
SET L[1] TO 0;
SET L[2] TO 0;
SET L[3] TO 0;
SET L[4] TO 0
```

Note that we've started at 0 rather than 1.

Now we can identify the pattern difference in the indices:

```
SET L[?] TO 0
```

At the same time, we can identify a sequence pattern in the indices: 0 1 2 3 4.

So this is an invitation to abstract with a repetition rather than a procedure:

```
FOR i FROM 0 TO 4 DO
```

```
 SET L[i] TO 0
END FOR
```

Let’s now think about counting each preference. The generalisation of our yes/no example, assuming inputs are always integers between 1 and 5, is:

```
WHILE more inputs DO
 IF next input is 0 THEN
   SET L[0] TO L[0]+1
 ELSE
 IF next input is 1 THEN
   SET L[1] TO L[1]+1
 ELSE
 ...
 ELSE
  SET L[4] TO L[4]+1
 END IF
END WHILE;
```

Looking at this ungainly chain of IFs, we can see:

- a difference pattern in the assignment: SET L[?] TO L[?]+1
- a sequence pattern in the assignment array indices: 0 1 2 3 4
- a difference pattern in the test: IF next input is ?
- a sequence pattern in the test difference: 0 1 2 3 4

We can now see that each element of the test difference sequence pattern is the same as the corresponding element of the assignment difference sequence pattern, that is whenever the test checks for *next* , the assignment modifies the array at index *next*.

Thus we can replace the IF chain with the succinct:

```
WHILE more inputs DO
 SET L[next] TO L[next]+1
END WHILE
```

This is both more succinct and more efficient code as we have eliminated a repeated chain of tests.

Next, let’s find the total.

```
SET total TO L[0]+L[1]+L[2]+L[3]+L[4]
```

This looks pretty succinct but if we added more questions we’d have to add more terms to the sum. Instead, let’s try finding a pattern by unwinding and generalising.

We could start the total off at 0 and then add each count in turn:

```
SET total TO 0;
```

```
SET total TO total+L[0];
SET total TO total+L[1];
...
SET total to tota+L[4]
```

Again, we spot the difference pattern:

```
SET total TO total+L[?]
```

identify the sequence pattern in the indices:0 1 2 3 4
and abstract with a repetition:

```
SET total TO 0;
FOR i FROM 0 TO 4 DO
 SET total TO total+L[i]
END FOR
```

While this is more succinct than the original, without optimisation, it's less efficient as we are repeatedly accessing and changing *total*. Still, this is far more general, and, like all our abstractions, can easily handle different number of cases in our questionnaire answers.

Finally, let's find the percentages. Before, we had variables for the yes and no percentages. Now, let's have an array *LP* of five elements:

```
SET LP[0] TO L[0]*100/total;
SET LP[1] TO L[1]*100/total
...
SET LP[4] TO L[4]*100/total
```

Yet again, we spot the difference pattern:

```
SET LP[?] TO L[?]*100/total
```

identify the sequence pattern in the indices:0 1 2 3 4
and abstract with a repetition:

```
FOR i FROM 0 TO 4 DO
 SET LP[i] TO L[i]*100/total
END FOR
```

5.4. Reflection

We have seen four techniques for identifying patterns with associated abstractions. They all share looking for differences in common structures:

- no differences – abstract as unparameterised function or procedure;
- differences but no sequences – abstract as parameterisied function or procedure;
- differences with assignment sequences – abstract as iteration with variable for difference;

- differences with expression sequences – abstract as iteration with variable for difference.

We've also seen that abstractions can make algorithms less as well as more efficient.

To identify the patterns, we've always started from a fully decomposed form; that is from a form where we only have individual instances, with no preconceptions about generalisations. Doing it blow by blow like this may seem like a waste of time: surely it's obvious here that we need arrays and iterations.

However, I think that it takes lots of experience to be able to generalise appropriately right from the beginning of solving a problem, and, for teaching, doing so makes it hard to understand retrospectively why particular choices have been made.

5.5. Exercises

1. Why did we introduce arrays and iterations for 5 choices but not for 2? How about for 3 or 4 choices?

2. Why would keeping a running count for the total in the input loop be less efficient than finding the total at the end by summing the counts?

3. Why can't we combine the loops to find the final total and then find the percentages?

6. Recursion

6.1. Introduction

Recursion is often thought of as a scary, advanced topic. It's rarely covered at school level. Even in first level undergraduate text books on imperative programming, it's often only treated briefly in a later chapter.

Certainly, many people who are well used to iteration seem to find recursion somehow unnatural. Part of the difficulty lies in the misleading conception that recursion involves defining something in terms of itself. While this is not wholly inaccurate, it does give the impression that recursion is some sort of trick: isn't defining something in terms of itself a tautology? Mutterings about the alleged ease of proving properties of recursive constructs in functional languages further contribute to the mystique.

Another source of recursion's poor reputation is the poor choice of examples, typically numerical, through which it is often introduced. Indeed, I suspect that many students come to believe that the greatest common divisor, and the factorial and Fibonacci sequences, were dreamed up solely to illustrate an otherwise pointless technique.

Anyway, I said above that every recursive form has an iterative equivalent. So, if there's a sequence of actions which are all pretty much the same, then why not just iterate through the sequence from first to last?

Recursion is really straightforward. A recursive sub-program is one that calls itself. That's all.

The big advantage of recursion is that it can often be far more succinct than the equivalent iteration. With recursion, lots of detail to do with remembering partial results in explicit auxiliary data structures can be hidden: consider explaining tree traversal or even Fibonacci as an iteration.

I think that recursion can come to feel even more natural than iteration, once we get our heads round how to spot the appropriate patterns and introduce the corresponding abstractions.

6.2. Recursive function pattern and abstraction

In chapter 5, we saw an iteration to find the total count from a sequence of sub-counts. We did this by finding a pattern in a sequence which ranged from a first value to a last value in a series of equal steps. And we then used an iteration abstraction:

```
FOR i FROM 0 to 4 DO
 SET total TO total+L[i]
```

A key step was identifying the initialisation to precede the iteration:

SET total to 0

Why 0? Well, we need to start with a value that won't affect the rest of the computation. here the computation is based on repeated addition and 0 is an identity for addition:

0+X = X.

We would still need that initialisation even if the sequence of sub-counts were empty.

Bearing that in brain, let's think again about the original computation:

SET total TO L[0]+L[1]+L[2]+L[3]+L[4]

and look at how to derive a recursive form. First we'll make the initialisation explicit:

SET total TO 0+L[0]+L[1]+L[2]+L[3]+L[4]

Now we'll bracket the largest left-most sub-expression:

SET total TO (0+L[0]+L[1]+L[2]+L[3])+L[4]

so we can see that summing the first 5 in L is like summing the first 4 in L and adding L[4].

Let's bracket the next largest left-most sub-expression:

SET total TO (*(0+L[0]+L[1]+L[2])*+L[3])+L[4]

Thus, summing the first 4 in L is like summing the first 3 in L and adding L[3].

Bracketing again:

SET total TO ((*(0+L[0]+L[1])*+L[2])+L[3])+L[4]

Thus, summing to the first 3 in L is like summing the first 2 in L and adding L[2].

And again:

SET total TO (((*(0+L[0])*+L[1])+L[2])+L[3])+L[4]

Thus, summing the first 2 in L is like summing the first 1 in L and adding L[1].

Once more:

SET total TO ((((*(0)*+L[0])+L[1])+L[2])+L[3])+L[4]

Thus, summing the first 1in L is like summing the first 0 in L and adding L[0].

Finally, summing the first 0 in L is 0. That's the initialisation from the original iteration.

We can see a *nested pattern* of *doing something to everything in the sub-sequence* until the sequence is empty:

(?+L[4]); (?+L[3]) (?+L[2]); (?+L[1]); (?+L[0])

If we abstract for the index pattern: 4 3 2 1 0
with n, then the general case is:

?(n-1)+L[n]

that is, summing the first n in L is like summing the first n-1 in L and then adding L[n].

So, we've found a *recursive function pattern* in an expression consisting of a sequence of the same operations where any sub-sequence from the start of the range has the same structure as the whole sequence.

We can now introduce a *recursive function abstraction* by:

- naming the operation sequence: sum;
- abstracting over the range values, with a parameter: n for 5 4 3 2 1;
- abstracting over any other variables, with parameters: A for L:

```
FUNCTION sum(A,n)
...
END FUNCTION
```

The function body is built from an IF...THEN...ELSE. For the base case, beyond the start of the range, the initialisation value is returned. Otherwise, in the recursion case, the function is called with a decremented range value to give a result for the rest of the range. The operator is then applied to that result and the end of range value.

```
FUNCTION sum(A,n)
  IF n<0 THEN
   RETURN 0;
  END IF
  RETURN sum(A,n-1)+A[n]
END FUNCTION
```

We can now call this function instead of the original expression, with the end of range value as actual parameter:

```
SET count TO sum(L,4)
```

Thus:

sum(L,4) ➔ sum(L,43)+L(4) ➔ (sum(L,2)+L[3])+L[4] ➔
((sum(L,1)+L[2])+L[3]+L[4] ➔ (((sum(L,0)+L[1])+L[2])+L[3])+L[4] ➔
((((sum(L,-1)+L[0])+L[1])+L[2])+L[3])+L[4] ➔
((((0)+L[1])+L[2])+L[3])+L[4]

which was our bracketed original expression.

6.3. Recursive procedure pattern and abstraction

Not yet convinced...? Next, let's look again at initialising the counts, which we first abstracted with the iteration:

```
FOR i FROM 0 TO 4 DO
  SET L[i] TO 0
END FOR
```

We could have used a recursive procedure here again.

Starting with the original:

```
SET L[0] TO 0;
SET L[1] TO 0;
SET L[2] TO 0;
SET L[3] TO 0;
SET L[4] TO 0
```

let's bracket using BEGIN...END:

```
BEGIN
    SET L[0] TO 0;
    SET L[1] TO 0;
    SET L[2] TO 0;
    SET L[3] TO 0;
END
SET L[4] TO 0
```

so setting the first 5 to 0 is like setting the first 4 to 0 and then zeroing the 5th.

Bracketing again we get:

```
BEGIN
     BEGIN
         SET L[0] TO 0;
         SET L[1] TO 0;
         SET L[2] TO 0
    END
    SET L[3] TO 0
```

```
END
SET L[4] TO 0
```

so setting the first 4 to 0 is like setting the first 3 to 0 and then zeroing the 4th.

Bracketing again we get:

```
BEGIN
    BEGIN
        BEGIN
            SET L[0] TO 0;
            SET L[1] TO 0
        END
        SET L[2] TO 0
    END
    SET L[3] TO 0
END
SET L[4] TO 0
```

so setting the first 3 to 0 is like setting the first 2 to 0 and then zeroing the 3rd.

Bracketing again we get:

```
BEGIN
    BEGIN
        BEGIN
            BEGIN
                SET L[0] TO 0
            END
            SET L[1] TO 0
        END
        SET L[2] TO 0
    END
    SET L[3] TO 0
END
SET L[4] TO 0
```

so setting the first 2 to 0 is like setting the first 1 to 0 and then zeroing the 2nd.

Bracketing again we get:

```
BEGIN
    BEGIN
        BEGIN
            BEGIN
                BEGIN
                    do nothing
                END
                SET L[0] TO 0
            END
```

```
            SET L[1] TO 0
          END
         SET L[2] TO 0
    END
    SET L[3] TO 0
END
SET L[4] TO 0
```

so setting the first 0 to 0 is like doing nothing and then zeroing the 1st.

We can again see a *nested pattern* of *doing something with everything in the sub-sequence* until the sequence is empty. And again we can see a difference pattern:

```
BEGIN ? END SET L[4] TO 0
BEGIN ? END SET L[3] TO 0
BEGIN ? END SET L[2] TO 0
BEGIN ? END SET L[1] TO 0
BEGIN ? END SET L[0] TO 0
```

If we abstract for the index pattern: 4 3 2 1 0
with n, then the general case is:

```
BEGIN ?(n-1) END SET total TO total+L[n]
```

So, we've found a *recursive procedure pattern* in a command sequence consisting of a sequence of the same commands where any sub-sequence from the start of the range has the same structure as the whole sequence.

We can now introduce a *recursive procedure abstraction* by:

- naming the operation sequence: initialise;
- abstracting over the range values, with a parameter: n for 5 4 3 2 1;
- abstracting over any other variables, with parameters: A for L:

```
PROCEDURE initialise(A,n)
...
END PROCEDURE
```

The procedure body is built from an IF...THEN...ELSE. For the base case, beyond the start of the range, nothing is done so the procedure simply returns. Otherwise, in the recursion case, the procedure is called with a decremented range value to do the commands for the rest of the range. The command is then applied to the end of range value.

```
PROCEDURE initialise(A,n)
  IF n<0 THEN
   RETURN;
  END IF
  initialise(A,n-1);
  SET A[n] TO 0
```

END PROCEDURE

We can now call this procedure instead of the original command sequence, with the end of range value as actual parameter:

initialise(A,4);

This expands as:

initialise(A,4) ➔

initialise(A,3);
SET L[4] TO 0 ➔

initialise(A,2);
SET L[3] TO 0;
SET L[4] TO 0; ➔

initialise(A,1);
SET L[2] TO 0;
SET L[3] TO 0;
SET L[4] TO 0; ➔

initialise(A,0);
SET L[1] TO 0;
SET L[2] TO 0;
SET L[3] TO 0;
SET L[4] TO 0; ➔

initialise(A,-1);
SET L[0] TO 0;
SET L[1] TO 0;
SET L[2] TO 0;
SET L[3] TO 0;
SET L[4] TO 0; ➔

SET L[0] TO 0;
SET L[1] TO 0;
SET L[2] TO 0;
SET L[3] TO 0;
SET L[4] TO 0;

6.4. Recursive songs

As mentioned in Chapter 1, I've used counting songs to introduce recursion with 1st year undergraduate classes meeting programming for the first time. Let's now think about three such songs.

"10 Green Bottles" is a well known English-language children's song. Here, though, we'll look at the more succinct "3 Green Bottles":

3 green bottles, hanging on the wall.
3 green bottles hanging on the wall.
And if 1 green bottle should accidentally fall,
There'd be 2 green bottles, hanging on the wall.

2 green bottles, hanging on the wall.
2 green bottles hanging on the wall.
And if 1 green bottle should accidentally fall,
There'd be 1 green bottles[15], hanging on the wall.

1 green bottles, hanging on the wall.
1 green bottles hanging on the wall.
And if 1 green bottle, should accidentally fall,
There'd be 0 green bottles hanging on the wall.

First, we can see a pattern in the verse where if lines 1 & 2 refer to n bottles then line 4 refers to n-1 bottles:

```
PROCEDURE verse(n)
   SEND n&” green bottles, hanging on the wall.\n”  TO DISPLAY;
   SEND n&” green bottles, hanging on the wall.\n”  TO DISPLAY;
   SEND “And if one green bottle should accidentally fall\n” TO DISPLAY
   SEND “There’d be “&n-1&” green bottles, hanging on the wall.\n”
     TO DISPLAY
END PROCEDURE
```

Then we can see a nested pattern in the sequence of verses, where to sing the song about n bottles, we sing a verse for n bottles and then sing the song about n-1 bottles:

```
PROCEDURE bottles(n)
   IF n=0 THEN
    RETURN;
   END IF
   verse(n);
   SEND “\n” TO DISPLAY;
   bottles(n-1);
END PROCEDURE
```

We call:

```
bottles(10)
```

for the full song.

[15] yes, the singular of bottles is bottle...

When I show this to students before they've ever seen iteration, they find it easy to understand that if procedures can call other procedures, then they can call themselves.

Next, let's consider the irritatingly non-terminating song "1 man went to mow", redolent for me of excessively long car journeys as a child:

1 man went to mow, went to mow a meadow.
1 man and his dog, went to mow a meadow.

2 men went to mow, went to mow a meadow.
2 men
1 man and his dog, went to mow a meadow.

3 men went to mow, went to mow a meadow.
3 men
2 men
1 man and his dog, went to mow a meadow.

etc

We can see that each verse contains a sub-verse that counts down, a bit like for "10 Green Bottles":

```
PROCEDURE men(n)
  IF n=1 THEN
   SEND "1 man and his dog, went to mow a meadow\n" TO DISPLAY
   RETURN
  END IF
  SEND n&" men\n" TO DISPLAY;
  men(n-1);
END PROCEDURE
```

Then the song is a sequence of a verse for n men and the song for n+1 men:

```
PROCEDURE mow(n)
   men(n);
   SEND "\n" TO DISPLAY;
   mow(n+1)
END PROCEDURE
```

So, we call:

```
mow(1);
```

to start the song.

Finally, let's try the carol "The 12 days of Christmas":

On the first day of Christmas, my true love sent to me:

A partridge in a pear tree.

On the second day of Christmas, my true love sent to me:
Two turtle doves and
A partridge in a pear tree.

On the third day of Christmas, my true love sent to me:
Three French hens
Two turtle doves and
A partridge in a pear tree.

On the twelfth day, the presents are:

- Twelve drummers drumming
- Eleven pipers piping
- Ten lords a-leaping
- Nine ladies dancing
- Eight maids a-milking
- Seven swans a-swimming
- Six geese a-laying
- Five golden rings
- Four calling birds
- Three French hens
- Two turtle doves
- A partridge in a pear tree

We seem to have the same verse structure as in “10 men went to mow” but there is no discernable pattern in the presents. So we cannot simply generate the descending sequence of presents as integer counts slotted into a common line template.

Also, for the song itself, each day has a different name:

first second third ... twelfth

Time for some computational thinking...

Suppose we made an information structure for the presents:

```
SET presents TO
 [“A partridge in a pear tree”,
  “Two turtle doves”,
  ...
  “Twelve drummers drumming”];
```

We could also make an information structure for the days:

```
SET days TO [“first”,”second”,...”twelfth”];
```

Now we can again use the descending sequence recursion. Days count from 12 to 1 but arrays are indexed down to 0, so for day n we'll need to access present n-1 on day n:

```
PROCEDURE day(n)
  IF n=0 THEN
   RETURN;
  END IF
  SEND presents[n-1]&"\n";
  day(n-1)
END PROCEDURE
```

For the whole song, we'll use the ascending sequence recursion again but stopping after day 12. As for presents, on day n the name is in days n-1:

```
PROCEDURE christmas(n)
  IF n>12 THEN
   RETURN;
  END IF
  SEND "On the "&days[n-1]&" day of Christmas, my true love sent to me:
  day(n);
  christmas(n+1)
END PROCEDURE
```

and call:

```
christmas(1);
```

6.5. Non-terminating recursion

What about writing recursive programs that don't terminate? We meet lots of these, well lots that we would prefer not to terminate, like our computer and table smart phone[16] operating systems. More seriously, we would really like heart pace makers and other medical devices not to stop unless we explicitly want them to.

With iteration, non-termination takes the form:

```
WHILE TRUE DO
  <something>
```

Non-terminating recursion is equally easy:

```
PROCEDURE forever()
<something>
forever()
END PROCEDURE
```

[16] I wonder how soon we'll drop the "smart"?

We just call the procedure without any test for a base case.

We can find non-termination in popular culture. There's a famous advert for salt which shows a wee boy chasing hen with a box of salt which shows a wee boy chasing a hen with a box of salt which shows a wee boy chasing a hen with a box of salt which shows a wee boy, and so on. This is a nice example, because it captures the nested[17] nature of recursion: each picture is inside another picture.

Finally, many cultures share a non-terminating recursive folk tale where a magical entity offers someone three wishes. They ask for health, and happiness, and for three more wishes. So long as they always remember to ask for more wishes with the third wish, they'll never run out.

6.6. Reflection

We have seen two new patterns:

- nested expression sequence;
- nested command sequence;

and explored how to use them to find the corresponding abstractions:

- recursive function;
- recursive procedure.

As in the previous chapter, we started with a fully decomposed form but we then exposed nesting using bracketing:

- (...) for expression sequences;
- BEGIN...END for command sequences.

When there was nothing more to bracket, we had found what became the base case for the recursion. The generalisation of the difference and sequence patterns then gave us the recursion case.

I hope that this has also given some flavour of the equivalence between recursion and iteration.

If you still don't feel comfortable with recursion or don't find this convincing then please tell me why.

6.6. Exercises

1. Derive recursive procedures for:

```
i) FOR i FROM 0 TO 4 DO
   SET LP[i] TO L[i]*100/total
  END FOR
```

```
ii) WHILE more inputs DO
```

[17] boom boom...

```
  SET L[next] TO L[next]+1
 END WHILE
```

Hint: for the base case, do nothing if there are no more inputs.

2. Derive a recursive procedure from:

```
SEND LP[0] TO DISPLAY
SEND LP[1] TO DISPLAY
SEND LP[2] TO DISPLAY
SEND LP[3] TO DISPLAY
SEND LP[4] TO DISPLAY
```

3. Derive iterative functions for:

```
i) FUNCTION fib(n)
    IF n=0 || n=1 THEN
     RETURN 1;
    END IF
   RETURN fib(n-1)+fib(n-2)
 END FUNCTION
```

```
ii) FUNCTION gcd(x,y)
     IF y=0 THEN
      RETURN x;
     END IF
     RETURN gcd(y,x%y)
    END FUNCTION
```

4. Change the bottles procedure to output “bottle” for 1 bottle and “bottles” otherwise.

5. Change the mow procedure to terminate after m men.

7. Sequences and patterns

7.1. Introduction

In teaching how to find algorithms from patterns in information, it is often tempting to start with sequences with known properties. We're going to now look at some concrete examples to tease out two approaches.

First of all, suppose we want to find the first N odd integers. We can write them down:

1 3 5 7 9 11...

and think about how each element is related to the next. We can spot that, apart from the initial 1:

3 = 1+2
5 = 3+2
7 + 5+2
9 = 7+2
11 = 9+2

so each element is 2 more than the previous element. If we name the elements:

O_1 O_2 O_3...

then:

$O_{i+1} = O_i+2$

We can turn this into an algorithm with an accumulator variable O that will take on the values of successive elements. We use:

$O_1 = 1$

to initialise the variable:

```
SET O TO 1
```

and a FOR loop to find each element in turn:

```
FOR I FROM 2 TO N
 SEND O TO DISPLAY
 SET O TO O+2
END FOR
```

We could now generalise this to construct an iterative function to find the Nth odd integer. We won't bother to output the intermediate values and will just return the final value:

```
FUNCTION ODD(N)
 SET O TO 1
 FOR I FROM 2 TO N DO
   SET O TO O+2
 END FOR
END FUNCTION
```

Incidentally, if we look at this a bit more closely, we can see that we have actually implemented a close relation of multiplication by repeated addition. We've added 2 to 1, N-1 times. So we could simplify this to:

```
FUNCTION ODD(N)
 RETURN 1+2*(N-1)
END FUNCTION
```

or even:

```
FUNCTION ODD(N)
 RETURN 2*N-1
END FUNCTION
```

because:

1+2*(N-1) == 1+2*N-2 == 2*N-1

7.2. Powers of 2

Next, suppose we want to find the first N powers of 2. We write down:

2 4 8 16 32...

We can see that, apart from the initial 1, we have:

4 = 2*2
8 = 2*4
16 = 2*8
32 = 2*16
...

so each element in the sequence is double the previous element. If we named the original sequence of elements:

$P2_1$ $P2_2$ $P2_3$ ...

then the abstracted pattern is:

$P2_1 = 2$
$P2_{i+1} = 2*P2_i$

So, our algorithm might be:

```
SET P2 TO 2
FOR 2 FROM 2 TO N
  SEND P2 TO DISPLAY
  SET P2 TO 2*P2
END FOR
```

As we odd integers, we can realise finding the Nth power of 2 as an iterative algorithm, by dropping the output and returning the final value:

```
FUNCTION POW2(N)
 SET P2 TO 2
 FOR I FROM 2 TO N DO
   SET P2 TO 2*P2
 END FOR
 RETURN P2
END FUNCTION
```

Now, we could have just started from a formal definition of power of 2:

$2^1 = 2$
$2^N = 2*2^{N-1}$

without looking at the sequence in any detail, to construct a recursive algorithm:

```
FUNCTION POW2(N)
 IF N= 1 THEN
  RETURN 2
 ELSE
   RETURN 2*POW2(N-1)
 END IF
END FUNCTION
```

But then all we'd have learnt would have been how to turn a recursive formula into a recursive algorithm, rather than how to derive the algorithm from the information.

7.3. Squares

Let's now look at a case where finding a relationship between members of a sequence doesn't seem to work so well. Suppose we want to find the first N squares:

1 4 9 16 25 36...

Can we find a pattern? There's no obvious summing or multiplication. How about looking at the literal difference:

4-1 = 3
9-4 = 5
16-9 = 7
25-16 = 9
36-25 = 11
...

or:

4 = 1+3
9 = 4+5
16 = 9+7
25 = 16+9
36 = 25+11
...

On the right hand side, we can see the original sequence:

1 4 9 16 25 36...

and a familiar sequence:

3 5 7 9 11...

So the i+1th square is the ith square plus the i+1th odd integer. We know that the i+1th odd integer is 2 more than the ith odd integer. If the squares are:

S_1 S_2 S_3 ...

then:

$S_1 = 1$
$S_{i+1} = S_i+O_{i+1}$

Thus, an algorithm to generate squares is:

```
SET S TO 1
SET O TO 3
FOR I FROM 1 TO N
 SEND S TO DISPLAY
 SET S TO S+O
 SET O TO O+2
END FOR
```

This does seem a wee bit opaque for what is really quite a simple problem[18].

[18] Curiously, though, the accumulation of successive differences was the basis of Charles Babbage's revolutionary mid-Victorian Difference Engine, an intricate mechanical device which was intended to generate mathematical table.

7.4. Generation from sequence indices

Let's start again. We might observe that we have:

1 = 1*1
4 = 2*2
9 = 3*3
16 = 4*4
25 = 5*5
...

and we can line the sequence up against indices as:

index: 1 2 3 4 5 ...
square: 1 4 9 16 25...

so the ith square is the ith sequence index times itself. This cries out for the pleasingly simple:

```
FOR I FROM 1 TO N
 SEND I*I TO DISPLAY
END FOR
```

The algorithm involves a FOR loop to generate the indices and an expression to find the sequence elements.

We now have a second strategy for finding patterns in sequences. As well as looking at the elements themselves, think about the relationship between the elements and their indices.

Let's apply this to finding odd integers:

index: 1 2 3 4 5 6...
element: 1 3 5 7 9 11...

If we start with the indices, we can see that:

3 = 2*2-1
5 = 2*3-1
7 = 2*4-1
9 = 2*5-1
...

so the ith element is 2 times the index minus 1:

$O_i = 2*i-1$

The algorithm is now:

```
FOR I FROM 1 TO N DO
 SEND 2*I-1 TO DISPLAY
```

```
END FOR
```

We also found this by thinking about our original:

```
SET O TO 1
FOR I FROM 1 TO N
 SEND O TO DISPLAY
 SET O TO O+2
END FOR
```

which we found from looking for a sequence relationship.

7.5. Filtering from generated sequences

Yet another approach is to start with all the integers:

1 2 3 4 5 6...

and choose those that satisfy some criterion, maybe having modified them first.

For example, for odd integers, we could choose every integer which doesn't divide exactly by 2; that is every integer whose remainder is 1 when divided by 2:

```
FOR I FROM 1 TO N DO
 IF N mod 2 = 1 THEN
  SEND N TO DISPLAY
 END IF
END FOR
```

This wouldn't work very well for finding squares, though. We'd have to find the square root of every integer and multiply it by itself to see if we got the original integer. If we didn't then we'd know that taking the square root hadn't given an exact integer and had been rounded down, so the integer was a proper square to begin with.

7.6. Reflection

We have seen a number of ways to find a pattern in a sequence and hence derive an algorithm.

First of all, we looked at the *relationship between successive elements*. We:

- worked out a general case in terms of element i+1 and element i;
- wrote down an initial case;
- used the initial case to initialise a variable;
- used a FOR loop and the general case to change the variable.

Second, we looked at the *relationship between each element and its index*. We:

- worked a general case for element i. This might be a simple function of i (*generation*) or involve applying some criterion (*filtering*).
- used a FOR loop and the general case to find the ith element.

The first approach really does involve finding a pattern between elements of a sequence. But it can lead to non-intuitive solutions.

The second approach can feel like a bit of a cheat. If we know in advance how to characterise the important properties of each element, we seem to wind up coding rather than problem solving.

7.7. Exercises

For each of the following sequences:

a) find a pattern by considering the relationship between successive elements;
b) write an algorithm from a) to generate successive elements;
c) write an iterative and a recursive function from b) to find the value of element i;
d) find a pattern by considering the relationship between each element and its index;
e) write an algorithm from d) to generate successive elements;
f) write a function from e) to find the value of element i.

The sequences are:

a) 2 4 6 8 10 12...
b) 1 3 7 13 21 31 43...
c) (1) 1 2 3 5 8 13 21 34...
d) for some X: X^1 X^2 X^3 X^4 X^5 ...
e) 1 2 3 5 7 11 13...

8. Imperative and object oriented programming

8.1. Introduction

I'm now going to look at how we might elaborate object oriented (OO) programming concepts by:

- starting with a procedural program to manipulate a bounded push down stack;
- successively applying the Computational Thinking (CT) stages of *pattern; identification* and *abstractio*n to both the data structure and the sub-programs.

8.2. The stack

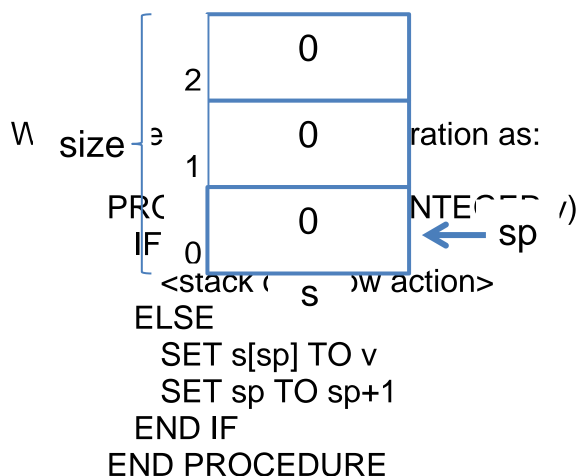
Suppose we want to make an integer stack from a:

- size;
- array of that size;
- stack pointer

For a new stack, we want every element set to 0, and the stack pointer to be set to 0 to indicate the bottom of the stack:

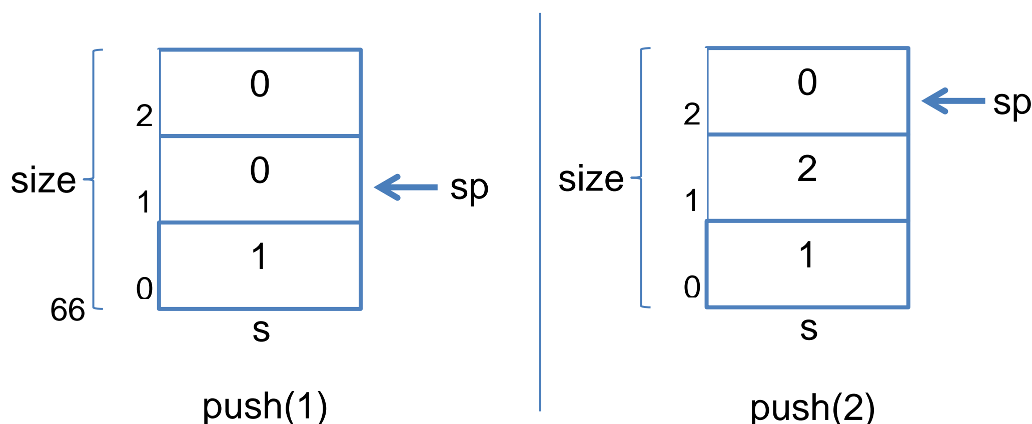
```
DECLARE size INITIALLY 3
DECLARE s IS ARRAY OF INTEGER INITIALLY [0]*size
DECLARE sp INITIALLY 0
```

We can visualise the initial stack as:



W ration as:

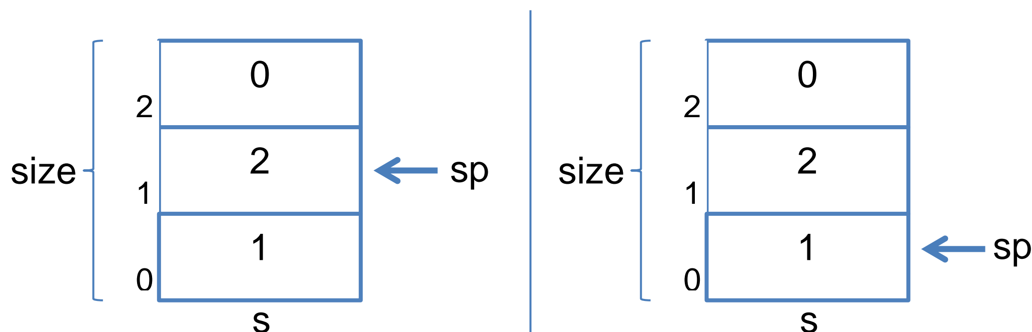
```
PR( NTE /)
  IF
    <stack ( )w action>
  ELSE
    SET s[sp] TO v
    SET sp TO sp+1
  END IF
END PROCEDURE
```



And we can define the pop operation as:

```
FUNCTION pop() RETURNS INTEGER
  IF sp=0 THEN
    <stack underflow action>
  ELSE
    SET sp TO sp-1
    RETURN s[sp]
  END IF
END FUNCTION
```

so:



This code has the apparent advantage that the variables are hard coded into the sub-programs that manipulate them, emphasising the strong conceptual connection between them.

However, the big disadvantages are that:

- the sub-programs can't be used with different variables representing other stacks;
- the variables are global to the whole program, so arbitrary code can change them with unpredictable effects.

8.3. Two stacks

Suppose we now want another stack but this time of size 4. We could just cut and paste the above code, rename all the variables and sub-programs, and change the initialisation:

```
DECLARE size1 INITIALLY 4
DECLARE s1 IS ARRAY OF INTEGER INITIALLY [0]*size1
```

```
DECLARE sp1 INITIALLY 0

PROCEDURE push1(INTEGER v)
  IF sp1=size1 THEN
    <stack overflow action>
  ELSE
    SET s1[sp1] TO v
    SET sp1 TO sp1+1
  END IF
END PROCEDURE

FUNCTION pop1() RETURNS INTEGER
  IF sp1=0 THEN
    <stack underflow action>
  ELSE
    SET sp1 TO sp1-1
    RETURN s1[sp1]
  END IF
END FUNCTION
```

Well, we certainly have two stacks. Alas:

- our code is every bit as insecure as before;
- and we now have twice as much of it;
- we have to explicitly call different sub-programs to manipulate different stacks.

8.4. Patterns and abstraction

Let's compare our two chunks of code and look for patterns by identifying differences. First of all, for the declarations, we have a common pattern:

```
DECLARE ? INITIALLY ?
DECLARE ? AS ARRAY OF INTEGER INITIALLY [0]*?
DECLARE ? INITIALLY 0
```

Conceptually, these three variables are strongly related in our stack model so we could abstract by:

- separating out the variables from their initialisations;
- defining a unitary record structure:

```
RECORD stack IS {INTEGER size,
                 ARRAY OF INTEGER s,
                 INTEGER sp}
```

- initialising record values when they're created:

```
DECLARE s1 IS stack(3,[0]*3,0)
DECLARE s2 IS stack(4,[0]*4,0)
```

Now we distinguish the variables:

s1.size, s1.s & s1.sp

from:

s2.size, s2.s & s2.sp

Similarly, for the sub-programs, we have common patterns:

```
PROCEDURE ?(INTEGER v)
  IF ?=? THEN
    <stack overflow action>
  ELSE
    SET ?[?] TO v
    SET ? TO ?+1
  END IF
END PROCEDURE

FUNCTION ?() RETURNS INTEGER
  IF ?=0 THEN
    <stack underflow action>
  ELSE
    SET ? TO ?-1
    RETURN ?[?]
  END IF
END FUNCTION
```

So, just as we abstracted our declarations with a record, we can do the same here by:

- introducing record formal parameters:
- abstracting inside the sub-programs with variable references relative to that record:

```
PROCEDURE push(stack st, INTEGER v)
  IF st.sp=st.size THEN
    <stack overflow action>
  ELSE
    SET st.s[st.sp] TO v
    SET st.sp TO st.sp+1
  END IF
END PROCEDURE

FUNCTION pop(stack st) RETURNS INTEGER
  IF st.sp=0 THEN
    <stack underflow action>
  ELSE
    SET st.sp TO st.sp-1
    RETURN st.s[st.sp]
```

```
    END IF
  END FUNCTION
```

Now, we can push to and pop from our two stacks with:

```
push(s1,value) ...pop(s1)...
```

and

```
push(s2,value) ...pop(s2)...
```

We seem to have solved our code bloat problem but we still have insecure code as the record variables are global. Furthermore, we've lost that strong connection between the stack information and the sub-programs that manipulate it.

8.5. Class = encapsulate(record + sub-programs)

It would be nice if we could somehow bundle together the record that holds the stack variables with the sub-programs that manipulate them, so that when we create a stack:

- the sub-programs know that they are only to work on the corresponding variables;
- it isn't possible to change those variables other than by using the sub-programs.

Before we see how to do this, let's think again about the record definition:

```
RECORD stack IS {INTEGER size,
                 ARRAY OF INTEGER s,
                 INTEGER sp}
```

Remember that this doesn't actually declare anything. Rather, it's a *specification* of what a record value contains. We can think of this as being like an architectural design for a house, which we certainly can't live in, unless it's 1:1 scale and made of tent cloth, but we can use to make actual houses.

Then, when we declare a RECORD, for example:

```
DECLARE s1 INITIALLY stack(3,[0]*3,0)
```

we're asking for a new individual stack record value to be created with the fields initialised to the given values. We call this individual value an *instance* of the record.

Note that we use the RECORD identifier as if it were the name of a function that when calls returns an appropriate value, so we term the identifier the *constructor*.

Just as a record is an abstraction for a related group of variables, a *clas*s is an abstraction for a related group of variables and sub-programs. Here the subprograms are called *methods*.

A class definition looks like a record definition with an additional section where the methods are defined, for example:

```
CLASS stack IS {INTEGER size,
                ARRAY OF INTEGER s,
                INTEGER sp}
METHODS

PROCEDURE push ... END PROCEDURE

FUNCTION pop ... END FUNCTION

END CLASS
```

Note that different people may refer to the variables of the class as *fields* or *attributes* or *class variables*.

Just as with records, a class definition doesn't actually create anything. Rather, it specifies how to make individual instances of the class, termed *objects*.

For example:

```
DECLARE s3 INITIALLY stack(10,[0]*10,0)
```

makes a new stack object associated with the variable s3, with size set to 10, s to [0...0] and sp to 0.

However, unlike a record value, the fields of an object can only be accessed by the methods of the object. For example, in our program we cannot refer to s3.size or s3.s or s3.sp.That is, the fields are *private*.

Nonetheless, we can access the methods of the object to manipulate the fields, by referring to them via the associated variable name, just as we refer to the fields of records. That is, the methods are *public*.

For example:

```
s3.push(...)
```

indicates that the push method for the object associated with s3 is to be called. This is also known as *message passing* as it's as if we're asking the object to perform the required method.

We say that an object *encapsulates* variables and sub-programs. We have achieved both code reuse through object creation, and code security through private class variables.

8.6. Madness in the methods

Let's now return to the methods. First of all, we no longer need to pass the class variables as parameters. Any variable mentions inside a method can only be to class variables, if not to formal parameters of local variables.

However, we don't just use the class variables themselves. Rather, we make use of the generic class variable THIS which always refers to the current object.

Thus, in our stack example, we might write

```
CLASS stack IS {INTEGER size,
                ARRAY OF INTEGER s,
                INTEGER sp)
METHODS

PROCEDURE push(INTEGER v)
  IF THIS.sp=THIS.size THEN
    <stack overflow action>
  ELSE
    SET THIS.s[THIS.sp] TO v
    SET THIS.sp TO THIS.sp+1
  END IF
END PROCEDURE

FUNCTION pop() RETURNS INTEGER
  IF THIS.sp=0 THEN
    <stack underflow action>
  ELSE
    SET THIS.sp TO THIS.sp-1
    RETURN THIS.s[THIS.sp]
  END IF
END FUNCTION

END CLASS
```

So when we call, say:

```
s3.push(42)
```

it's as if we've replaced every occurrence of THIS in push with s3.

Frankly, I was alarmed when I first saw this style of coding as I'd been long used to structured programming where it's deemed wrong for sub-programs to manipulate global variables. But, with encapsulation, this is wholly appropriate as only the methods of an object can change its class variables.

Anyway, let's create another stack object:

```
DECLARE s4 INITIALLY stack(5,[0,0,0,0,0],0)
```

Now let's push three values onto s3, and then pop them off and push them onto s4 in reverse order:

```
FOR i FROM 1 TO 3 DO
  s3.push(i)
END FOR
FOR i FROM 1 TO 3 DO
  s4.push(s3.pop())
END FOR
```

Notice that when we call s3.push(i) and s3.pop() we're asking s3 to change its own s and sp, with reference to its own size. That is, for these calls THIS means s3. And when we call s4.push(i), we're asking s4 to change its own s and sp, again with reference to its own size. That is, for this call THIS means s4.

8.7. Overloading constructors

Right now, when we make a stack, we have to explicitly nominate the size, the initial stack contents, and the initial stack pointer. We do this by calling the implicit constructor via the class identified. But we said we'd like every stack element to be initialised to 0 and to start with the stack pointer set to 0, so it would be nice if we could just supply the stack size and have standard code to set the stack contents and pointer.

We can define an explicit constructor:

```
CONSTRUCTOR (INTEGER sz)
  DECLARE THIS.size INITIALLY sz
  DECLARE THIS.s INITIALLY [0]*size
  DECLARE THIS.sp INITIALLY 0
END CONSTRUCTOR
```

In general, we can have multiple constructors without ambiguity provided they can be distinguished by the number and/or types of formal parameters. Here we are said to have *overloaded* the constructor. Note that the implicit constructor is still valid.

Note that an overloaded constructor:

- must declare and initialise all the class variables;
- doesn't have an identifier.

Now, we can create a stack with:

DECLARE s5 INITIALLY stack(30)

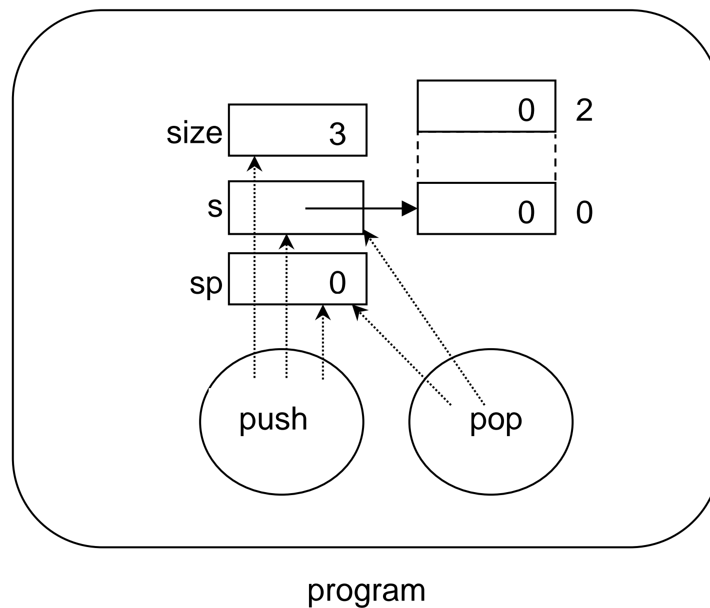
which will have the same effect as:

DECLARE s5 INITIALLY stack(30,[0]*30,0)

8.8. Summary

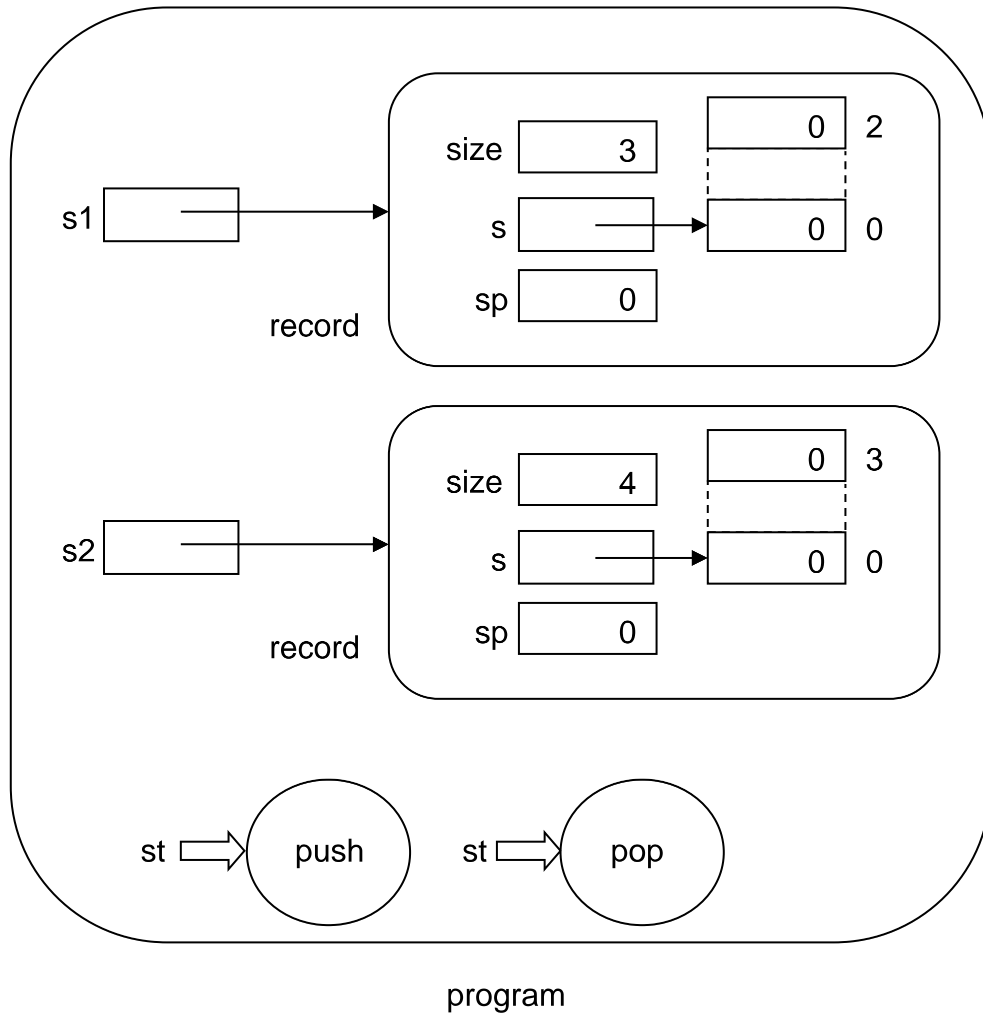
Let's draw things together by thinking about how we've represented stacks at each stage.

We started with global variables manipulated directly by global sub-programs:



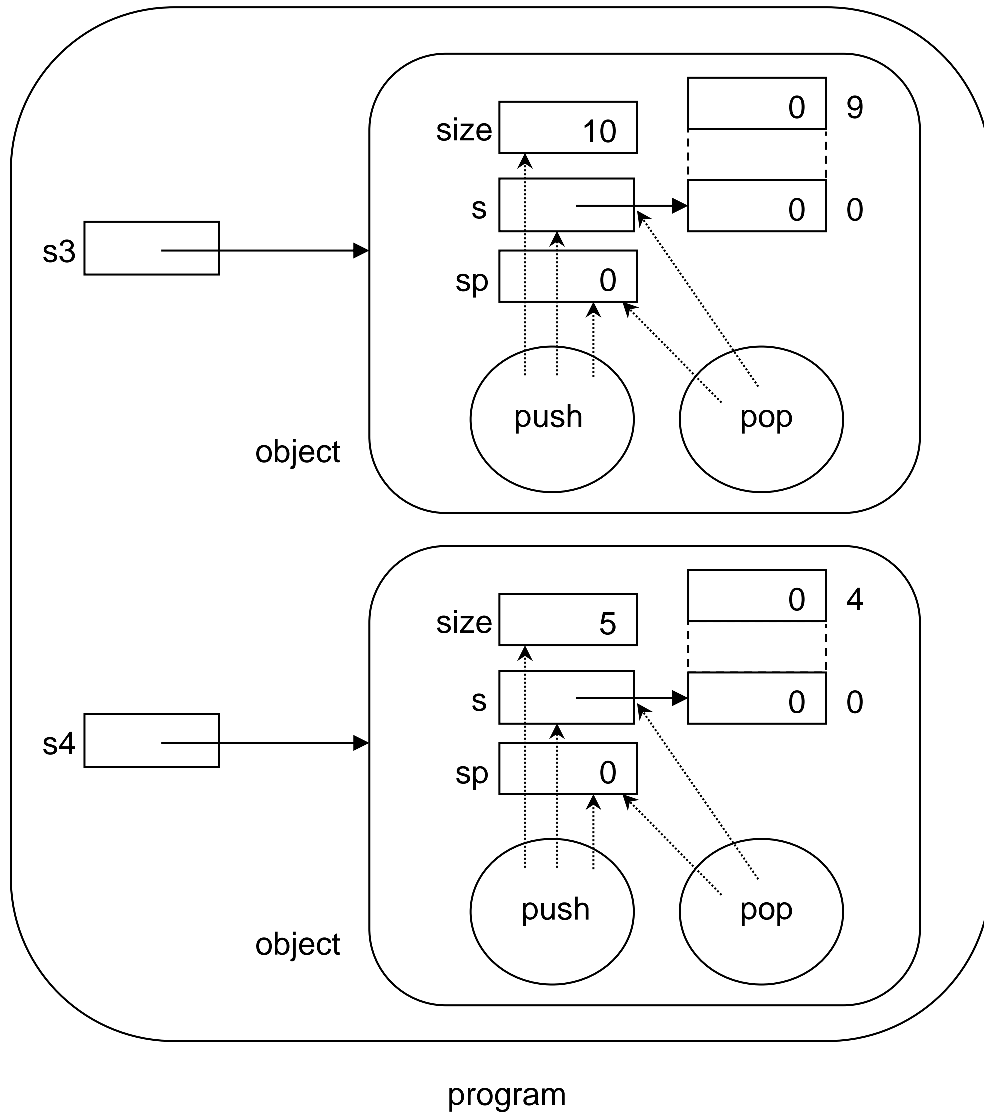
program

Next, we introduced records accessed by global variables, manipulated indirectly as parameters by global sub-programs:



program

Finally, we encapsulated the record and sub-programs to give objects accessed by global variables, with local variables manipulated directly by local methods.



program

Note how the objects in the final stage have the same structure as the whole program in the initial stage.

9. Basic algorithms and data structures

9.1. Linear array

9.1.1. Linear Search

```
FUNCTION linearSearch(ARRAY OF INTEGER a,
                      INTEGER length,
                      INTEGER v) RETURNS INTEGER
   DECLARE I INITIALLY 0
   DECLARE found INITIALLY FALSE
   WHILE NOT found AND i<length DO
      IF v=a[i] THEN
         SET found TO TRUE
      ELSE
         SET i TO i+1
      END IF
   END WHILE
   RETURN i
END FUNCTION
```

9.1.2. Binary search – ascending order - iterative

```
FUNCTION binarySearch(ARRAY OF INTEGER a,
                      INTEGER length,
                      INTEGER v) RETURNS INTEGER
   DECLARE left INITIALLY 0
   DECLARE right INITIALLY length-1
   DECLARE middle INITIALLY 0
   DECLARE result INITIALLY -1
   DECLARE found INITIALLY FALSE
   WHILE NOT found AND left <= right DO
      SET middle TO (left+right)/2
      IF a[middle]=v THEN
         SET result TO middle
         SET found TO TRUE
      ELSE
         IF a[middle]>v THEN
            SET right TO middle-1
         ELSE
            SET left TO middle+1
         END IF
      END IF
   END WHILE
   RETURN result
```

```
END FUNCTION
```

9.1.3. Binary search – ascending order – recursive

```
FUNCTION recBinarySearch(ARRAY OF INTEGER a,
                         INTEGER left,
                         INTEGER right,
                         INTEGER v) RETURNS INTEGER
   DECLARE middle INITIALLY (left+right)/2
   IF left>right THEN
      <not found action>
   ELSE
      IF a[middle]=v THEN
         RETURN middle
      ELSE
         IF a[middle]<v THEN
            RETURN recBinarySearch(a,left,middle,v)
         ELSE
            RETURN recBinarySearch(a,middle+1,right,v)
         END IF
      END IF
   END IF
END FUNCTION
```

9.1.4. Swap

```
PROCEDURE swap(ARRAY OF INTEGER a,
               INTEGER i,
               INTEGER j)
    DECLARE temp INITIALLY a[i]
    SET a[i] TO a[j]
    SET a[j] TO temp
 END PROCEDURE
```

9.1.5. Bubble sort – ascending order

```
PROCEDURE bubbleSort(ARRAY OF INTEGER a,
                     INTEGER length)
   FOR i FROM length-2 TO 0 STEP -1 DO
      FOR j FROM 0 TO i DO
         IF a[j]>a[j+1] THEN
            swap(a,j,j+1)
         END IF
      END FOR
   END FOR
END PROCEDURE
```

9.1.6. Bubble sort – ascending order - with success check

```
PROCEDURE fastBubbleSort(ARRAY OF INTEGER a,
                         INTEGER length)
   DECLARE swaps INITIALLY true
   DECLARE i INITIALLY length-2
   WHILE swaps AND i>=0 DO
      SET swaps TO false
      FOR j FROM 0 TO i DO
         IF a[j]> a[j+1] THEN
            swap(a,j,j+1)
            SET swaps TO true
         END IF
      END FOR
      SET i TO i-1
   END WHILE
END PROCEDURE
```

9.1.7. Quicksort – ascending order

```
PROCEDURE quickSort(ARRAY OF INTEGER a,
                    INTEGER left,
                    INTEGER right)
   IF left<right THEN
      DECLARE middle INITIALLY partition(a,left,right)
      quickSort(a,left,middle)
      quickSort(a,middle+1,right)
   END IF
END PROCEDURE

FUNCTION partition(ARRAY OF INTEGER a,
                   INTEGER left,
                   INTEGER right) RETURNS INTEGER
   DECLARE l INITIALLY left
   DECLARE r INITIALLY right
   DECLARE pivot INITIALLY a[l]
   WHILE l<r DO
      WHILE a[l]<pivot DO
         SET l TO l+1
      END WHILE
      WHILE a[r]>pivot DO
         SET r TO r-1
      END WHILE
      IF l<r THEN
         swap(a,l,r)
         SET l TO l+1
         SET r TO r-1
      END IF
   END WHILE
```

```
   RETURN l
END FUNCTION
```

9.1.8. Insert – ascending order

```
PROCEDURE insert(ARRAY OF INTEGER a,
                 INTEGER next,
                 INTEGER length,
                 INTEGER v)
   DECLARE i INITIALLY 0
   IF next=length THEN
      <array full action>
   ELSE
      WHILE i<next AND v>a[i] DO
         SET i TO i+1
      END WHILE
      FOR j FROM next TO i+1 STEP -1 DO
         SET a[j] TO a[j-1]
      END FOR
      SET a[i] TO v
      SET next TO next+1
   END IF
END PROCEDURE
```

9.1.9. Delete – ascending order

```
PROCEDURE delete(ARRAY OF INTEGER a,
                 INTEGER next,
                 INTEGER v)
   DECLARE i INITIALLY 0
   DECLARE found INITIALLY false
   WHILE NOT found AND i<next DO
      IF v=a[i] THEN
         SET found TO true
      ELSE
         SET i TO i+1
      END IF
   END WHILE
   IF found THEN
      FOR j FROM i TO next-2 DO
         SET a[j] TO a[j+1]
      END FOR
      SET next TO next-1
   ELSE
      <not found action>
   END IF
END PROCEDURE
```

9.2. Stack

```
CLASS stack IS {ARRAY OF INTEGER s,
               INTEGER sp,
               INTEGER size}
METHODS

CONSTRUCTOR (INTEGER sz)
   DECLARE THIS.size INITIALLY size
   DECLARE THIS.s INITIALLY [0]*size
   DECLARE THIS.sp INITIALLY 0
END CONSTRUCTOR
```

9.2.1. Push

```
PROCEDURE push(INTEGER V)
   IF THIS.sp=THIS.size THEN
      <stack overflow action>
   ELSE
      SET THIS.s[THIS.sp] TO v
      SET THIS.sp TO THIS.sp+1
   END IF
END PROCEDURE
```

9.2.2. Pop

```
FUNCTION pop() RETURNS INTEGER
   IF THIS.sp=0 THEN
      <stack underflow action>
   ELSE
      SET THIS.sp TO THIS.sp-1
      RETURN THIS.s[THIS.sp]
   END IF
END FUNCTION

END CLASS
```

9.3. Queue

```
CLASS queue IS {ARRAY OF INTEGER q,
                INTEGER qp,
                INTEGER size}
METHODS

CONSTRUCTOR (INTEGER sz)
   DECLARE size INITIALLY sz
   DECLARE q INITIALLY []*size
   DECLARE qp INITIALLY 0
```

```
END CONSTRUCTOR
```

9.3.1. Join

```
PROCEDURE join(INTEGER v)
   IF THIS.qp=THIS.size THEN
      <queue overflow action>
   ELSE
      SET THIS.q[THIS.qp] TO v
      SET THIS.qp TO THIS.qp+1
   END IF
END PROCEDURE
```

9.3.2. Leave

```
FUNCTION leave() RETURNS INTEGER
   DECLARE result INITIALLY <whatever>
   IF THIS.qp=0 THEN
      <queue underflow action>
   ELSE
      SET result TO THIS.q[0]
      FOR i FROM 0 TO THIS.qp-2 DO
         SET THIS.q[i] TO THIS.q[i+1]
      END FOR
      SET THIS.qp TO THIS.qp-1
      RETURN result
   END IF
END FUNCTION

END CLASS
```

9.4. Linked list – iterative/update

```
RECORD cell IS {INTEGER value,
                cell next}

CLASS list IS {cell first}
METHODS
```

9.4.1. Show

```
PROCEDURE show()
   DECLARE f INITIALLY THIS.first
   WHILE f!=[] DO
      SEND f.value TO DISPLAY
      SET f TO f.next
```

```
   END WHILE
END PROCEDURE
```

9.4.2. Insert – ascending order

```
PROCEDURE insert(INTEGER v)
   IF THIS.first=[] THEN
      SET THIS.first TO cell(v,[])
   ELSE
      IF v<THIS.first.value THEN
         SET THIS.first TO cell(v,THIS.first)
      ELSE
         DECLARE f INITIALLY THIS.first
         DECLARE done INITIALLY false
         WHILE NOT done DO
            IF f.next=[] THEN
               SET f.next TO cell(v,[])
               SET done TO true
            ELSE
               IF v<f.next.value THEN
                  SET f.next TO cell(v,f.next)
                  SET done TO true
               ELSE
                  SET f TO f.next
               END IF
            END IF
         END WHILE
      END IF
   END IF
END PROCEDURE
```

9.4.3. Delete- ascending order

```
PROCEDURE delete(INTEGER v)
   IF first=[] THEN
      <not found action>
   ELSE
      IF THIS.first.value=v THEN
         SET THIS.first TO THIS.first.next
      ELSE
         DECLARE f INITIALLY THIS.first
         DECLARE done INITIALLY false
         WHILE NOT done DO
            IF f.next=[] THEN
               <not found action>
            ELSE
               IF f.next.value=v THEN
                  SET f.next TO f.next.next
                  SET done TO true
```

```
               ELSE
                  SET f TO f.next
               END IF
            END IF
         END WHILE
      END IF
   END IF
END PROCEDURE

END CLASS
```

9.5. Linked list – recursive/copy

```
CLASS recList WITH {INTEGER value, recList next}
METHODS
```

9.5.1. Show

```
PROCEDURE show()
   IF THIS!=[] THEN
      SEND THIS.value TO DISPLAY
      THIS.next.show()
   END IF
END PROCEDURE
```

9.5.2. Insert – ascending order

```
FUNCTION insert(INTEGER v) RETURNS recList
   IF THIS=[] THEN
      RETURN recList(v,[])
   ELSE
      IF v<THIS.value THEN
         RETURN recList(v,THIS)
      ELSE
         RETURN recList(THIS.value,THIS.next.insert(v))
      END IF
   END IF
END FUNCTION
```

9.5.3. Delete – ascending order

```
FUNCTION delete(INTEGER v) RETURNS recList
   IF THIS=[] THEN
      <not found action>
   ELSE
      IF THIS.value=v THEN
         RETURN THIS.next
```

```
      ELSE
         RETURN recList(THIS.value,THIS.next.delete(v))
      END IF
   END IF
END FUNCTION
```

9.5.4. Sort – ascending order

```
FUNCTION sort() RETURNS recList
   IF THIS=[] THEN
      RETURN []
   ELSE
      RETURN (THIS.next.sort()).insert(THIS.value)
   END IF
END FUNCTION

END CLASS
```

9.6. Exercises

For each of the following, trace the changes to the variables and associated data structures:

9.6.1. Linear array

```
1. DECLARE b INITIALLY [1,2,3,4,5,6,7,8,9,10]
2. DECLARE r INITIALLY 0
3. SET r TO linearSearch(b,10,5)
4. SET r TO linearSearch(b,10,11)
5. SET r TO binarySearch(b,10,9)
6. SET r TO binarySearch(b,10,11)
7. SET r TO recBinarySearch(b,0,9,9)
8. SET r TO recBinarysearch(b,0,9,11)

9. DECLARE c INITIALLY [4,1,3,2,5]
10.  bubbleSort(c,5)
11.  fastBubbleSort(c,5)  # with original c
12.  quickSort(c,0,4)        # with original c

13.  DECLARE d INITIALLY []*4
14.  DECLARE n INITIALLY 0
15.  insert(d,n,4,4)
16.  insert(d,n,4,1)
17.  insert(d,n,4,3)
18.  insert(d,n,4,4)
19.  insert(d,n,5,5)
20.  delete(d,n,2)
21.  delete(d,n,5)
```

9.6.2. Stack

```
1. DECLARE ss INITIALLY stack(3)
2. ss.push(2)
3. ss.push(1)
4. ss.push(3)
5. ss.push(4)
6. SET r TO ss.pop()
7. SET r TO ss.pop()
8. SET r TO ss.pop()
9. SET r TO ss.pop()
```

9.6.3. Queue

```
1. DECLARE qq INITIALLY queue(3)
2. qq.join(2)
3. qq.join(1)
4. qq.join(3)
5. qq.join(4)
6. SET r TO qq.leave()
7. SET r TO qq.leave()
8. SET r TO qq.leave()
9. SET r TO qq.leave()
```

9.6.4. Linked list – iterative/update

```
1. DECLARE l1 INITIALLY list([])
2. l1.insert(1)
3. l1.insert(4)
4. l1.insert(2)
5. l1.insert(3)
6. l1.show()
7. l1.delete(4)
8. l1.delete(1)
9. l1.delete(5)
```

9.6.5. Linked list – recursive/copy

```
1. DECLARE l2 initially recList([])
2. SET l2 TO l2.insert(1)
3. SET l2 TO l2.insert(4)
4. SET l2 TO l2.insert(2)
5. SET l2 TO l2.insert(3)
6. l2.show()
7. SET l2 TO l2.delete(4)
8. SET l2 TO l2.delete(1)
9. SET l2 TO l2.delete(5)
```

Appendix A. Haggis Pseudocode