Software Language Engineering

Adrian Johnstone

Edition:

Joseph Wright of Derby; An Iron Forge
©Tate Gallery 1992

See: `https://en.wikipedia.org/wiki/The_Blacksmith's_Shop`

Contents

1 Formalisation

> *What one programmer can do in one month, two programmers can do in two months* — Fred Brooks

At their simplest, computers come pre-equipped with a repertoire of machine level operations such as `add` and `subtract`, a memory space and a set of addressing modes that allow data to be read from memory, combined using one of the operations and then written back to memory. It is remarkable that the subtle and immersive effects of, say, a realtime virtual reality recreation of a forest can be generated merely by performing large numbers of simple arithmetic and logic operations at high speed.

Constructing programs using only basic machine operations is arduous. For instance, until the mid-1980's it was common for machines to not even have a full set of arithmetic operations since hardware implementations of multiplication and division were expensive. Implementing multiplication and division through sequences of shift and add operations was well understood, and so a standard set of instructions to perform multiplication would normally be provided by the manufacturer. These instruction sequences could be bodily inserted into a program that required a multiplication, or to save space, a single copy of the instruction sequence could be held in memory and a call-and-return mechanism used to execute the multiplication as a sub-program of the main application.

**** Todo: Autocodes for arithmetic**

**** Todo: Programmers and lines of code: higher level = higher productivity?**

**** Todo: What counts as a software language**

1.1 It's languages all the way down...

Modern computer systems are composed of many layers of languages which may be translated into other languages or interpreted directly. At the lowest level, we have a particular processor's machine code which is interpreted by the hardware.

For instance, a computer game may be implemented as a script running on a *game engine*. The engine's script language is not directly executable by any real computer, so instead a program called a *game script interpreter* reads the script and performs appropriate actions. The game script interpreter could be written directly in the machine language of some real computer, but that would

then mean the the interpreter could only run on computers with a particular central processor, and in any case machine languages are very fine grained, and require a programmer to keep track of a multitude of clerical details. Instead, the programmer could elect to write the interpreter in a high level language such as Java, and another program called the *Java compiler* could then translate that into the compact machine-like language of the *Java Virtual Machine* (JVM). Yet another program (the JVM interpreter) reads that code, and performs actions by directly accessing the underlying machine instructions.

At first glance much of this sounds circular: we put in extra layers between the script interpreter and the JVM, but the JVM is itself interpreting, so what is the advantage? In terms of absolute performance and functionality, none. We could have written a game script interpreter in the machine language of some specific machine, and it might run faster than the layered implementation described above. However, it would be non-portable to other machines, and it would almost certainly take very much longer to write. By using mature established sub-systems such as the Java ecosystem we can reuse the work of others, allowing us to construct complex artifacts such as computer games by *implementing* our ideas using the languages of existing tools. In a sense, the first step of implementation is itself a translation: from concepts in our minds into specifications written in some software language, and this forms the topmost layer of the translation stack.

In detail, systems involve other languages too. Many applications make use of XML files to store data, and XML itself is a language which must be interpreted when constructing internal representations of data. Online systems often use relational databases, with access mediated *via* SQL statements embedded in other languages. When developing software, the build systems (such as `make`) that we use when developing large software systems also have specification languages associated with them. We can also view file formats such as JPEG as language specifications, as well as the many protocols that we use to communicate via networks or with specialised hardware such as 3D printers and music synthesizers. The testing that we do may generate results stored in spreadsheets, whose formulae and scripting systems are software languages. Not least, documentation such as this book may be prepared in LaTeX, which is a set of routines written as macros in the underlying TeX language, the interpreter for which is written in a 'literate' language called WEB, which can be translated into both Pascal and C and then compiled into the machine language of a variety of processors. Software languages are all-persuasive.

**** Todo: 3D printing as a web of languages**

**** Todo: Turtles all the way down anecdote, finishing off with languages all the way down to the hardware**

1.2 Lessons from natural language

When learning a new human language, we might first concentrate on *vocabulary* (the set of generally understood words in that language) and perhaps try to match up words for objects in the new language with the equivalent words in

our native language.

Taking French as an example, an English speaker might note that maison in French corresponds to house in English and that voiture corresponds to car. We are not limited to names for objects: attributes such as colour also have closely corresponding words such as rouge for red and noir for black.

In an emergency, and with some good will on both sides, it is possible to communicate simple ideas using just vocabulary, but most communication requires the construction of complete sentences by sequencing together words from the vocabulary. In typical human languages, the order of words within these sequences is significant.

Some sequences are 'wrong' in that a native speaker would not utter them. For instance, in English 'The car black.' sounds very odd: we expect 'The black car.' It seems as though we expect the attributes of an object (colour in this case) to be listed before uttering the word for the object. In French, however, the attributes typically appear after the name of the object: 'la maison rouge' corresponds to the 'the red house' in English. Interestingly, although 'The car black.' is not a valid English sentence, if we were speaking to somebody who was learning English we could probably guess what they meant.

The rules governing valid word orderings are called the *syntax* of the language, and ignoring the rules can cause deep confusion. For instance a word-by-word translation of 'La maison rouge.' yields 'The house red.', which is a valid colloquial phrase since *The house red* in a restaurant means their non-label (and usually cheaper) wine. Simply by putting the colour last we have completely changed the meaning of the sentence.

Our programming languages are also like this. The symbol −, for instance means 'negate' if it appears at the beginning of an expression, but means 'subtract 3 from 2' if it appears between the symbols 3 and 2. In the Java language, the symbol + can stand for the addition of integers, the addition of reals or the concatenation of strings. Not only the order of symbols, but the kinds of things they stand for controls the meaning of phrases in programs.

1.2.1 Semantics, syntax, ambiguity and 'sayability'

The fact that we can translate between languages shows that in some deep sense, meaning is independent of vocabulary and syntax. The *semantics* of a sentence is the collection of meanings it may have, and the semantics of a language is thus the collection of meanings that can be expressed in that language. The language itself is just a set of word sequences. The syntax rules and the vocabulary together form the *syntax* which is a specification of the language itself. The *translation* of a phrase from language E to Language F involves changing the syntax from that of E to that of F in such a way as to preserve the semantics.

Unfortunately, syntax rules occasionally allow multiple interpretations of a sentence. My favourite example is an advertising slogan for a headache remedy that we shall call X. The slogan is *Nothing works faster than* X. This might mean that X is a faster-acting remedy than anything else (because No Thing works faster than X) or it might mean that simply doing nothing will clear a

headache more quickly than X because doing nothing is faster than taking X. A closely related sentence is *Take X, you won't get better.* When a sentence has multiple meanings, we say that it is *ambiguous.*

Although semantics and syntax are often presented as two independent, orthogonal aspects of language, each may constrain the other. The idea that human syntax and vocabulary in some sense limit what can be thought is intriguing. George Orwell in his satirical novel 1984 posited a world in which the English language had been deliberately pruned into a language called Newspeak so as to make it impossible for the populace to construct anti-establishment narratives. Given the speed at which humans develop new linguistic expressions, it is highly debatable as to whether such a strategy could ever really succeed. Nevertheless, if we want to communicate a notion (and possibly if we even want to be able to internally think about a notion) then surely we need some language to describe it.

1.2.2 Formal languages: the need for precision

Human languages are rather slippery things. New words are coined all the time, so the vocabulary certainly is not fixed. Syntactic forms also come in and out of vogue, so even the syntax is gradually evolving. In addition, sentences may be ambiguous, that is have multiple interpretations, and some syntactically well-formed sentences are nonsensical and thus have no interpretations at all. A famous example coined by Noam Chomsky in 1957 is *Colourless green ideas sleep furiously* which contains multiple contradictions and conceptual confusions: colourless and green cannot both be attributes of a thing at the same time; ideas are not entities that can sleep, and so on.

Apart from these difficulties, the process by which a sentence that I write can summon up meanings in the reader's mind is deeply mysterious. Generations of philosophers and linguists have debated the rules by which we communicate and what it means to understand and convey meaning. That debate continues.

When we turn to computer languages, we seek precision of expression above all else; we certainly do not want our computer programs to have unpredictable results because of ambiguity. The main way of achieving this is to have very straightforward syntax (compared to natural languages), and to give explicit *disambiguation rules* that ensure that there is at most one meaning for each syntactically correct phrase. Our software languages must seem like very thin, limited things to a linguist; but that is how we want them. In particular we do not want a multitude of ways to express the same idea in our formal languages, whereas in journalism and literature the ability to find a novel way to say something is valued.

We might also hope that we could eradicate meaningless sentences in our software languages, but that is not usually possible: for instance division by zero is meaningless but cannot in general be detected simply by inspecting the program. Nevertheless, we try to organise things so that common errors may be reported before we actually start running the program. Program properties that can be deduced by looking at the code and which are independent of any particular data input are called *static* properties, and we shall sometimes talk

about the *static semantics* of a programming language.

Very roughly speaking, the static semantics may be evaluated independently of any program input which means that static properties such as type-correctness may be checked by a compiler before a program is run. The dynamic semantics of a language involves behaviour and properties which *are* input data dependent and thus reveal themselves only when the program is executed. One strand of programming language development (exemplified by Pascal) emphasises the use of compile-time rules to catch programmer errors; an opposing approach (exemplified by Smalltalk and Scheme) is to do most checking at run-time. Real programming languages are always a mix of these two but a language may be said to be more or less static in its approach. Both styles of language can deliver precision: the difference is the point within the development process at which errors may be detected. From a programmer's perspective, highly static languages offer early error reporting and are easier to generate efficient code for, but dynamic languages allow greater freedom of expression, and in that sense are sometimes thought of as more 'powerful'.

**** Todo: Adje van Wijngaarden's comments on semantics**

1.3 Utility and power in programming languages

What does it mean for one programming language to be 'more powerful' than another? Well, for all conventional general purpose languages 'more powerful' does not mean that there are that one language can ultimately do more than another.

**** Todo: Turing completeness**

If a general purpose language A can be used to implement an interpreter for general purpose language B (and indeed vice versa) then clearly they are formally of equivalent capability. However, when a programmer talks about the power of a particular language they are usually referring to the 'naturalness' and conciseness with which concepts may be expressed.

**** Todo: Fibonacci example with and without recursive routines**

For instance, early versions of FORTRAN required the arguments to a subroutine and its return point to be placed in machine locations which were fixed for the duration of a particular program execution. This simplified the design of the compiler, and was also efficient on some 1950's and 1950's architectures compared to the modern solution which is to place arguments on a stack. However, this used of fixed argument locations means that many recursive algorithms could not be expressed using recursive functions: since each recursive instance would use the same memory locations to hold arguments every time a function was called it would overwrite the previous call's arguments and also its return point. It is still possible to implement recursive algorithms in early FORTRAN using an explicitly managed stack data structure, but the resulting code is ugly, verbose, and more prone to errors.

1.4 The software engineering challenges in language design

Here is a common scenario. A company develops and sells set of applications which are specific to a particular domain. For example, we might

**** Todo: Lead programmer wants to write a language. Tapestry of tools and ad hoc solutions. Enormous difficulty of testing a language. Maintainability. Cost.**

1.5 An idealised methodology

**** Todo: See practice and theory section from next chapter - move back to here and then summarise our approach**

> *Mathematics is a game played according to certain simple rules with meaningless marks on paper.*

1.6 Formal systems

A mathematical proof of a theorem is a very careful explanation of why something must be true. Finding a proof can be very hard and require deep insight, but reading a proof should be more like reading a program, and once we understand the constructs of a programming language like Java, we should be able to read a Java program and manully 'execute' each individual action in isolation. To make full sense of a program you need a good grasp of the domain of application of the program or you will not understand the intent of the author or the utility of program, but the actual program text itself should not present difficulties of interpretation. Both programs and proofs are examples of *formal* reasoning, and in fact programs and proofs are closely related to one another.

We shall see several examples of formal systems, each of which comes with some objects that may be manipulated and some rules about how they may be changed. In this sense, a formal system is a game in which we can set up some initial configuration and then let the moves of the game evolve that configuration. One well known example is Conway's Game of Life in which patterns evolve on a rectangular grid. Once one has set up the rules of game, then we can ask questions about it and probe its properties. For instance, an early research question posed by Conway was whether any initial configuration would generate sequences of configurations whose area grows without limit. (The answer is yes: we shall examine Conway's game in more detail in Section 1.7.)

Of course, we need to be able to specify the objects and rules within our formal system unambiguously. English is a poor vehicle for precise communication, and so formal systems almost invariably come with some formal language (in which *well formed formulae* (wff) may be expressed) and a way of describing how, given a set of wff we can generate a new wff by applying the rules of the system to infer a new formula. One view of mathematics (which we shall call the *symbol-pushing* view) is that proofs are no more than sequences of well formed formulae in some formal system, and a theorem is the last line of such a sequence.

1.6.1 Practice and theory

In computing, *engineering* is the use of mathematics and scientific knowledge to invent new, useful systems and to improve the utility, performance, economics and reliability of existing systems. Implicit in this process is some notion that engineered systems have attributes are too complex to model mathematically, but which nevertheless contain subsystems that can be so modeled.

Historically, there has been a tension in the computing world between the practitioner community, who are focused on getting new systems working, and the theory community who seek insights via formal systems. For many problems, it has turned out to be hard to find formal descriptions which scale to real world problems and which are comfortable for practitioners to use, especially if they are unused to the culture of research level mathematics. As a result, some practitioners are dismissive of the utility of theoretical studies. This is a great shame, because the reality is that many, perhaps the majority, of our software systems are unreliable due to the high incidence of construction errors. In an ideal world, practitioners and theoreticians would work symbiotically to convert practitioner observations into mathematical models, which then allow the construction of new tools that embody principled design based on theory, with automatic generation of parts of those systems. This sort of virtuous circle is by and large absent in our field, although in more mature subjects (such as physics research and electronic engineering) it is standard practice.

Our goal in formalising system should be allow us to reason about useful properties of programs, such as whether they are complete (and might thus execute without ever raising exceptions); whether they terminate in all cases; whether the results meet their specification; and the way in which the time and space demands of a program vary with size of input. Perhaps surprisingly, it can be the case that an implementation based on a formalisation is not only more trustworthy, but also more efficient. We shall illustrate this in Section 1.7 when we shall look at a typical elementary programming challenge which has an inefficient 'natural' implementation, but for which a simple formalisation has a corresponding implementation that makes much better use of memory and requires far less computation for a given input.

Encouragingly, our field — software language engineering — is one of the areas of computing that has most benefited from cooperation between theoreticians and practitioners, and there now exists a set of mathematical techniques that allow us to specify and analyse software languages, and which (most importantly) also admit practical implementation. As a result, many aspects of language engineering can be managed using compact specifications which are then used to automatically generate implementations. However, there remain important aspects of language engineering that still rely on the insight and creativity of the engineer, either because there is no consensus on best practice (such as the syntax details of the 'best' general purpose programming languages), or because the mathematical techniques do not scale well to large language projects (such as non-modular semantic specification styles), or because automatic generation of efficient implementations for different machine architectures remains elusive (as is the case for many aspects of code generation

and interpretation). The research literature on software language topics is vast and will continue to grow.

Our approach in this book, then, is to emphasise the use of principled techniques and their tools where ever possible. In this we are aided by advances in the capacity and performance of our computing systems. Many standard practitioner techniques in software language engineering are rooted in the engineering constraints of 1970's era computer hardware, and in fact even the design of some programming languages reflects the inability of computers from that time to hold a complete representation of a program's source code in memory. For instance, in languages such as C and Pascal, it is necessary for function signatures to be declared before they are used so that the heavily constrained compilation system on those machines could generate code in a single pass over the source. In later languages such as Java this constraint is relaxed. More significantly, the currently conventional techniques used to analyse syntax and semantics are also limited in ways that is are longer necessary on modern hardware. An important goal of this book is to encourage the use of more general (though also much more resource hungry) techniques, freeing the designer to write specifications which are natural to them rather than having to rework their specifications into an artificial style so as to be admissable by a particular toolset.

1.6.2 Modularity and scalability

The key to efficiency (both in terms of the time take to develop a system and of the computer time it takes to execute) is decomposition into tractable subtasks.

A program is a specification for some behaviour that we require a computer to exhibit, and the actual behaviour is the semantics of the program. For instance, consider a program that reads four integers a, b, c and d from the keyboard and prints the result of evaluating $(a * b) + (c/d)$. Computers are finite, and the range of allowable integers is also finite, so in principle we could specify the semantics of the program as a finite set of tuples $\langle a, b, c, d, y \rangle$ where y is the output for inputs a, b, c and d.

Apart from the difficulty of expressing the value of y when d is zero (which we could overcome by defining some special value such as $\perp$ which indicates an illegal result) this is not really a useful way to express the semantics because, although finite, the list is absurdly long. If our computer is a modern 64-bit machine then there are $2^{(4*64)}$ different inputs, which is about 10^{77}. Estimates for the number of atoms in the universe are in the 10^{70}–10^{80} range, so clearly, exhaustively enumerating the entire semantics of this trivial program is impossible.

To reinforce our understanding of this challenge it is worth reading what Dijkstra had to say in 1972:

> Let us restrict, for a moment, our attention to the hardware and let us wonder to what extent one can convince oneself of its being properly constructed. Some years ago a machine was installed on the premises of my University; in its documentation it was stated

> that it contained, among many other things, circuitry for the fixed-point multiplication of two 27-bit integers. A legitimate question seems to be: "Is this multiplier correct, is it performing according to the specifications?".
>
> The naive answer to this is: "Well, the number of different multiplications this multiplier is claimed to perform correctly is finite, viz. 2^{54}, so let us try them all." But, reasonable as this answer may seem, it is not, for although a single multiplication took only some tens of microseconds, the total time needed for this finite set of multiplications would add up to more than 10,000 years! We must conclude that exhaustive testing, even of a single component such as a multiplier, is entirely out of the question. (Testing a complete computer on the same basis would imply the established correct processing of all possible programs!)

Technology has moved on, and my current machine can perform integer multiplications at rates of around 10^9 per second rather than the 10^5 per second mentioned by Djistra, so now the exhaustive test of 27-bit multiplication would only take more than one year, rather than more than 10,000 years. Set against that, my machine is doing 64-bit multiplication, so we now have 2^{128} unique inputs to test, which is about 10^{38}, requiring some 10^{29} seconds to compute which is more than 10^{21} years. Now, all is not lost because enumeration is not the only tool available to us. We shall use a divide-and-conquer strategy to avoid having to write out every output for every input.

When we learn arithmetic, we learn rules by which individual operations may be concisely specified: the algorithms for adding multi-digit numbers and performing long multiplication, for instance. We also learn that expressions may be *composed* from a sequence of individual operations in the order specified by the bracketing of an expression. (We also have rules of priority and associativity that allow us to omit parentheses in some circumstances, but really they are just a form of shorthand for the fully parenthesized expression.) In principle, then, we could reduce long multiplication to an algorithm which aggregates the results of multiplying two single digit numbers which is a small enough set of inputs that we can indeed exhaustively enumerate the outcomes, and check our design. That leaves the problem of deciding whether the algorithm by which we multiply multi-digit numbers together is correct.

1.7 Formalisation as an aid to engineering: Conway's Game of Life

In this section we explore the benefits to the working programer of concise formal descriptions using the example of Conway's Game of Life (CGL), which was described by Martin Gardner in the October 1970 Scientific American. Along the way, we shall encounter the notion of a system whose state is transformed using a relation over those states. We shall meet this idea again in Chapter **??** where we shall use it to produce very precise descriptions of the semantics of

programming languages.

CGL is an example of a cellular automaton. Here is a (not very good) formal description of a cellular automaton.

> A cellular automaton is a formal system comprising a set Γ of cells, each of which must be labeled with an element from a finite set of states Σ, and a transformation rule which specifies the evolution of the cell labels in discrete time steps $\Gamma_0, \Gamma_1, \ldots$ called generations.
>
> It is usual to think of Γ as having structure, in the sense that some cells are 'adjacent', and for the transformation rule to be a function over a 'neighbourhood' $f(p_1, p_2, \ldots, p_k)$ with signature $f : \Sigma \times \Sigma \times \ldots \times \Sigma \to \Sigma$ where the p_k are the elements of the neighbourhood.

Now, this second paragraph does not constitute a particularly useful formalisation in that we have not explained how there neighbourhoods are defined or used, and as a result we could not automatically generate a symbol-pushing implementation. We shall improve the formalisation in Section 1.7.4.

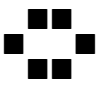
In CGL, the cells in Γ are arranged as an unbounded two-dimensional rectangular array; Σ is the set 0, 1 and the transformation rule defines the new state of a cell σ' in terms of the its existing state σ and the population P as the sum of the states of its eight-connected neighbours according to this function:

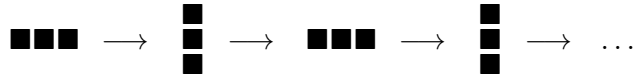
$$\sigma' = \begin{cases} 1, & \sigma = 0, P = 3 \\ 0, & \sigma = 1, P < 2 \vee P > 3 \\ \sigma, & \text{otherwise} \end{cases}$$

1.7.1 CGL examples

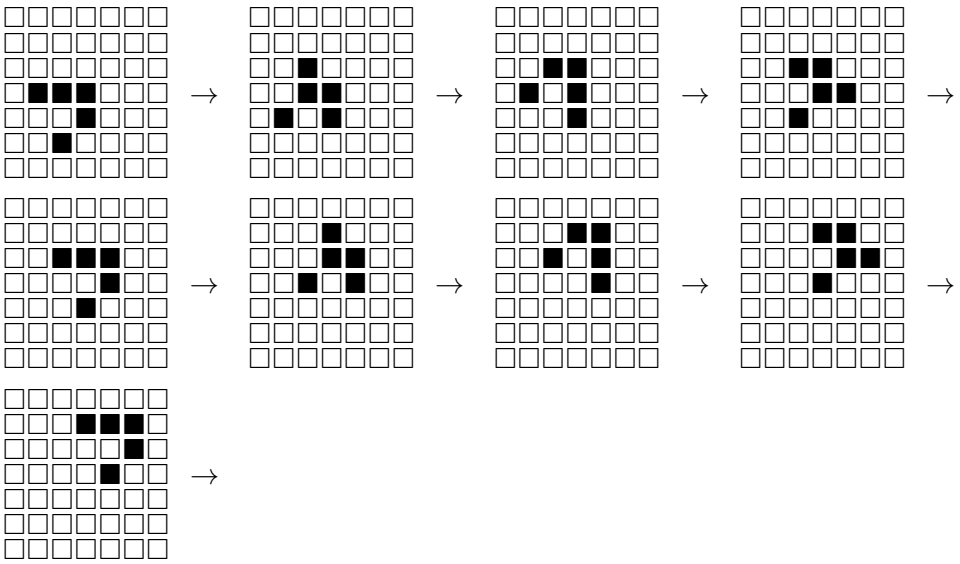
It is interesting to randomly fill a large part of a CGL array and allow the system to evolve. Surprisingly, regularities soon emerge. By looking closely at small parts of the evolving grid, we can identify small patterns that have interesting properties. Over the years, researchers have developed a common nomenclature for various kinds of CGL patterns: you can read about these at `http://conwaylife.com`

For instance, a *still life* is a pattern that does not change from generation to generation.

Well known still lives include the block  and the beehive 

An *oscillator* is a pattern that is a predecessor of itself, the simplest of which is the blinker 

A *spaceship* is a pattern that is a predecessor of itself up to position: that is it repeats like an oscillator, but each instance is shifted across the playing field. The most famous spaceship is the *glider*: a period 4 oscillator which incorporates displacement of one cell per period in both the x and y dimensions: in a high speed animation gliders appear to race diagonally across the playing field.



If we position a single still life and a spaceship on the playing field, then we may have constructed a CGL pattern whose area will grow without limit (which presents an existence proof for the question raised in Section 1.6). We say may, because it is possible for the spaceship to move in such a way that it 'collides' with the still life, and the results of that, whilst deterministic, can be difficult to predict.

The still-life-and-spaceship example shows that CGL patterns can spread without limit, but the population remains essentially constant. In fact, for the case of a still life and a glider, the population remains exactly constant throughout (as long as the glider does not collide with the block) since all four phases of the glider's transitions have five occupied cells; however other spaceships may have varying numbers of cells in their phases, which would cause cyclic variations in the population

Do there exist patterns which grow both in extent and population? Wonderfully, yes. There is a 36×9 cell structure (found by Bill Gosper in 1970) called the *Gosper glider gun* may be viewed as an almost-oscillator which 'emits' some extra pattern. In this case, the extra structure is a glider, which moves away smoothly. Some cycles later, the gun will emit another glider which moves off on the same trajectory. Since the gliders are moving at the same speed, are on the same trajectory, and are separated when initially created, they will never collide and the pattern will grow without limit on either area or population. Conway initially conjectured that such patterns could not exist, and Gosper won a prize of $50 for his discovery. (Conway has said that he had hoped such a pattern did exist, and offered the prize as a stimulus to research in cellular automata.)

Structures called *eaters*, when hit by a glider are initially disrupted, but after some cycles return to their original state, with the glider disappearing. It is also possible to find structures that move a fixed distance when hit by a glider, and can move back again when hit by a glider on a different trajectory. Together, these may be used to make logic gates in which the information is encoded as the presence or absence of a stream of gliders.

As is well known, all binary digital logic systems may be decomposed into networks of two-input NAND gates, and so if we can build such a gate in the CGL playing space (and we can) then we can in principle simulate arbitrarily complex digital systems (up to and including Turing machines) within CGL. We say that CGL, as a computational device, is *Turing complete*.

Hence, if we were patient enough, we could design a small processor architecture, decompose it into NAND gates, lay it out on a CGL playing field, write code for that architecture which interprets the JVM instructions and then run one of the implementations discussed later in this section within the CGL game. Such an implementation then represents a sort-of higher level CGL space, in which could also host the same program, and so on to as many levels as one likes.

1.7.2 Emergent behaviour in CGL

CGL is an interesting example of so-called *emergent behaviour*: a simple game from which quite complex and interesting phenomena appear when allowed to run. In emergent systems, we deliberately construct an abstract model and 'see what it will do', hoping that higher level organisation will emerge from a few simple rules. These experiments are thought provoking, and may yield insight, but there is no direct link between our observations of the emergent system and our observations of physical reality.

This kind of emergent game is, in a sense, the philosophical dual of the scientific method. In science, we make observations, construct (usually mathematical) paramaterised models of processes which may account for those observations, and then use the model to construct predictions of behaviour for as-yet unobserved arguments to those parameters. Experimentalists then construct physical experiments that concretise new values of those arguments, and if the new observations match the model, then confidence in the model as being an accurate description of reality increases. If there is disagreement between reality and model, then the model must be changed.

Formalisation in computing sits somewhere between the two. We typically have an overall system that is too complex to be modeled as whole. Instead, we extract key algorithmic parts and build simple mathematical models for them, much as scientists do, but with the goal of increasing precision and removing ambiguity rather than 'explaining' phenomena: in fact typically we deliberately use formalisations which are divorced from our implementation. Once content, the formalisation becomes the *specification* of the system: it is the job of an implementation to then conform to that specification. From a scientist's perspective, we change our reality to fit the model.

In this book, we are using CGL in two ways: as an example of an easy formal system but also as an exemplar program for the small languages that we shall develop and a demonstration of the specification-implementation approach to software development that underpins software language engineering as a field.

1.7.3 Naïve implementation of CGL using an array

In our initial formalisation on page 10 we said that CGL is played on a rectangular grid, and that the transformation rule summed the eight connected neighbours. This leads to a natural implementation, beloved of introductory programming courses, using an array of integers.

We first need some infrastructure declarations and methods. We declare two arrays, one called `G` representing Γ, the current playing field; and one `Gp` (Γ') for the next playing field which we shall compute. At the end of each generation,we swap over Γ and Γ' using method `swapgG_Gp()`, and then display Γ using method `showG()`. (In later implementations we shall also clear Γ' back to a set of empty cells, but in this array based implementation that is not necessary.)

```
1  int extent = 10; // or some other value of your choice
2  int[][] G = new int[extent][extent]; // The current playing field Γ
3  int[][] Gp = new int[extent][extent]; // The next playing field we are computing Γ'
4
5  void setGp(int x, int y, int sigma) { // Set the state σ of an element of Γ'
6    Gp[y][x] = value;
7  }
8
9  int getG(int x, int y) { // Get the state of an element of the Γ playing field
10   if (G[y][x] > 0) return 1; else return 0;
11 }
12
13 void swapG_Gp() {// Swap Γ and Γ' (and clear Γ' in later implementations)
14   int[][] temp = G; G = Gp; Gp = temp;
15 }
16
17 void showG(int generation) { // display the state of the Γ playing field
18   System.out.println("Array " + generation + " dCount = " + dCount);
19   for (int y = 1; y < G.length; y++) {
20     for (int x = 1; x < G[0].length; x++)
21       System.out.print(getG(x, y) > 0 ? getG(x, y) : ".");
22     System.out.println();
23 }
```

We need a method which computes the population of P of the eight connected neighbours for some cell at (x, y) in Γ.

```
1  int d(int x, int y) { // return the 8-connected population of cell (x, y) in Γ
2    return getG(x − 1, y − 1) + getG(x − 1, y) + getG(x − 1, y + 1) +
3           getG(x, y − 1) + getG(x, y + 1) +
4           getG(x + 1, y − 1) + getG(x + 1, y) + getG(x + 1, y + 1);
5  }
```

We are now ready to focus on the implementation of moves. We assume that the Γ' array `Gp` has been initialised with some pattern specified by the user. At line 2 we swap the arrays and display Γ as generation zero. We are going to run the game for some fixed number of moves specified in the argument `generationLimit` so at line 3 we use a suitable `for` loop.

The body of that loop computes a new Γ' by using two nested loops at lines 4 and 5 to raster scan over the cells in Γ. At each coordinate x, y, we set variable `s` to the state (σ) of the cell, and variable `p` to the population of the eight-connected cells. We then call method `setGp()` at line 9 to set the state of cell (x, y) in Γ' according to the CGL rule.

Finally at line 12 we swap the arrays and display the new generation.

```
1  void run(int generationLimit) { // Run the CGL game for generationLimit transitions
2    swapG_Gp(); showG(0);
3      for (int generation = 1; generation <= generationLimit; generation++) {
4        for (int y = 1; y < extent - 1; y++)
5          for (int x = 1; x < extent - 1; x++) {
6            int s = getG(x, y);
7            int p = d(x, y);
8
9            setGp(x, y, (s==0 && p==3) ? 1 : (s==1 && (p<2 || p>3)) ? 0 : s);
10         }
11
12     swapG_Gp(); showG(generation);
13   }
14 }
```

An uncomfortable aspect of this implementation is that the size of the playing field is bounded by constant `extent`. Now, of course, all real computers are finite, so the fact that our implementation of CGL is restricted to a finite playing field is not in itself surprising. In addition to that, though, we have hard edges beyond which our implementation cannot 'see' and that will mean that in general our implementation misbehaves at the edges. If we run with a board of, say, 100×100 cells and then run again with a board of 200×200 cells then the games may diverge.

There are lots of ways in which this might happen: for instance a glider might emerge outside of the 100×100 board but move back towards the centre before crashing into a structure within the 100 central zone. The small version of the program would never create that glider, and would continue to evolve the central area as though it never appeared. This argument shows that in fact any finite board may diverge from the formally defined system.

Apart from this, the edges must in some way disrupt the computation of the population around a cell. In this version, we have elected to never use the outermost cells, and this policy is enforced by subscanning the array: in lines 4 and 5 above, the `for` loops scan from `1` to `extent-1`. This has the effect of surrounding our working area with a ring of empty cells which will never give birth. It is a useful compromise.

This version does not use the computer's resources very well. If we want to simulate a sigle glider moving acoss, say, a 100×100 board, then we have a total of $10,000$ cells to check (less the border) of which only five will ever be occupied. Those cells and their immediate neighbours are the only ones that can ever change state, but we shall continue to blindly compute populations for all of the cells nevertheless. We can do much better, but first we need to revisit our formalisation.

1.7.4 A better formalisation

**** Todo: Review relations and functions**

Our first attempt at formalising CGL leaves the structure of the grid only informally specified. Here is a better attempt.

> A cellular automaton is a formal system comprising a set Γ of pairs $\langle c, \sigma \rangle$ and a relation $R = \Gamma \times \Gamma$ where $\sigma \in \Sigma$ is a cell-state ($|\Sigma| < \infty$) and c is a tuple $\langle d_0, d_1, \ldots d_n \rangle, n \in \mathbb{Z}$.
>
> The transition graph of a cellular automaton has a vertex for each possible Γ. There is an edge from Γ_x to Γ_y if $\langle \Gamma_x, \Gamma_y \rangle \in R$
>
> The cellular automaton game is 'played' by starting with some Γ_0 and tracing out the transition graph by applying the transition relation.

The $d_1, \ldots d_n$ are just integers, but may be interpreted as coordinates within an n-dimensional space which forms the 'playing field' for the cellular automaton. The states form an arbitrary alphabet of symbols since Σ must be finite.

If R is a partial or complete function, then the cellular automaton is deterministic. If the cellular automaton is deterministic, then the vertices of the transition graph must have at most one out edge. Note that the transition graph need not be connected, since there may be loops. In addition, if R is a partial function, then there must be a Γ_s which appears as the second element of some pair in R but not as the first element of any pair in R, and thus on arrival at Γ_s the game must stop.

If R is not a function, then the graph of the relation will have vertices with multiple out-edges; we could no longer represent such a game with a linear sequence(s) of playing field pictures.

This notion of a game with states and a transition relation is one we shall return to when we consider formalising the semantics of a programming language. Using an approach based on Plotkin's *Structural Operational Semantics* we shall represent the execution of a program as the evolution of a computer's state under the control of a transition relation.

In CGL, the tuple c has two elements (which we think of as dimensions), the cell-state set Σ has only two values {1,0}, and the transition relation is computed by applying this function to every element $\langle (x, y), \sigma \rangle$ of Γ:

$$f((x,y),\sigma) = ((x,y),\sigma') \quad \text{where} \quad \sigma' = \begin{cases} 1, & \sigma = 0, P(x,y) = 3 \\ 0, & \sigma = 1, P(x,y) < 2 \vee P(x,y) > 3 \\ \sigma, & \text{otherwise} \end{cases}$$

where

$$P(x,y) = \begin{array}{l} \sigma(x-1,y+1)+\sigma(x-1,y)+\sigma(x-1,y-1)+ \\ \sigma(x,y+1)+\sigma(x,y-1)+ \\ \sigma(x+1,y-1)+\sigma(x+1,y-1)+\sigma(x+1,y-1) \end{array}$$

and $\sigma(x,y)$ is the value of state σ in tuple $\langle(x,y),\sigma\rangle$

**** Todo: Eamples of fragments of the trnsition relation graph such as still life and osciallator**

**** Todo: Note that CGL relation is a function; compare with chess**

When we come to model programming language semantics, we shall be working with tuples that contain a program and objects representing the store, the input and the output. The transition function will 'reduce' the program term by, for instance, removing an assignment from the program whilst changing the store object to represent the effect of that assignment. By running through a sequence of transitions we shall build a trace of the execution of a program written in, say, Java by applying a transition relation R_j that encodes the execution-time behaviour of Java programs, just as we display a series of CGL playing fields by applying the CGL transition relation. We shall call the transition relation R_j a *formal semantics* for Java. Defining the transition relation can be challenging. The basic approach is to inductively define the relation by constructing a rule for each feature of the language and allowing the programming language's compositional structure to guide the evaluation order. We shall study this approach in detail in Chapter **??**.

1.7.5 Improved formalisation can improve implementations

This improved formalisation (as a set of tuples rather than a geometrically organised set of cells with states) immediately suggests a new implementation strategy. It turns out, perhaps surprisingly, that this more abstract implementation of CGL is significantly more efficient than the 'natural' implementation of the geometrically based formalisation.

At the heart of the new implementation is a *map* from c to σ. We can ask the map what the value of σ is at coordinate c, which thus implements the function $\sigma(x,y)$ above. The complete list of mapping elements constitutes the set Γ.

Now, our computers are finite, but cellular automata playing fields are typically infinite, so we cannot directly implement the set-of-tuples formalism. However, we can select one element of $\sigma_0 \in \Sigma$ as a default 'inactive' state and only store active tuples in our implementation. In this encoding, we can ask the map for $\sigma(x,y)$ and if the map contains no element for (x,y) then we assume $((x,y),\sigma_0)$. Note that this approach only works if there are no 'holes' in the playing field which would genuinely need to be represented as coordinates that map to nothing: our formalisation does in principle allows this sort of game if the relation R is defined over only a subset of the elements of c.

In the case of CGL, there are only two states, and the playing field has no holes. Let us select state $\sigma_0 = 0$ to be the inactive state. That means that for

CGL, every element of our map will be from some coordinate to state $\sigma = 1$, so in fact our map degenerates to a set: we simply need to store the coordinates of our active cells on Γ and Γ'.

We change the Java code so that the variables `G` and `Gp` are sets of coordinates, and make a class `Coord` to represent a coordinate (which simply has two fields for CGL) which hold the x and y coordinates of the element.

```
1 Set<Coord> G = new HashSet<Coord>();
2 Set<Coord> Gp = new HashSet<Coord>();
```

```
1 class Coord {
2   int x, y;
3
4   public Coord(int x, int y) { super(); this.x = x; this.y = y;}
5
6   public int hashCode() ...
7   public boolean equals(Object obj) ...
8   public String toString() ...
9 }
```

and make the corresponding changes to the `setGp()`, `getG()`, `swapG_Gp()` and `showG()` methods. The method `d()` which computes the population of the neighbours does not change, since it uses `getG()` which encapsulates the access to the playing field.

In the run method, we know that we must check every element that is in the set to see if its neighbourhood population allows it to live, but we also need to check *some* elements that are not in the set. In CGL, an empty cell can only be replaced by a filled cell if there are filled cells in its neighbourhood; that means that the only empty cells we need to consider are those immediately bordering each filled cell. We use the Java *for-each* construct at line 4 to iterate over the coordinates c in Γ, and then we check a 3×3 block off coordinates centred on c using a raster scan generated by the inner loops at lines 5 and 6.

```
1  void run(int generationLimit) { // Run the CGL game for generationLimit transitions
2    swapG_Gp(); showG(0);
3    for (int generation = 1; generation <= generationLimit; generation++) {
4      for (Coord c : G) {
5        for (int y = c.y - 1; y <= c.y + 1; y++)
6          for (int x = c.x; x <= c.x + 1; x++) {
7            int s = getG(x, y);
8            int p = d(x, y);
9
10           setGp(x, y, (s==0 && p==3) ? 1 : (s==1 && (p<2 || p>3)) ? 0 : s);
11         }
12     }
```

```
13        swapG_Gp(); showG(generation);
14    }
15  }
```

1.7.6 Using a visited set to avoid recomputation

There is an easy further optimisation. Although the set based implementation avoids scanning empty cells which have only empty cells as neighbours, it will in general repeatedly recompute coordinates because we raster scan around every occupied cell. We add a further set of coordinates `checked`. Every time we compute a cell we store its coordinates in *checked*; just before we compute a cell we check to see if its coordinated are in `checked`, and if they are we continue with the next element.

```
1  void run(int generationLimit) { // Run the CGL game for generationLimit transitions
2    swapG_Gp(); showG(0);
3    for (int generation = 1; generation <= generationLimit; generation++) {
4      checked.clear();
5      for (Coord c : G) {
6        for (int y = c.y − 1; y <= c.y + 1; y++)
7          for (int x = c.x − 1; x <= c.x + 1; x++) {
8            Coord nc = new Coord(x, y);
9            if (checked.contains(nc)) continue;
10           checked.add(nc);
11
12           int s = getG(x, y);
13           int p = d(x, y);
14
15           setGp(x, y, (s==0 && p==3) ? 1 : (s==1 && (p<2 || p>3)) ? 0 : s);
16         }
17     }
18     swapG_Gp(); showG(generation);
19   }
20 }
```

If we instrument these three implementations to count the number of times the `d()` method is called to compute a neighbourhood, we can get a sense of effectiveness of these optimisations. For example, the code that generated the glider figures on page 11 used a 10×10 grid containing a single glider. The array based implementation computes 64 cells per generation, as might be expected since the 10×10 array has only an 8×8 active area as a result of the sub-scanning to reduce edge effects. The set implementation for this example computes 30 cells for each generation, and the final implementation with the additional check set computes only 22 cells per generation.

These are substantial savings, and of course with larger boards the effects might be even more extreme since the space and time complexity of the array

implementation will be proportional to the area of the playing field, whereas the set and checked set implementations will have both time and space requirements that are broadly proportional to the number of active cells.

1.8 Thinking formally about program execution

Programming languages grew out of attempts to reduce the clerical overhead of writing machine level programs for von Neumann style architectures. In their simplest form, these machines comprise a single central processor coupled to a memory S, an input sequence I and an output sequence P.

The memory may be thought of as a finite set of cells, each of which has a fixed index number called its address. Just as in our cellular automata, there is a finite set of cell-states or *values* and each cell will display exactly one of those values at any one time. Modern hardware typically uses cells which can have one of only 256 states, but high level programming languages suppress this reality by allowing the programmer to define their own sets of values (or *user defined datatypes*) and by providing variables that can be set to any of those values. These values are then mapped onto sequences of the underlying hardware values automatically, and those sequences may be stored in contiguous elements of the hardware store. Since we are mostly thinking about high level languages in this book, we shallow allow our store to directly map identifiers to an arbitrary finite set of values.

When thinking in an abstract, formal way, we call this structure a *store*, a set of tuples $\langle a, \sigma \rangle$ where $\sigma \in \Sigma$ is a cell-state ($|\Sigma| < \infty$) and $a \in \mathbb{N}$ (or, more concisely, $S = \{\langle a, \sigma \rangle \mid \sigma \in \Sigma, |\Sigma| < \infty, a \in \mathbb{N}\}$. Thus the store looks just like a playing field for a cellular automaton which is one-dimensional.

The input can take many forms, but in this simplest of cases we shall assume that it is just a sequence (or list) of elements from Σ. We write lists in square brackets: $[\sigma_1, \sigma_2, \ldots, \sigma_n]$ and say that n is the length of the sequence, σ_1 is the *head* of the list, and σ_n is the *tail* of the list. The concatenation of two lists is written as $\alpha_1 : \alpha_2$.

Reading from the input I means taking the head of I and replacing with a new input sequence I' comprising the rest of the list. Similarly, the output P is an initially empty sequence of elements from Σ. When we output a value σ, we replace P with $P' = P : [\sigma]$.

We can think of the instantaneous state of an entire von Neumann computer's data resources as a tuple $\Gamma = \langle I, S, P \rangle$. Note that we have said nothing about programs yet, but as the computer executes a program we can think of it generating a sequence of sates $\Gamma_1, \Gamma_2, \ldots$ just as we thought of a cellular automaton generating a sequence of playing fields. The program ultimately represents a transition relation which tells us how to get from one data state to another, that is from some Γ_j to some Γ_k (or at least it does for a so-called sequential computer in which we hope that the transition relation is a function; however if we wanted to model concurrency and the spawning of independently executing sub-processes or threads, then we might expect the relation to not be a function).

We shall now make the discussion a little more concrete by considering a real algorithm specified using a very simple programming language.

1.8.1 Euclid's Greatest Common Divisor algorithm

The algorithm we shall use is Euclid's integer Greatest Common Divisor method, described in the second proposition of *Elements VII* some 2,300 years ago. It is worth looking up the original description which is written in quite verbose prose. For many of the examples in this book we a programming language called *Cava*, thus named because it is neither C nor Java but will accept programs that are quite C/Java-like in their appearance.

```
a := input();
b := input();

while a != b
    if a > b
        a := a − b;
    else
        b := b − a;

output(a);
```

We shall discuss Cava in more deatil in Chapter **??**. For now, we note assignment to a variable is denoted by `:=`, not by `=`. As in C and Java statements are terminated with (not separated by) a `;` The phrase `input()` reads a 32-bit integer value from the standard input stream, and `output(a)` appends a textual representation of the value of `a` to the output sequence. The names of numeric types indicate their precision: is equivalent to `Integer` in Java. Variable names are not pre-declared.

For comparison, here is the above Cava example turned into a runnable Java program: it is rather cluttered with extra syntax which is not significant from our perspective.

```
import java.util.Scanner;

public class GCD {
   public static void main(String[] args) {
      int a, b;
      Scanner input = new Scanner(System.in);

      a = input.nextInt();
      b = input.nextInt();

      input.close();

```

```
13      while (a != b)
14        if (a > b)
15          a = a - b;
16        else
17          b = b - a;
18
19      System.out.println(a);
20    }
21  }
```

1.8.2 The fixed-code-and-program-counter interpretation

When we are developing software, we write code, load it into a development environment such as Eclipse and then run it to see if its behaviour matches our expectations. Figure 1.1 shows a screenshot of the Java CGD program being run under the debugger within Eclipse.

We can see the program's Java code, the input $[9, 6]$ (in green) and the state of the store with variables `a` and `b`, both presently mapped to the value 3. The program has stopped just before executing the output statement. I can tell this because line 19 of the code window has a small pointer arrow on the left hand side and is highlighted in grey. In this system, I can execute one more line of code (moving the pointer to line 20) by pressing the 'step-over' button.

This idea of code which is essentially fixed during a program's execution along with the use of a pointer (called the *program counter* or PC) is fundamental to most programmers' ideas of how computers operate. It is a direct legacy of the von Neumann architecture that is used in almost all modern computing devices: in these machines most programs are indeed static lists of instructions that reside in the store; and the instructions for a particular program do not change as it is being executed. The dynamic aspects of a program's execution are under the control of the PC which points to the next piece of code to be executed: at a branch point we may test a condition and update the PC with one of two values depending on that outcome. The sequence of values displayed by the program counter during a program's execution constitutes the *control flow* for this particular input. (We shall sometimes talk about the control flow of a *program* which is the union of the control flows exhibited by every possible input.)

This static-code-and-program-counter model breaks down for some situations. Firstly, since the instructions reside in the store, it is entirely possible for a running program to change its own instructions, and indeed some early processor architectures relied on this idea when executing subroutines. Such *self-modifying code* is widely recognised as being very difficult to reason about, and as a result almost no high level languages allow it. However, operating systems, and programming environments (such as Java) which allow dynamic loading of classes at run time do effectively present a form of self modifying code in a restricted way: we allow a completely new block of code to be loaded (possibly replacing an existing block or subprogram) and then pass control to

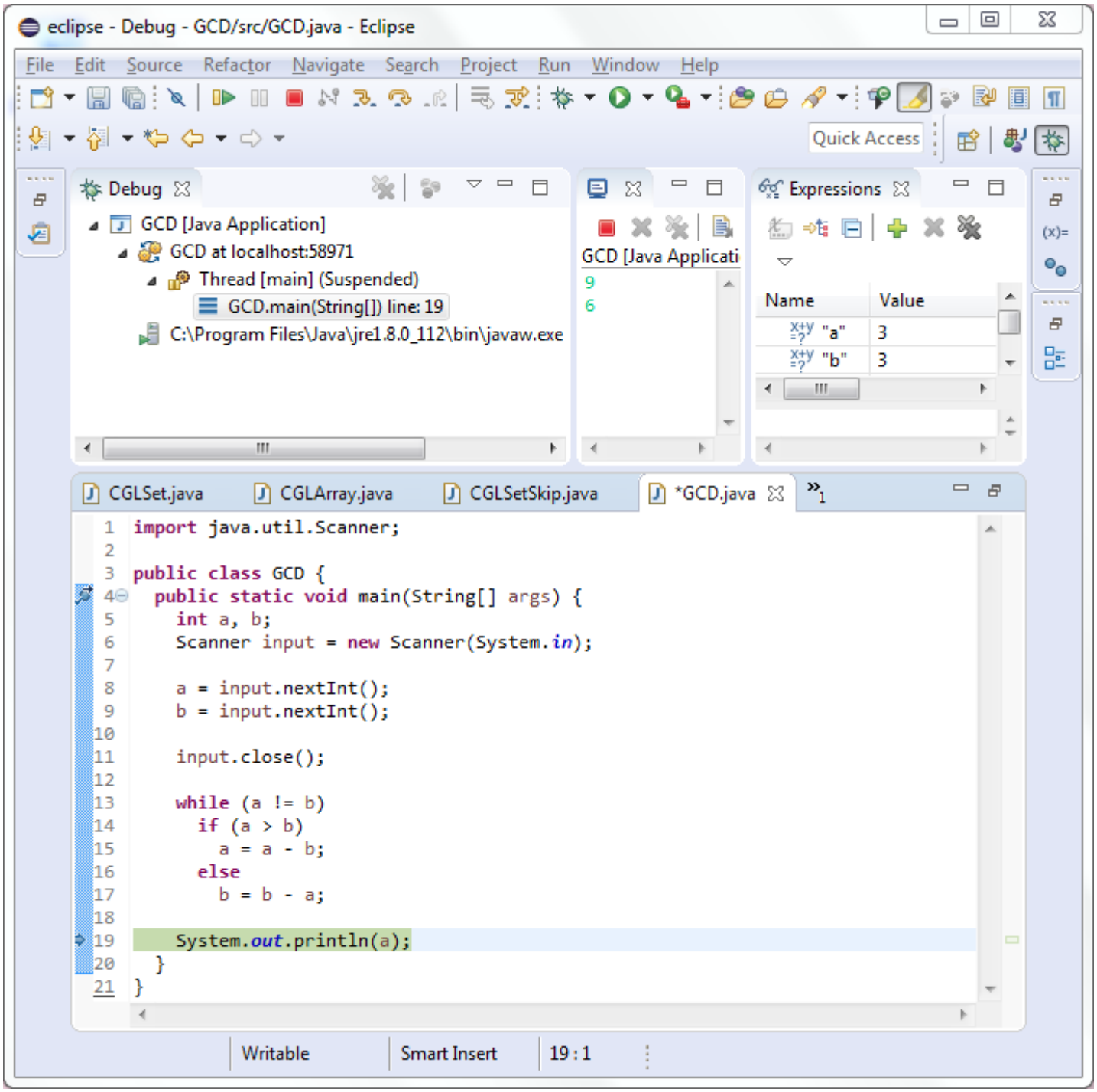


Figure 1.1 Java GCD implementation during an Eclipse debug session

it. During a load, the code is treated as passive data, and only once fully installed do we allow the PC to access its contents. We do not allow individual instructions within an executing piece of code to be changed. The hardware often enforces a policy in which the store is internally divided into *blocks* and at any one time a block is either read-only or writable. During a code load, the target blocks are made writable, but then changed to read-only when the code has been integrated. The PC is never allowed to hold an address from a writable block.

Now, although the hardware works with (mostly) static code and a program counter, that does not mean that a formal model of program execution must take the same view. Just as aviation pioneers had to learn that wing-flapping was not a useful way to get humans airborne (propellers and jet engines being a better engineering compromise) the pioneers of formal approaches to programming language semantics had to find a way of dispensing with the program counter. Why is this?

1.8.3 What is equality?

It turns out that the *substitution model of equality* that is used in most mathematical reasoning is much simpler than the *assignment model of equality* used in procedural programming languages. In mathematics, if I say $x = 3$ I mean that x and 3 are synonyms, and in fact anywhere that x appears subsequently I could cross it out and write 3. In procedural programming languages like Java, if I write $x = 3$ I may subsequently write $x = 4$, and so the relationship between x and its value depends on the most recent assignment to x under the history execution history for a particular input.

The substitution model is simple and easy to reason about; the assignment model is efficient in that identifiers (in detail, named cells with machine addresses) may be re-used rather than having to be maintained throughout the runtime of a program. There are languages that use substitution semantics: they are loosely called functional languages; Haskell is perhaps the purest of the functional languages. Other mostly-functional languages such as ML and Scheme do allow assignment, but the culture of programmers in those systems discourages assignment. In procedural languages, assignments are probably the most common operation performed during execution.

The use of assignment presents a challenge to formal analyses of program semantics, but it is particularly problematic that the program counter itself works by assignment. If we adhere strictly to von Neumann dogma, then we cannot even execute *functional* code without using assignment, and this is very uncomfortable.

1.8.4 The reduction interpretation

There is a straightforward way of thinking about program execution that does not require the use of a program counter. The trick is to think of the program code itself as something that can be progressively rewritten until all that we have left is a result.

Consider this Cava program fragment

```
1 output(3);
2 output(10+2+4);
```

The first thing the program does is output the value 3. We can represent this by constructing the output list [3] and then discarding the first line of the program (since we do not need it again). There is a sense in which the tuple

⟨"output(3); output(10+2+4);", []⟩

means the same thing as

⟨"output(10+2+4);", [3]⟩

because the externally visible effect of starting with an empty output and executing lines 1 and 2 above is the same as starting with the output [3] and only executing line 2. Let us therefore represent each step of a program's execution by a pair comprising the output and a program that represents only what remains to be done:

```
1 output(10+2+4); [3]
```

Now we have to evaluate the expression 10+2+4 before we can execute the next output statement. In detail, the computer can only execute one arithmetic operator at a time; let us choose to execute 10+2, and rewrite it to the result 12.

```
1 output(12+4); [3]
```

Now we do the other arithmetic operation: 12+4 is rewritten to 16.

```
1 output(16); [3]
```

Finally we can execute the output statement, and add 16 onto the end of the output list.

```
1 [3, 16]
```

Execution is now complete. Note that we could start in any of the five states above and end up with the same output.

We call this kind of display of machine states the *reduction semantics transition graph* for our program. Just as a sequence of CGL playing fields is a fragment of the graph of the transition relation for CGL, this trace is a fragment of the graph of the transition relation for Cava which is defined over tuples of ⟨program, output⟩. It is a precise, architecture independent description of the step-by-step evaluation of our program. Do note, though, that we have not

yet said anything at all about how that transition relation may be practically specified. In this example, and in the larger example in the next section, we have simply chosen plausible operations informally. Later on this book we shall use sets of *inference rules* to define the relation in a way which would allow the automatic generation of this style of reduction interpreter; and that will in fact be a true symbol-pushing formulation.

1.8.5 A reduction evaluation of GCD with input [6, 9]

A reduction semantics is so-called because we attempt to rewrite programs to values, which usually means replacing part of the program term with a smaller one, and thus reducing the program. Occasionally, though, we will actually rewrite terms to longer terms.

We now present the reduction semantics trace for the GCD program running on input $[6, 9]$ – exactly the program and input shown in the debugger screenshot in Figure 1.1 . There are 36 steps in this trace, which make for intimidating reading, but bear in mind that a step (very roughly) corresponds to a machine operation such as fetching an operand or adding two numbers. Useful programs entail the execution of a *lot* of operations: some of the programs we run on modern processors take an appreciable amount of time to execute even though a 3GHz processor will, in two seconds, execute one instruction for every person on the planet — a number well beyond our abilities to directly comprehend. This is just a roundabout way of saying that machine operations are fine grained, and we need an awful lot of them to do useful work. Any attempt to list all of the steps that are gone through by a non-trivial running program is going to generate a long list.

We shall use a slightly more compact form to display the steps. First, we shall write the entire program term on a single line: rather than the nicely laid out version shown on page 20, we say

```
a:=input(); b:=input(); while a!=b if a>b a:=a−b; else b:=b−a; output(a);
```

In our initial example of a reduction trace, the complete state of the machine could be represented as a tuple containing a program term and an output list. All of the calculations used constants, so there was no need to represent the store, or any input. In the GCD program, we *shall* need these entities, so our trace will be a sequence of tuples $\langle I, S, P, T\rangle$ displaying the current values for the input, store, output and term, respectively. The initial term has input $[6, 9]$, an empty store and an empty output:

$\langle [6, 9], \{\,\}, [\,],$ `a:=input(); b:=input(); while a!=b if a>b a:=a−b; else b:=b−a; output(a);` $\rangle$

A physical store is a fixed set of cells, each with a fixed address but containing a value which may be changed. One mathematical model of a store is a map from identifiers to values, and we only put into the map those identifiers we need. Evaluating a *declaration* in the program term has the effect of creating a new store S' from S which has all of the bindings in S *and* the new binding required by the declaration. Assigning a new value to a variable has the effect

of changing the mapping of one variable in the store, and using a variable in an expression requires us to look up the value mapped to the variable's identifier. We use the notation $X \mapsto y$ for an element of S, and the special symbol $\perp$ (read as 'bottom') to represent the special value 'none'. A declaration of X with no associated initialisation of X creates a binding $X \mapsto \perp$. In Cava, declarations are usually implicit, in that the first time we encounter an assignment to a variable x, we declare x and perform the assignment together.

Each step of our trace involved identifying a part of the program term that we shall execute, and then rewriting the program term to represent what is left to do of the original term. We call the subterm that is to be replaced a *reducible expression* or *redex* for short. In the trace below, we have highlighted the chosen redex in red at each stage. Sometimes there is a choice of redexes available: for instance when processing the GCD program declarations for `a b`, it does not mater which order we process them in. We have chosen to do `b` first.

A reduction semantics for linear code is straightforward, but we need to think carefully about loops. The approach we have taken here is to make use of a *program identity*, that is a program transformation that does not change the semantics of a program term, but does change the syntax, and thus the reduction trace. If we have a loop of the form

```
1 while booleanExpression do statement;
```

then we can always transform it into

```
1 if booleanExpression { statement; while booleanExpression do statement; }
```

We have effectively unpacked the first iteration of the loop and are handling it directly with an **if** statement followed by a new copy of the `while` loop which will compute any further iterations. When we have completed all of the iterations we shall encounter a term like

```
1 if false { statement; while booleanExpression do statement; }
```

which rewrites to the empty subterm. This device, then, allows us to treat **while** loops using only **if** statements.

When reading the reduction trace below, the bold headings should simply be treated as comments: they are there to break up the reductions into related blocks as an aid to comprehension and have no part in the formal, symbol-pushing, description of program execution. At each step i, look for the highlighted redex: the tuple for step $i + 1$ should contain a term which has all of the non-highlighted parts from step i, and some new (possibly empty) subterm which has replaced the redex. The entities will display any changes arising from side effects of the reduction, such as reading input, declaring a variable, redefining the value of a variable or writing to output.

Start of trace

Initialise variables from input

$\langle [6,9], \{\}, [\,], \texttt{a:=input(); b:=input(); while a!=b if a>b a:=a−b; else b:=b−a; output(a);}\rangle$

$\langle [9], \{a \mapsto 6\}, [\,], \texttt{b:=input(); while a!=b if a>b a:=a−b; else b:=b−a; output(a);}\rangle$

Rewrite using `while p s` → `if p { s ; while p s }`

$\langle [\,], \{a \mapsto 6, b \mapsto 9\}, [\,], \texttt{while a!=b if a>b a:=a−b; else b:=b−a;output(a);}\rangle$

Evaluate $a \neq b$ with store $\{a \mapsto 6, b \mapsto 9\}$

$\langle [\,], \{a \mapsto 6, b \mapsto 9\}, [\,], \texttt{if a!=b \{ if a>b a:=a−b; else b:=b−a; while a!=b if a>b a:=a−b; else b:=b−a; \} output(a);}\rangle$

$\langle [\,], \{a \mapsto 6, b \mapsto 9\}, [\,], \texttt{if 6!=b\{ if a>b a:=a−b; else b:=b−a; while a!=b if a>b a:=a−b; else b:=b−a; \} output(a);}\rangle$

$\langle [\,], \{a \mapsto 6, b \mapsto 9\}, [\,], \texttt{if 6!=9\{ if a>b a:=a−b; else b:=b−a; while a!=b if a>b a:=a−b; else b:=b−a; \} output(a);}\rangle$

$\langle [\,], \{a \mapsto 6, b \mapsto 9\}, [\,], \texttt{if true \{ if a>b a:=a−b; else b:=b−a; while a!=b if a>b a:=a−b; else b:=b−a; \}output(a);}\rangle$

Evaluate $a > b$ with store $\{a \mapsto 6, b \mapsto 9\}$

$\langle [\,], \{a \mapsto 6, b \mapsto 9\}, [\,], \texttt{if a>b a:=a−b; else b:=b−a; while a!=b if a>b a:=a−b; else b:=b−a; output(a);}\rangle$

$\langle [\,], \{a \mapsto 6, b \mapsto 9\}, [\,], \texttt{if 6>b a:=a−b; else b:=b−a; while a!=b if a>b a:=a−b; else b:=b−a; output(a);}\rangle$

$\langle [\,], \{a \mapsto 6, b \mapsto 9\}, [\,], \texttt{if 6>9 a:=a−b; else b:=b−a; while a!=b if a>b a:=a−b; else b:=b−a; output(a);}\rangle$

$\langle [\,], \{a \mapsto 6, b \mapsto 9\}, [\,], \texttt{if false a:=a−b; else b:=b−a; while a!=b if a>b a:=a−b; else b:=b−a; output(a);}\rangle$

Evaluate $b - a$ with store $\{a \mapsto 6, b \mapsto 9\}$

$\langle [\,], \{a \mapsto 6, b \mapsto 9\}, [\,], \texttt{b:=b−a; while a!=b if a>b a:=a−b; else b:=b−a; output(a);}\rangle$

$\langle [\,], \{a \mapsto 6, b \mapsto 9\}, [\,], \texttt{b:=b−6; while a!=b if a>b a:=a−b; else b:=b−a; output(a);}\rangle$

$\langle [\,], \{a \mapsto 6, b \mapsto 9\}, [\,], \texttt{b:=9−6; while a!=b if a>b a:=a−b; else b:=b−a; output(a);}\rangle$

$\langle [\,], \{a \mapsto 6, b \mapsto 9\}, [\,], \texttt{b:=3; while a!=b if a>b a:=a−b; else b:=b−a; output(a);}\rangle$

Rewrite using `while p s` → `if p { s ; while p s }`

$\langle [\,], \{a \mapsto 6, b \mapsto 3\}, [\,], \texttt{while a!=b if a>b a:=a−b; else b:=b−a;output(a);}\rangle$

Evaluate $a \neq b$ with store $\{a \mapsto 6, b \mapsto 3\}$

$\langle [\,], \{a \mapsto 6, b \mapsto 3\}, [\,], \texttt{if a!=b \{ if a>b a:=a−b; else b:=b−a; while a!=b if a>b a:=a−b; else b:=b−a; \} output(a);}\rangle$

$\langle [\,], \{a \mapsto 6, b \mapsto 3\}, [\,], \texttt{if 6!=b\{ if a>b a:=a−b; else b:=b−a; while a!=b if a>b a:=a−b; else b:=b−a; \} output(a);}\rangle$

$\langle [\,], \{a \mapsto 6, b \mapsto 3\}, [\,], \texttt{if 6!=3\{ if a>b a:=a−b; else b:=b−a; while a!=b if a>b a:=a−b; else b:=b−a; \} output(a);}\rangle$

$\langle [\,], \{a \mapsto 6, b \mapsto 3\}, [\,], \texttt{if true \{ if a>b a:=a−b; else b:=b−a; while a!=b if a>b a:=a−b; else b:=b−a; \} output(a);}\rangle$

Evaluate $a > b$ with store $\{a \mapsto 6, b \mapsto 3\}$

$\langle [\,], \{a \mapsto 6, b \mapsto 3\}, [\,], \texttt{if a>b a:=a−b; else b:=b−a; while a!=b if a>b a:=a−b; else b:=b−a; output(a);}\rangle$

$\langle [\,], \{a \mapsto 6, b \mapsto 3\}, [\,], \texttt{if 6>b a:=a−b; else b:=b−a; while a!=b if a>b a:=a−b; else b:=b−a; output(a);}\rangle$

$\langle [\,], \{a \mapsto 6, b \mapsto 3\}, [\,], \texttt{if 6>3 a:=a−b; else b:=b−a; while a!=b if a>b a:=a−b; else b:=b−a; output(a);}\rangle$

$\langle [\,], \{a \mapsto 6, b \mapsto 3\}, [\,],$ **if true** a:=a−b; **else** b:=b−a; **while** a!=b **if** a>b a:=a−b; **else** b:=b−a; output(a);$\rangle$

Evaluate $a - b$ with store $\{a \mapsto 6, b \mapsto 3\}$

$\langle [\,], \{a \mapsto 6, b \mapsto 3\}, [\,],$ a:=a−b; **while** a!=b **if** a>b a:=a−b; **else** b:=b−a; output(a);$\rangle$

$\langle [\,], \{a \mapsto 6, b \mapsto 3\}, [\,],$ a:=a−3; **while** a!=b **if** a>b a:=a−b; **else** b:=b−a; output(a);$\rangle$

$\langle [\,], \{a \mapsto 6, b \mapsto 3\}, [\,],$ a:=6−3; **while** a!=b **if** a>b a:=a−b; **else** b:=b−a; output(a);$\rangle$

$\langle [\,], \{a \mapsto 6, b \mapsto 3\}, [\,],$ a:=3; **while** a!=b **if** a>b a:=a−b; **else** b:=b−a; output(a);$\rangle$

Rewrite using `while p s` → `if p { s ; while p s }`

$\langle [\,], \{a \mapsto 3, b \mapsto 3\}, [\,],$ **while** a!=b **if** a>b a:=a−b; **else** b:=b−a;output(a);$\rangle$

Evaluate $a \neq b$ with store $\{a \mapsto 3, b \mapsto 3\}$

$\langle [\,], \{a \mapsto 3, b \mapsto 3\}, [\,],$ **if** a!=b { **if** a>b a:=a−b; **else** b:=b−a; **while** a!=b **if** a>b a:=a−b; **else** b:=b−a; } output(a);$\rangle$

$\langle [\,], \{a \mapsto 3, b \mapsto 3\}, [\,],$ **if** 3!=b{ **if** a>b a:=a−b; **else** b:=b−a; **while** a!=b **if** a>b a:=a−b; **else** b:=b−a; } output(a);$\rangle$

$\langle [\,], \{a \mapsto 3, b \mapsto 3\}, [\,],$ **if** 3!=3{ **if** a>b a:=a−b; **else** b:=b−a; **while** a!=b **if** a>b a:=a−b; **else** b:=b−a; } output(a);$\rangle$

$\langle [\,], \{a \mapsto 3, b \mapsto 3\}, [\,],$ **if false** { **if** a>b a:=a−b; **else** b:=b−a; **while** a!=b **if** a>b a:=a−b; **else** b:=b−a; } output(a);$\rangle$

$\langle [\,], \{a \mapsto 3, b \mapsto 3\}, [\,],$ output(a);$\rangle$

Evaluate output

$\langle [\,], \{a \mapsto 3, b \mapsto 3\}, [\,],$ output(3);$\rangle$

Empty term indicates normal (successful) termination

$\langle [\,], \{a \mapsto 3, b \mapsto 3\}, [3], \rangle$

End of trace

The process terminates when we get to a term for which no further reductions are available, that is, a term that contains no redexes. We call such terms *normal forms*. In this case, the final term is empty, which naturally has no redexes.

Upon termination, the output list shows [3] which is indeed the greatest common divisor of 6 and 9.

1.9 Next steps

We now have the language for thinking about programs as specifications for transition system in which a program term is gradually reduced to a normal form, with side effects being registered in auxiliary entities. However, we have not described exactly how redexes are recognised, or what form the specification of the transition relation should take. We need both if we are to write software tools which can take a reduction semantics specification and a program term and then work out the reduction trace without human intervention — an automatic interpreter for this style of programming language specification. Much of the

rest of this book is devoted to the development of the formal machinery and generator tools needed for this task.

In a separate thread, we shall look at the broad diversity of software languages and execution styles that have appeared since 1952, when Glennie developed the autocode for the Manchester Mk I computer. From the perspective of the current day, we can identify the first appearances of programming language concepts (or *notions*) such as user defined datatypes and control flow via recursion that are now present in most production languages, and we can also see the development of various forms of syntax or *notations* for writing specifications. We must also consider the manner in which the language is executed: some languages (such as XML) describe data layout rather than computation; other languages are interpreted as a side effect of translation and yet others are translated directly into a machine level language or *compiled* for direct execution. A modern Java Virtual Machine exhibits some of the characteristics of all three: the basic JVM interpreter collects statistics on program execution, and for pieces of code which have been intensively used it will stop and create compiled versions. This complexity militates against traditional treatments of compilers and interpreters which tend to focus on one particular style of execution: in this book we shall use a *formal first* approach in which we set up abstract models of language translation that may then be used to generate concrete translators in these various styles.

1.10 Exercises

1. Create compilable versions of the array, set and set-with-check implementations of CGL. (You will need to embed the source code within a class wrapper, add a suitable `main()` method, fill in the missing definitions in class Coord, and produce variants of the infra structure methods that use the **Set** rather than the array representation.

2. Add instrumentation to your CGL programs to count the number of calls to the `d()` method performed in each generation and replicate the results given in Section 1.7.5 for the first eight generations of a glider.

3. Add a method which fills the playing field with a random selection of occupied cells, and generate statistics showing the time and space requirements of the three implementations for various sizes of playing field.

4. Write a Cava program which repeatedly reads three integers from the input and checks to see if they are a Pythagorean triplet, terminating when such a triplet is found. Write out the reduction semantics transition graph (the *trace*) when running on an input of six integers, the last three of which form a pythagorean triple.

2 Rewriting

Sometimes things look different but mean the same thing. For instance the mathematical expression $3 + 4$ evaluates to the same result as $4 + 3$. If we are only interested in the result of an expression, then we say they are *equal*, and we can write $3 + 4 = 4 + 3 = 7$.

If we are being very careful, then we would say that the expressions are equal *up to evaluation*. In some contexts, these expressions would not be thought of as equal. For instance the expression $3 + 4$ comprises three characters, and the expression 7 only one, so if we are interested in how much storage we need in a computer to hold an expression, then 7 is not equal to $3 + 4$.

An *equation* is two expressions separated by the *equality symbol* $=$. At a fundamental level, this tells is that the two expressions either side are interchangable because they evaluate to the same mathematical object, and that means that we can freely replace one by the other. It turns out that we can do a great deal of useful mathematics (and useful program translation) just by using equations.

For instance, imagine that we are given two complicated looking expressions and asked to decide if they are the same. Consider for instance the logical expressions

$$a \wedge b...$$

Now we know a few facts about Boolean algebra.

2.1 Equality of programs

In programming languages we are used to the idea of 'equivalent' programs. For instance, this Java loop:

```
for (int i = 1; i < 10; i++) System.out.print(i + " ");
```

generates the same output as

```
int i = 1; while (i<10) { System.out.println(i + " "); i++ }
```

If all we are interested in is output of a program, we might say that these two fragments are *equal* up to output, or just output-equal. More loosely, we often say that two programs are *semantically equivalent* if they produce the same effects. In this example, the iteration bounds are constant, and we could just have written

```
System.out.println("1 2 3 4 5 6 7 8 9 ")
```

These three fragments are semantically equivalent, but the third one will almost certainly run faster as it does not have the overhead of the loop counter and only makes one call to `println()`. Our notion of semantically equivalent does *not* include performance, but only the values computed by a program.

Code improvement

High quality translators for general purpose programming languages typically attempt to improve program fragments by surveying semantically equivalent alternatives, and selecting ones that are improvements with respect to some criteria. In the literature, these tools are usually called *optimising* compilers which is something of a misnomer since in general it is very hard to find a truly optimal implementation: perhaps they should be called code-improving compilers.

The conventional optimisation criteria are (i) execution speed, (ii) memory consumption and (iii) energy consumption. These three are not independent; for instance we can often speed things up by using more memory. Small battery operated systems will emphasise (iii) and (ii) over (i); high performance scientific computations such as weather prediction will emphasise (i).

In this book we are mostly interested in the meaning of programs up to, but not including, their performance, so we will have no more to say about code improvers and optimising compilers. However, there is a vast research literature describing often-ingenious techniques for improving program performance that you might wish to explore.

2.2 Mathematical objects, their denotations and software implementations

When thinking about programming languages, we need to carefully distinguish between (a) mathematical objects, (b) the textual forms (the denotations) that we use to name and manipulate those objects, and (c) the implementation of those objects inside a computer.

Mathematical objects When we are thinking mathematically, we are usually *imagining* abstract objects and operations regardless of whether we can make a concrete example. For instance we might decide to think about the set of all prime numbers, even though we have no easy way of deciding what the elements of that set are. We can give it a name (a denotation) and then go about investigating its properties: for instance Euclid proved that there must be infinitely many primes.

Denotations When we are communicating about mathematics or programs we need conventions that enable us to write down what we mean. Consider the mathematical object that we get by adding unity to zero six times: we might denote

that as 6, 06, *six* or *vi* (in Roman numerals). Which form we use is just a convention, and real programming languages usually support more than one convention: for instance Java allows us to write six as `6`, `06`, `0x06` or `0_6` and these are all denotations for the same mathematical object.

Implementations When we are programming a computer to perform addition we need some sort of *implementation* of an integer. Sadly, our implementations will never have the same properties as the mathematical integers, because our computers are finite. As a result in our programs there will always be some integer which, if we add one to it, will not generate the integer that mathematically we would expect. So, for instance, if we were using an eight-bit two's complement implementation of the integers, then $126 + 2$ would not generate 128 as that needs nine bits for its two's complement representation. On many systems, only the eight least significant bits would be retained, yielding `-128`. Some systems have so-called *saturated* addition in which case the outcome would be 127 (the largest positive number in that representation).

Note that even using arbitrary precision representations for integers such as Java's BigInteger we cannot faithfully represent mathematical integers as there will be an infinite set of integers that are too large to fit into our finite memory.

2.3 String rewriting

2.4 Term rewriting

Programs often contain *expressions* such as

$$17/(4 + (x/2))$$

They have a well-defined syntax: for instance $4) * (x + 2)$ is not a syntactically well formed expression because of the orphaned opening parenthesis.

This particular way of writing expressions follows the style that we learn in school which makes use of *infix operators* like + and / to represent the operations of addition and division; they are called infix because are written in between the things they operate on. Expressions can nest and we understand that evaluation of an expression proceeds from the innermost bracket: to compute $17/(4 + (x/2))$ we first need to divide the value of x by 2, then add 4, and then divide the result into 17.

The choice of infix notation is just that: an arbitrary choice, and we could have decided to use a different syntax to specify the same sequence of operations, such as

```
divide(17, add(4, divide(x, 2)))
```

We call this form a *prefix* syntax because each operation is written in front of the (parenthesized) list of arguments that it is to operate on.

Yet another form, often called *Reverse Polish Notation* enumerates the arguments and then specifies the operation:

```
x,2,divide,4,add,17,divide
```

This format has the advantage that the operations are encountered in the order in which they are to be executed, and so no parentheses are required. That is a significant advantage, but many of us who grew up with infix notation find these sorts of expression hard to read.

All three of these forms are formally equivalent in that we can unambiguously convert between then without losing any information, and in fact it is easy to write a computer program to perform that conversion.

Although infix notation is familiar from everyday use it does not extend very comfortably to operations with more than two arguments. As a rare example: Java and C both provide the `p ? et : ef` notation for an expression in which predicate `p` is evaluated and then either expression `et` or expression `ef` is evaluated depending on whether the result of `p` was true or false.

In practice most programming languages provide infix notation for commonly understood operations such as addition, less than and logical-AND, but use prefix notation for other operations. Usually we can define procedures which are then called using a prefix notation. So, for instance, in Java we might write

```
System.out.println(Math.max(x,y))
```

.

If you are interested in the design of external language syntax then there are some alternatives to this approach that you might like to investigate. For instance Scheme and other LISP-like language use an exclusively prefix style; the printer control language PostScript uses Reverse Polish Notation; the Smalltalk language effectively uses an infix notation to activate all methods; the C++ language allows the dyadic operator symbols like + to have their meanings extended to include new datatypes, and the Algol-68 language allowed completely new dyadic operator symbols to be defined. We shall return to these matters of syntactic style in Chapter 6.

2.5 Internal syntax style

As language *implementors* and specifiers, we are mostly concerned with *internal* syntax — that is, how to represent programs compactly within the computer. We would like a general notation which is quite regular and thus does not require us to switch between different styles of writing what are essentially similar things. We should like to be able to easily transform programs so that if we chose, we could rewrite an expression such as $3+(5-(10/2))$ into $3+(5-5)$ or even 3.

The *prefix* style is both familiar from mathematics and programming, and easy to manipulate inside the computer so we shall use that style almost exclusively to describe entire programs, and not just expressions. For instance the program

```
x = 2;
while (x < 5) { y = y * y; x++;}
```

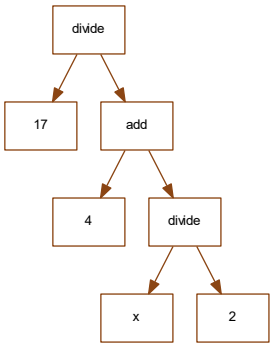
might be written

```
sequence(assign(x,2),
         while(greaterThan(x,5),
               sequence(assign(y, mul(y, y)),
                        assign(x, add(x, 1)))))
```

Here, the concatenation of two statements X and Y in Java is represented by `sequence(X, Y)` and an assignment such as `x = 2;` by `assign(x,2)`.

This notation has the great merit of uniformity: the wide variety of syntactic styles which are used in high level languages to improve program readability for humans is replaced by a single notation that requires us to firstly specify what we are going to do, say `add` and then give a comma-delimited parenthesised list of arguments that we are going to operate on.

The heavily nested parentheses can make this a rather hard-to-read notation although careful use of indentation is helpful. Sometimes, for small expressions at least, it can be helpful to use a tree diagram to see the expression. For instance $17/(4+(x/2))$, which we would write `divide(17,add(4,divide(x,2)))` can be drawn as



2.6 Terms

We call the components of a prefix expression *terms*. Syntactically, we can define terms using an inductive (recursive) set of rules like this.

1. A symbol such $\boxed{1}$, $\boxed{\pi}$ or $\boxed{:=}$ is a term.
2. A symbol followed by a parenthesized comma-delimited list of terms is a term.

Rule one defines terms made up of single symbols. Rule 2 is recursive, and this allows us to construct terms of arbitrary depth by building one upon another.

The *arity* of a term is the number of terms within its parentheses. Terms from rule 1 have no parentheses: they are arity-zero. Equivalently, the arity is the number of children a term symbol has in its tree representation. Rule 1 terms have no children and so are the leaves of a term tree.

Quite often, all instances of a symbol will have the same arity. For instance, addition is usually thought of as a binary (arity-two) operation, and an expression $3 + 4 + 5$ could be represented by the term `add(add(3, 4),5)`. However, we could instead decide to have *variable* arity addition, in which case $4 + 4 + 5$ could be represented as `add(3,4,5)`.

2.6.1 Denoting term symbols

We are very permissive about what constitutes a symbol. When we are thinking about theory, we allow the symbols to be any mathematical object. In this book, when we are thinking about computer based tools we shall allow a symbol to be *any* valid text string over the Unicode alphabet.

Now, great care is needed when reasoning about and writing down terms. Rule 2 above make comma and parentheses special: how would we go about writing a symbol that contained parentheses or command? We call these special characters *metacharacters* because they are used in the denotation of terms.

If we do want a parenthesis or a comma within a symbol, we usually write it with a preceding back-slash (`\( \) \,` or sometimes back-quote character. Of course, we have now added another meta-symbol, so if we want a back-slash in a symbol name we have to write it as `\\`.

2.6.2 Typed terms

Our definition of terms allows any term to be a subterm of any other term. Often we want to place constraints on our terms by limiting

2.7 Terms and their implementation in Java

Assume that we have types *Str*(strings), *Nat*(natural numbers) and *Obj*(any data type).

We can move from

Pure text labels with embedded arity *Str* label $\times$ *Nat* arity $\times$ term*

String map *Str* $\leftrightarrow$ *Nat* *Nat* label $\times$ *Nat* arity $\times$ term*

Fixed arity map *Nat* $\leftrightarrow$ *Nat* *Nat* label $\times$ *Nat* arity $\times$ term* $\vee$ *Nat* label $\times$ term*, label $\in$ arity map

Types mapped to -1 in arity table, structure map *Nat* $\leftrightarrow$ *Obj* *Nat* label$\times$ *Nat* data

Small types bool, char, int and real mapped to negatives *Nat* label $\times$ data

Arrays should just be a vector of children

3 Structural Operational Semantics

If we want to reason about systems which ultimately execute via an implementation language then we have to reason about the behaviour and correctness of that implementation language. If we want to be able to make provable statements about the correctness and completeness of the languages that we develop, then we must rely the formal correctness of the implementation language and of its own implementation. But how are we to establish formally the correctness of the implementation languages? We have a rather circular problem here.

When we use mathematics we typically try to establish relations and equations between mathematical objects, and then use substitution to propagate those relationships to derived objects. We should like to use those techniques to establish properties of the languages that we are developing, without having to rely on the semantics of some pre-existing programming language. If we can explain the execution of programs using only simple mathematical notions and symbol pushing games, then we have a way of talking about programs that is independent of their implementation on a real computer, that is a *formal semantics*. Even better, if we can find a way to *operationalise* the mathematical description of semantics, then we can in principle automatically generate interpreters from the formal semantics. If the automated construction process is sound, then our generated interpreters will faithfully meet their specification.

Now, in engineering terms formal specifications are still only as good as the person who wrote them: we can still write rubbish and so formal specification does not cure all ills. However, automatic generation certainly reduces the error rate, and the availability of very compact specifications allows us to share our design easily with other experts who can immediately see what we are doing, and can help refine our specification. The ideal situation would be for us to also be able to re-use parts of existing specifications, just as we re-use code in conventional software engineering by accessing libraries and API's.

3.1 The basic idea

The core idea is that we shall model program execution taking the tree form of the program and successively rewriting it into a new tree until no further rewrites are possible, at which point execution halts. We shall specify the kinds of rewrites that may be applied using a set of *rules* and an interpreter which will apply those rules to the tree: the set of rules together make up the *formal semantics* of the language.

For a pure functional language, the rewrites alone will completely model the

language. Very few languages are purely functional though: real programs have *side effects* which may be as simple as outputting a sequence of characters or may involve complex manipulation of the contents of memory *via assignments*. Our rules will therefore allow us to specify side-effects that accumulate as the tree is repeatedly rewritten.

An important simplification is that the label of the root of a tree to be rewritten will be used to select which rule to use. If we did not make this restriction, then we would have to search all over a (potentially huge) tree to find putative rewrites, and for languages with side effects we would also need some mechanism for specifying the rewriting order. Our rule, then, will simply be that we must rewrite using a rule for the root node label, and if there is no such appropriate rule then execution will cease even if there are other possible rewrites elsewhere in the tree. This has the twin effects of establishing a rewrite order, and improving efficiency since we do not have to hunt around for rule matches.

We should be careful here though. Although only the root node can be used to select rules, there might still be multiple rules that can be activated, and we then need to consider how to process such specifications. One approach is to exploit this property in modelling *concurrency* and, where multiple rules are active, proceed with all of them at once, each effectively creating a separate thread of control. Alternatively, we may have some mechanism for prioritising the rules, say by checking them in the order that they appear in the specification and taking the first one.

We should also note that, although the root-node-first approach is much more efficient than the allowing rewrites anywhere, it is still a rather slow way to run a program. Depending on your application, it may be fast enough. In later chapters we shall consider program execution via another technique called an *attribute grammar* which can provide good performance, but which is less tractable when we want to reason about properties of the programming language or of programs themselves. For ultimate performance, both the structural operational semantics and attribute grammar formalisms may be used to output *compiled* machine code (or its assembly language equivalent) which is then executed in the conventional way by a real computer, and we shall examine approaches to compilation later in this book.

3.2 Execution via substitution

We now need to formalise our approach into a *game* which can run as an automaton. We shall use ideas from mathematics – relations, equations and substitution – to describe a running computer program.

In Chapter 1.5 we have already seen (in an informal setting) that we can describe program execution as a set of states with transitions between them representing the execution steps of a program. More formally, we developed the idea of a program execution trace as a series of steps that walk a *transition relation* over *configurations*. A configuration represents the state of a computer, and configuration Γ_1 is related to Γ_2 if and only if Γ_2 can appear immediately

after Γ_1 in *some* execution of *some* program. Configurations always contain a program term, and in addition we add entities that represent whatever side effects of program execution we need to record.

By 'some execution of some program' we mean any valid program step that you can imagine - it does not need to be useful or sensible, it just needs to be allowed by the language that we are writing a formal semantics for.

3.2.1 Configurations

When modelling a programming language, we begin by deciding on the configurations of that language. A configuration is a tuple of terms comprising at least a program term θ, and possibly including a store term σ which represents the values of program variables, an environment ρ (which holds information about the scope and location of program objects), an output stream α, an input stream β and some signals ν which RE used to model exception handling. In the first part of this chapter we shall use configurations of the limited form $\langle \theta, \alpha \rangle$ comprising a program term θ and an output term α.

Execution starts with θ being equal to the whole program to be executed. We then pick one small part of it, such as the addition of two constant integers, that we could directly execute, and then rewrite the program to some new term θ_1, replacing the addition with its result.

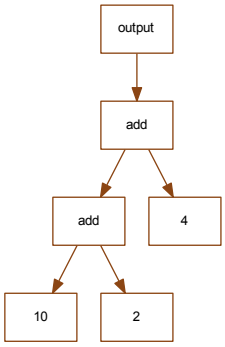
For example, this program fragment when interpreted will eventually result in the value 16 being output.

```
1 output(10+2+4)
```

Conventional language compilers would typically convert this into machine-level instructions that add together the 10 and the 2, then add in the 4, and finally output the result. So as to avoid discussing the complexities of infix operators with their priorities and associativities, let us represent the program as a tree, or equivalently as a parenthesized term thus:

```
1 output(add(add(10, 2),4))
```

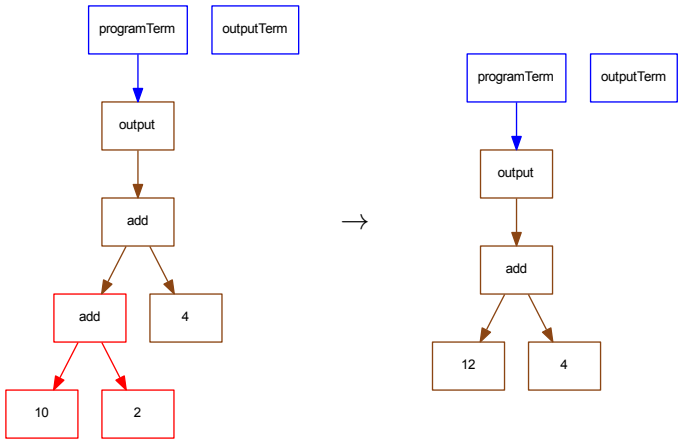
This sort of prefix functional term corresponds directly to a tree if we think of the written term as being the textual trace of a pre-order tree traversal, with parentheses being added to show when we pass down or up a tree edge:



The first computation step for this program reduces the expression to this simpler term

```
output(add(12, 4))
```

which we can show graphically as



As in Chapter 1.5, we have highlighted the program part that is about to be rewritten in red. We also have blue nodes representing the various state components of a running program: in this case limited to the program term and a presently empty term intended to represent the list of outputs made by a program.

The full execution of the program is a sequence of three such steps, which we represent as tuples of the program term and the output.

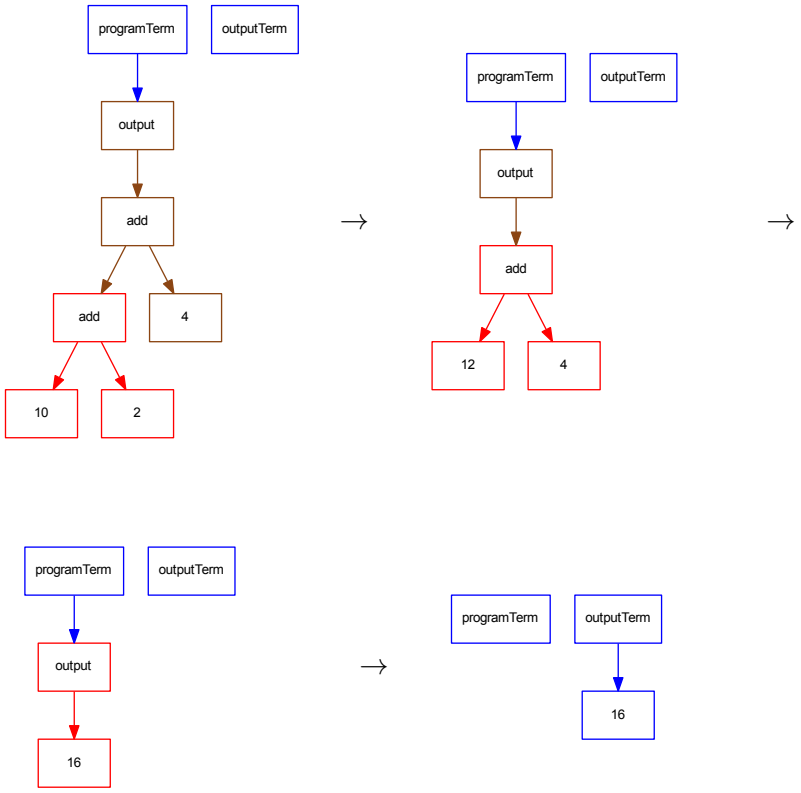
⟨output(add(add(10, 2),4)), [ ]⟩

⟨output(add(12,4)), [ ]⟩

⟨output(16), [ ]⟩

⟨, [16]⟩

As before, the subterm to be evaluated is in red, and we record any side effects of the computation in an appropriate semantic entity term. In this case, we have an output term which receives the element 16 as the output statement is reduced. Here is the graphical representation of this sequence of three transitions.



3.3 Avoiding empty terms – the special value done

It turns out to be uncomfortable to have 'empty' terms. For instance, the output statement could be just the first component of longer program such as

```
output(10+2+4);output(6)
```

Here, the ; symbol is a *sequencing* operator: a sequence action requires first the left hand side and then the right hand side to be evaluated. In prefix form, we might represent this as

```
seq(
    output(add(add(10, 2),4)),
    output(6)
)
```

and by using nested seq() instances we can construct sequences of arbitrary length.

Reducing the first output() statement to nothing would leave us with the curious term

```
seq(
    ,
    output(6)
)
```

and to proceed further we should need some notation for representing these sorts of missing or empty terms. To avoid this, we instead invent a special value done which represents the completion of a command.

```
seq(
    done,
    output(6)
)
```

3.4 Term variables are metavariables

A *term variable* is a name which stands for an arbitrary term (tree): it a sort of metavariable (as opposed to the program variables which are represented by elements of a program term). We shall write term variables in *italics* to distinguish them from actual term elements which we have been writing in sans-serif.

The idea of a term variable is to allow us to speak generally about expressions with arbitrary subexpressions. For example, here is a rule describing how sequences containing the special value done may be rewritten.

> *A term describing the sequence of* done *and then any other sub-program may be rewritten to just that sub-program. So, for instance, we can rewrite*
>
> ```
> seq(done, output(6))
> ```
>
> *as*
>
> ```
> output(6)
> ```

This rather clumsy piece of English and its example can be more concisely, precisely and generally be expressed as follows.

If X is a term variable, then we can then say that a term of the form seq(done, X) can be rewritten as simply X where X is any valid term, or just

$\mathsf{seq}(\mathsf{done}, X) \rightarrow X$

We shall make extensive use of term variables as placeholders within trees.

3.5 Pattern matching of terms

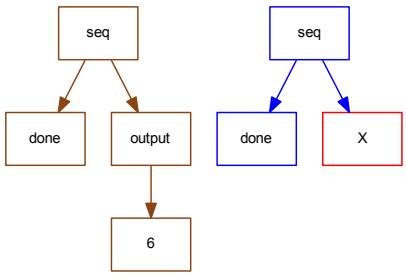
Since we are trying to build a formal game which will execute a program without human intervention, we need some syntax in which to write the rules of the game. As we have already seen, for this reduction of a sequence with done as its left hand argument we write

$$\mathsf{seq}(\mathsf{done}, X) \rightarrow X$$

This kind of rule is an unconditional rule: anywhere that we find a term that matches the *pattern* $\mathsf{seq}(\mathsf{done}, X)$ we can directly replace it with whatever the term variable X stands for.

A pattern is a term which may contain term variables. A term which has no term variables in it is called a *closed* term. *Pattern matching* is the process of comparing a closed term to a pattern to decide if they match and if they do, constructing a table of term variables showing what they represent. The relationship between a term variable and its corresponding subterm is called a *binding*; a set of bindings is called an *environment*.

We can represent this process visually as follows:



The sepia coloured closed term is to be matched against the blue pattern. Some of the leaves of the pattern term may be coloured red: these are nodes labeled with term variables. We perform the matching by recursively traversing both trees in tandem. If we arrive at a node which is blue in the pattern but for which the label does not match the label of the corresponding sepia node, then the pattern match fails. If we arrive at a node which is red in the pattern then we have found a term variable-labeled pattern node: we create a binding between that term variable and the corresponding sepia node (which of course represents the entire subtree rooted at that node). Otherwise we descend into the children and continue the recursive traversal.

We can encode this using a recursive function which takes an environment of bindings, a node from the closed term and a node from the pattern as follows:

```
1 match(M: set of term variables, E: environment, t: term, p: term)
2         returns environment OR bottom
3   if label(p) in M then add p |-> t to E
4   else if label(p) != label(t) then return bottom
5   else for ct in children(t), cp in children(p) do add match(E, ct, cp) to E
6   return E
```

Note that our unusual syntax for p in q, r in s do stands for sequential pairwise traversal of the two lists t and p. We initiate a pattern match by calling match(M, {}, t, p) where M is the set of term variables in the pattern, t and p are the root nodes of the term and pattern trees respectively and {} is an empty environment.

A pattern term may be arbitrarily deep, but in the version of pattern matching that we shall use term variables will always be the labels of leaf nodes. We shall also restrict ourselves to matching patterns against closed terms. It is easy to imagine more baroque pattern matching operations, but this will be sufficient for our style of semantics specification.

A further important restriction is that a term variable X may only appear at most once within a pattern. Again, one could imagine a version of pattern matching in which the appearance of two instances of a term variable X meant that they must each match the same subtree, but we shall not allow this.

We shall write

$$\theta \triangleright \pi$$

for the operation of matching closed term θ against pattern π. The result of such a pattern match is either *failure* represented by $\bot$, or a set of bindings. So

$$\mathsf{seq}(\mathsf{done}, \mathsf{output}(6)) \triangleright \mathsf{seq}(\mathsf{done}, X)$$

returns

$$\{X \mapsto \mathsf{output}(6)\}$$

and

$$\mathsf{seq}(\mathsf{done}, \mathsf{output}(6)) \triangleright \mathsf{seq}(\mathsf{done}, \mathsf{output}(Y))$$

returns

$$\{Y \mapsto 6\}$$

whereas

$$\mathsf{seq}(\mathsf{done}, \mathsf{output}(6)) \triangleright \mathsf{seq}(X, \mathsf{done})$$

returns

$$\bot$$

because output(6) does not match done.

An important special case of pattern matching uses a pattern which is itself a closed term: in such a case, the pattern matcher will return an empty environment if the two terms are identical, or $\bot$ if they differ.

3.6 Pattern substitution

Pattern matching is a way to extract subtrees (subterms) from within closed terms. The bindings will associate term variable names with these subterms, which will themselves be closed (i.e. they will not contain nodes labeled with term variables).

Pattern substitution is the process by which we stitch subterms into a pattern to create a new closed term by substituting the bound subterms for term variables in the pattern

We shall write

$$\pi \triangleleft \rho$$

for the operation of replacing term variables in pattern π with their bound terms from the environment ρ. The result of such a substitution is a closed term; it is an error for π to contain a term variable that is not bound in ρ. So

$$\mathsf{plus(X,\ 10)} \triangleleft \{X \mapsto 6\}$$

returns

$$\mathsf{plus(6,\ 10)}$$

Here is a recursive function to perform substitution

```
1 substitute(M: set of term variables, E: environment, t: term) returns term
2   if label(t) in M then return E.get(label(t)).deepCopy()
3   else {
4     ret = t.shallowCopy()
5     for ct in children(t)
6       t.addChild(substitute(M, E, ct)
7     return ret
8   }
```

3.7 Rules and rule schemas

We now have most of the machinery we need to construct our formal semantics game; we know how to decompose and compose terms (trees) to give the effects we showed in Chapter 1.5. There is one major gap though, and that is the *selection* of sub-phrases to rewrite. Of course, the ordering of these selections is important: for instance we know that $(x - y) - z$ is not in general the same as $x - (y - z)$ so the order in which we formally evaluate the subtractions will affect the final result.

For very simple languages, it might be practical to define their semantics by enumeration. Consider a language which allows a single expression to be output, and limits that expression to a single addition over numbers in the range 0–2. using configurations $\langle \theta, \alpha \rangle$ there are only nine possible programs

that can be written in this language each of which we could evaluate directly using these nine rules:

$$\langle \mathsf{output}(\mathsf{plus}(0,\ 0)), [\ ] \rangle \rightarrow \langle \mathsf{done}, [0] \rangle$$

$$\langle \mathsf{output}(\mathsf{plus}(0,\ 1)), [\ ] \rangle \rightarrow \langle \mathsf{done}, [1] \rangle$$

$$\langle \mathsf{output}(\mathsf{plus}(0,\ 2)), [\ ] \rangle \rightarrow \langle \mathsf{done}, [2] \rangle$$

$$\langle \mathsf{output}(\mathsf{plus}(1,\ 0)), [\ ] \rangle \rightarrow \langle \mathsf{done}, [1] \rangle$$

$$\langle \mathsf{output}(\mathsf{plus}(1,\ 1)), [\ ] \rangle \rightarrow \langle \mathsf{done}, [2] \rangle$$

$$\langle \mathsf{output}(\mathsf{plus}(1,\ 2)), [\ ] \rangle \rightarrow \langle \mathsf{done}, [3] \rangle$$

$$\langle \mathsf{output}(\mathsf{plus}(2,\ 0)), [\ ] \rangle \rightarrow \langle \mathsf{done}, [2] \rangle$$

$$\langle \mathsf{output}(\mathsf{plus}(2,\ 1)), [\ ] \rangle \rightarrow \langle \mathsf{done}, [3] \rangle$$

$$\langle \mathsf{output}(\mathsf{plus}(2,\ 2)), [\ ] \rangle \rightarrow \langle \mathsf{done}, [4] \rangle$$

This is clearly not a very practical approach. What we need to do is to be able to express the pattern of additions more concisely. We call rules that have term variables in them *rule schemas* because they are really a compact way of generating a (possibly infinite) set of real rules.

In pseudo code, we might say something like this:

```
let x, y and z be term variables
if program term theta matches output(add(x,y))) with some alpha and
    x is bound to an integer in the range 0—2 and
    y is bound to an integer in the range 0—2 and
    z is bound to the result of adding x and y together then
    rewrite theta to done and alpha to the substitution of z
```

More formally, using the notations we have developed we might say

$$\begin{aligned} \text{if} \quad & \langle \rho_1 = (\theta \triangleright \mathsf{output}(\mathsf{plusOp}(X, Y))), \alpha \rangle \\ & \text{and} \quad \texttt{is012}(X) \triangleright \mathit{true} \\ & \text{and} \quad \texttt{is012}(Y) \triangleright \mathit{true} \\ \text{and} \quad & \rho_2 = ((\texttt{addOp}(X, Y) \triangleleft \rho_1) \triangleright Z) \\ \text{then} \quad & \theta \rightarrow \theta' = \langle \mathsf{done}, [Z \triangleleft \rho_2] \rangle \end{aligned}$$

We have introduced a new mechanism here: simple functions that take terms and return terms which we write in `teletype font`. We can think of these as pre-existing mathematical functions whose definition is obvious, or if we are writing an interpreter then we might think of these as lookup tables, or calls to very small programs that compute results. The important thing is that these functions must be so small as to allow us to trivially check their correctness.

In this case we are using two new functions: `is012(x)` which returns a term *true* or a term *false* depending on whether `x` is in the set $\{0, 1, 2\}$ or not;

and `addOp(x, y)` which returns a term labeled with the number formed from the addition of terms `x` and `y`. Notice how everything we are doing reduces to operations over terms: our functions are not returning values such as *true* or *false*; they are returning trees made up of a single node *labeled* with *true* or *false*. Notice also that we are using names such as `addOp` for the function that computes the operation of addition, as opposed to the term constructer `add` which is a tree label from a program term. You should think of `add` as the piece of syntax that requests an addition, and `addOp` as the name of the function (or perhaps even machine instruction) which will actually perform the addition. We shall follow this convention throughout: names with the `Op` suffix are used for functions that perform computations, and they must not also appear as the names of a term (tree) element.

Even our formal version of the rule schema is rather a lot of writing. It will turn out that usually several of these rule schemas will be used together in a way that corresponds to inference in a logical system, and we use a special form of syntax that allows us to show derivations in that logical system as trees of rule schemas. A general inference rule looks like this:

$$\frac{C_1 \quad C_2 \quad \ldots \quad C_n}{\langle \theta, \alpha \rangle \rightarrow \langle \theta', \alpha' \rangle}$$

The elements above the line (the C_i) are called *conditions*. Conditions can themselves be transitions although we have not yet encountered examples of that style. Conditions may also be simple matches against the return value of a function, in which case they are called *side conditions*. The single transition below the line is called the *conclusion*. You might read an inference rule in this style as:

> if you have a configuration $\langle \theta, \alpha \rangle$,
> and C_1 succeeds and C_2 succeeds and ... and C_n succeeds
> then transition to configuration $\langle \theta', \alpha' \rangle$

so one reads this kind of rule by checking that the current configuration matches the left hand side of the conclusion, then by checking the conditions, and if everything succeeds rewriting the current configuration into the right hand side of the conclusion. We sometimes refer to this rather operational view of logical inference as 'reading round the clock'.

The inference rule representation of our schema is

$$\frac{(\texttt{is012}(X) \triangleleft \rho_1) \triangleright \mathit{true} \quad (\texttt{is012}(Y) \triangleleft \rho_1) \triangleright \mathit{true} \quad \rho_2 = ((\texttt{addOp}(X,Y) \triangleleft \rho_1) \triangleright Z)}{\langle \rho_1 = (\theta \triangleright \mathsf{output}(\mathsf{plusOp}(X,Y)), \alpha \triangleright [\,]\rangle \rightarrow \langle \mathsf{done}, [\alpha, Z \triangleleft \rho_2]\rangle}$$

We have been careful here to represent all of the pattern matching and substitution operations explicitly. In practice, it is understood that (i) each rule has its own set of term variables even if the same term variable name is used in multiple rules (that is, there is no communication of bindings from one rule to the next) and that (ii) as a rule is checked, a private environment is developed as we go round the clock, (iii) the first time we meet a term variable it is being used to create a binding, (iv) subsequent appearances of a term

variable are to be substituted by its binding and (v) that the term in the left hand side of the conclusion is a pattern to be matched against θ in the current configuration.

This allows us to abbreviate our inference rule to:

$$\frac{\texttt{is012}(X) \triangleright \mathit{true} \quad \texttt{is012}(Y) \triangleright \mathit{true} \quad \texttt{addOp}(X,Y) \triangleright Z}{\langle \mathsf{output}(\mathsf{plusOp}(X,Y)), \alpha\rangle \rightarrow \langle \mathsf{done}, [\alpha, Z]\rangle}$$

and this is the style that we shall use in future.

3.8 The interpreting function F_{SOS}

Now that we have pattern matching and substitution operations along with notions of transitions and side conditions, we can think about a function which takes an input term and *interprets* it by looking through the rules for possible transitions.

3.8.1 Managing the local environment

When implementing F_{SOS} using a procedural language with assignment, there is a useful optimisation that we can apply. Recall that when we wrote out the full version of the inference rule we were careful to create new environments $\rho_1, \rho_2, \ldots$ each time we performed a pattern match. As we moved from the detailed version of a rule schema to the abbreviate form we noted that:

> (iii) the first time we meet a term variable it is being used to create a binding
>
> and
>
> (iv) subsequent appearances of a term variable are to be substituted by its binding

As a result term variables will never be reused (that is a binding cannot subsequently be changed) and we can use a single environment to which bindings are added as we go round the clock. We shall call this mutable environment E.

3.8.2 Procedural pseudo-code for F_{SOS}

This is a procedural implementation of the rule application function Fsos. The function takes a configuration made up of a program term and zero or more semantic entities and either returns a new configuration or $\perp$. It accesses a set of rule schemes R each of which has a conclusion and a set of conditions. In the pseudo code, we use the operators |> and <| for the pattern matching $\triangleright$ and substitution $\triangleleft$ operations.

```
1 let R be the set of rule schemas
2
3 Fsos(C: configuration) returns configuration OR bottom
```

```
4   for r in R
5     if C |> r.conclusion.lhs then
6       let E be an empty set of bindings
7       for c in R.conditions
8         if isSideCondition(c)
9           let res be (c.lhs <| E) |> c.rhs
10          if res = bottom then next r else add res to E
11        else
12          let T be c.lhs <| E
13          if isvalue(T) then return T
14          let res be Fsos(T) |> c.rhs
15          if res = bottom then next r else add res to E
16      return r.conclusion.rhs <| E
17  return bottom
```

The basic approach is just as we described in our 'round-the-clock' informal description of how to read an inference rule.

We start with a configuration, perhaps the initial program and an empty output list. We then scan through all of the rule schemas until we find one that matches the root node of our term. (As an aside, we can make this process more efficient by storing R as a map from constructor label L to subsets of R that have L as the root constructor of their conclusion's left hand side; we have not used this optimisation here.)

We then create a new, empty environment called E and work our way across the conditions evaluating them; if they succeed we add their bindings into E, but if any fail we abort the processing for this rule and throw away E, seeking another rule whose conclusion left hand side matches our term.

Conditions can be either side conditions or transitions.

- For a side condition we call the left hand side function and then pattern match the result against the condition's right hand side.
- If the condition is a transition, we first let T be the condition's left hand side after substitution; if T is a value we return that, otherwise we recursively call $F_{SOS}(T)$ and pattern match the result against the condition's right hand side.

There is an important technical detail here: a call to Fsos() may result in $\bot$ but in line 12 we pattern match the result of the call. Now, the pattern match operator $\triangleright$ is usually only defined over terms, but here we extend the definition so that an attempt to pattern match against $\bot$ will yield $\bot$, and in that way the failure propagate up.

If all of the conditions succeed, then at line 14 F_{SOS} returns the right hand side of the conclusion after substitution against the final value of E.

If we exhaust the rules set R then we have arrived at a terminal configuration, that is one from which no further transitions may be made. For a correct program, this terminal transition will correspond to the program's final version. If the rules were ill-formed, it would be possible to run out of applicable

transitions prematurely: such a configuration is called a *stuck configuration* and would be reported as an error by the interpreter which requires the rules to be changed. In detail, we nominate some program terms as *values*. If an evaluation terminates with a value term, then we have a normal execution. If an evaluation terminate with a non-value term, then we have a stuck execution. The *done* constructor in our rule is an example of a value: it marks normal termination of commands. Numeric and other literals such as strings are also usually values; and we may indicate the successful completion of expression evaluations by reducing them to one of these values.

3.8.3 Program term rewrites - the outer interpreter

The Fsos() function only performs a single transition on the program term, so we usually need to wrap the initial call in an interpret() function which repeatedly applies Fsos() until we arrive at a terminal configuration; flagging an error if that configuration is not a value.

```
1  interpret(C: configuration) returns configuration
2    C' = Fsos(C)
3
4    while C' not bottom
5      C = C'
6      C' = Fsos(C)
7
8    if not isValue(C.theta) error('Stuck configuration')
9
10   return C
```

As we shall see below, sometimes we shall write *transitively closed* rules which internally manage the rewriting of the program term so that the whole program is redued to a value by a single call to Fsos(). In that case we would not need to change the interpret() function but the body of the while would only execute once; if we knew that the rules had this property then we could instead just directly call Fsos() once.

3.9 Structural Operational Semantics and F_{SOS} traces

It is clear that in some sense the structure of the term to be executed specifies the execution order. When we looked at attribute grammars we focused on so-called L-attributed specifications which we processed by descending as deeply as possible into the tree and then propagating values back up using synthesized attributes, a sort of inside-out evaluation.

Using a technique due to Plotkin called *Structural Operational Semantics*, we shall specify similar inside-out steps often using repeated term traversals. The basic idea is to conditionally rewrite the term by isolating a subcomputation that can be performed immediately, and we ensure that the rewrites are done

in the inside-out order by building inference rules that have the abstract syntax of our language embedded in the conclusions.

3.9.1 SOS rules for an addition language

As a first example, let us generalise the 0, 1, 2 addition language of the previous section to allow expressions involving an arbitary number of additions over general additions.

We begin with a rule that performs addition for expressions such as $3 + 4$, that is where each operand is a single integer. The rule uses our abstract syntax, which would encode this example as add(3, 4). Here and in future, we shall name rules for reference purposes by giving a unique tag in square brackets at the start of the rule. These names have no meaning in themselves but we often use names that indicate the purpose of the rule. Sometimes we shall just number them.

$$[\text{add}] \qquad \frac{\texttt{isInt}(n_1) \triangleright \mathit{true} \quad \texttt{isInt}(n_2) \triangleright \mathit{true} \quad \texttt{addOp}(n_1, n_2) \triangleright V}{\langle \mathsf{add}(n_1, n_2), \alpha \rangle \rightarrow \langle V, \alpha \rangle}$$

This is very similar to the rule we wrote for our 012 language, except that we have taken away the output operation, and we now allow the term variables n_1 and n_2 to match any integer, not just the integers 0, 1 and 2. We allow this simply by using side conditions to test the term variables with a function `isInt()`.

Next we re-introduce the output statement using a separate rule for the output() constructor which checks that its argument is an integer, and if so transfers it to the output list before rewriting the program term to the value done.

$$[\text{outputInt}] \qquad \frac{\texttt{isInt}(n) \triangleright \mathit{true}}{\langle \mathsf{output}(n), \alpha \rangle \rightarrow \langle \mathit{done}, [\alpha, n] \rangle}$$

Now we have a problem. This rule will successfully interpret programs such as output(3) since 3 is an integer so the side condition will succeed. However, a program like output(add(3, 4)) will become stuck, since add(3, 4) is not an integer: it is an expression that we can imagine being reduced to an integer using rule [add], but there is nothing in rule [outputInt] to say how that is to be done.

One way of fixing this problem would be by creating variants of the [outputInt] rule which handled various expressions. Of course, in any realistic language, there is an infinite set of expressions of ever increasing depth, even when (as here) we are allowing only one operator, and of course we do not want to write out the corresponding infinite set of [output] rules. Just as with syntax specification, where we used inductive definitions formed from recursive rules to generate infinite language, here we shall use recursion to handle the unbounded nature of expressions. Effectively, we shall allow the rule for [outputInt] to *ask* its operand to evaluate itself. Here is a rule that has that effect.

[outputExpr] $$\frac{\langle E, \alpha\rangle \rightarrow \langle E', \alpha\rangle}{\langle \mathsf{output}(E), \alpha\rangle \rightarrow \langle \mathsf{output}(E'), \alpha\rangle}$$

This is the first rule we have seen that has a transition as one of its conditions rather than a simple side-condition. Looking at the Fsos() function pseudo-code, you can see that this rule simply takes any program with output() as its outer constructor, pulls out the argument (which can be of arbitrary complexity) by pattern matching it to term variable E, and then recursively calls Fsos() on the configuration $\langle E, \alpha\rangle$. Whatever comes back from that call is then used to rewrite the program term into a simpler form.

We shall record the behaviour of our interpreters by showing traces of the calls and returns from Fsos(). Consider the program output(add(3, 4)) with an initially empty input list. The program term will be rewritten in two stages: first to the term output(7) and then to the term done. We will represent each in a separate block the first line of which is the rewrite number and the outer call to Fsos() with arguments. Each block thus corresponds to a call to Fsos() that has been made by interpret(). Within each block, we shall show the rule selected by Fsos() for interpretation and, for side conditions the yield, and for transition conditions the trace of recursive calls to Fsos().

```
1. Fsos(output(add(3, 4)), [ ])
  [outputExpr].C1 calls Fsos(add(3, 4), [ ])
    [add].SC1 yields true
    [add].SC2 yields true
    [add] rewrites to 7, [ ]
  [outputExpr] rewrites to output(7), [ ]
```

```
2. Fsos(output(add(7)), [ ])
[outputInt].SC1 yields true
[outputInt] rewrites to done, [7]
```

done is a value, so interpretation terminates.

3.9.2 Expression nesting

The three rules [add], [outputInt] and [outputExpr] together allow us to interpret programs such as output(6) and output(add(6, 7)) but they do not cover programs such as output(add(add(6,7),8) because the only rule we have for the add() constructor requires its operands to be simple integers. This is just another manifestation of the problem we had with the output() constructor, and the resolution follows the same principle.

Now, addition is a left associative operator with two arguments. We should like to evaluate the arguments one at a time, with the leftmost argument being done first. Here are two additional rules for add() which have that effect.

$$\text{[addLeft]} \qquad \frac{\langle E_1, \alpha\rangle \rightarrow \langle I_1, \alpha\rangle}{\langle \mathsf{add}(E_1, E_2), \alpha\rangle \rightarrow \langle \mathsf{add}(I_1, E_2), \alpha\rangle}$$

$$\text{[addRight]} \qquad \frac{\langle E_2, \alpha\rangle \rightarrow \langle I_2, \alpha\rangle \quad \texttt{isInt}(n) \triangleright \mathit{true}}{\langle \mathsf{add}(n, E_2), \alpha\rangle \rightarrow \langle \mathsf{add}(n, I_2), \alpha\rangle}$$

Rule [addLeft] rewrites the left argument to an add() constructor to a simpler expression whilst preserving the second argument. Rule [addRight] will only process terms that had a single integer as the left hand argument, and rewrites the second argument. These two rules together with the [add] rule allow arbitrarilly deep expressions over add() to be evaluated.

Here is an interpretation trace for the program output(add(add(6,7),8)).

```
1. Fsos(output(add(add(6, 7), 8), [ ])
  [outputExpr].C1 calls Fsos(add(add(6, 7), 8), [ ])
    [add].SC1 yields false: backtrack, and seek another rule
    [addLeft].C1 calls Fsos(add(6, 7), [ ])
      [add].SC1 yields true
      [add].SC2 yields true
      [add] rewrites to 13, [ ]
    [addLeft] rewrites to add(13, 8}
  [outputExpr] rewrites to output(add(13,8), [ ]
```

```
2. Fsos(output(add(13, 8)), [ ])
  [outputExpr].C1 calls Fsos(add(13, 8), [ ])
    [add].SC1 yields true
    [add].SC2 yields true
    [add] rewrites to 21, [ ]
  [outputExpr] rewrites to output(21), [ ]
```

```
3. Fsos(output(add(21)), [ ])
[outputInt].SC1 yields true
[outputInt] rewrites to done, [21]
```

Notice that we only use rules [outExpr], [outInt], [addLeft] and [addInt]. We never need to invoke [addRight] because the right hand argument of the outer add() is already a simple integer. You can probably see that there is a rekationship between the number of add() constructors in the original term and the number of program term rewrites that will be performed during interpretation. One of the most useful aspects of this style of semantics specification is that it allows proofs of these kinds of properties. In the next chapter we shall see how the proof technique of *structural induction* on our inference rules may be used to formally prove properties of languages.

3.10 An SOS for a language with flow control, variables and expressions

We have now seen the basic interpretive mechanisms at work so we are ready to look at a non-trivial language. In this section we shall develop a formal semantics for a language powerful enough to implement Euclid's algorithm, and show how the transitions in Chapter 1.5 may be generated from our style of interpreter.

3.10.1 Configurations

As before, we begin by setting the configuration. Our new language will have variables, so we need a *store* in which to hold their values. Our store will be a simple map from identifiers to integer values, and we shall denote it by σ. We shall not have multiple scope regions, so a single level map suffices and we do not need an environment ρ; in addition our programs will be over the store only, with no output or input. As a result, our configurations will be just $\langle \theta, \sigma \rangle$, that is a program term and a store.

3.10.2 Variable handling

Our store comes with two functions that may be used in side conditions called `get()` and `put()`. These operate just as the equivalent methods do in a Java map, and indeed within the interpreter we implement an instance of σ as a `Map<String,Integer>`, mapping variable identifiers to integers. As usual, our the domain and codomain of our side condition functions is the set of terms, so one must be careful to remember that a value such as the integer 27 is in detail a term comprising a single tree node labeled with the integer 27 rather than some machine-specific representation of 27.

Assignment is handled by using term variables to extract the components of a configuration which has assign() at the top of its program term. We then substitute those term variables into a call to `put()` and bind the resulting new store to a new term variable. The program term is mapped to the tuple of done and the new store.

[assign]
$$\frac{\texttt{isInt}(n) \triangleright \mathit{true} \quad \texttt{put}(\sigma_1, X, n) \triangleright \sigma_2}{\langle \mathsf{assign}(X, n), \sigma_1 \rangle \rightarrow \langle \mathit{done}, \sigma_2 \rangle}$$

Variable access is handled in a way that is rather inefficient for the interpreter — we shall return to this issue in the next chapter. We have a rule which has a *single term variable* for its program term. This means that this rule will be triggered for *any* program configuration; it will usually turn out to be more efficient to arrange things so that this rule is only checked if everything else has failed. In the Java interpreter we shall develop in the next section, we can get this effect by placing the rule at the end of a specification.

[variable]
$$\frac{\texttt{get}(R, \sigma) \triangleright V}{\langle R, \sigma \rangle \rightarrow \langle V, \sigma \rangle}$$

Although the rule can be activated for any program term, we ensure that the side condition function `get()` returns $\perp$ if it is asked to look up a term which is not in the store. In this way, program terms that have not previously been used as a variable name in an assign() constructor will cause the rule to fail.

So far we have only shown how to assign an integer to a variable; we shall need one more rule so as to ensure that expressions more complex than a single integer are suitably evaluated.

[assignCongruence] $$\frac{\langle E, \sigma\rangle \to \langle I, \sigma\rangle}{\langle \mathsf{assign}(X, E), \sigma\rangle \to \langle \mathsf{assign}(X, I), \sigma\rangle}$$

3.10.3 Arithmetic operations

Our rules for addition are essentially the same as before, except that this time our configurations are over $\langle theta, \sigma\rangle$ instead of $\langle theta, \alpha\rangle$.

[addLeft] $$\frac{\langle E_1, \sigma\rangle \to \langle I_1, \sigma\rangle}{\langle \mathsf{add}(E_1, E_2), \sigma\rangle \to \langle \mathsf{add}(I_1, E_2), \sigma\rangle}$$

[addRight] $$\frac{\langle E_2, \sigma\rangle \to \langle I_2, \sigma\rangle \quad \mathtt{isInt}(n) \triangleright \mathit{true}}{\langle \mathsf{add}(n, E_2), \sigma\rangle \to \langle \mathsf{add}(n, I_2), \sigma\rangle}$$

[add] $$\frac{\mathtt{isInt}(n_1) \triangleright \mathit{true} \quad \mathtt{isInt}(n_2) \triangleright \mathit{true} \quad \mathtt{addOp}(n_1, n_2) \triangleright V}{\langle \mathsf{add}(n_1, n_2), \sigma\rangle \to \langle V, \sigma\rangle}$$

We can use the same pattern for any other arithmetic operators we have, and indeed extend the rules to handle floats and other data types as necessary by using different side conditions.

[mulLeft] $$\frac{\langle E_1, \sigma\rangle \to \langle I_1, \sigma\rangle}{\langle \mathsf{mul}(E_1, E_2), \sigma\rangle \to \langle \mathsf{mul}(I_1, E_2), \sigma\rangle}$$

[mulRight] $$\frac{\langle E_2, \sigma\rangle \to \langle I_2, \sigma\rangle \quad \mathtt{isInt}(n) \triangleright \mathit{true}}{\langle \mathsf{mul}(n, E_2), \sigma\rangle \to \langle \mathsf{mul}(n, I_2), \sigma\rangle}$$

[multiply] $$\frac{\mathtt{isInt}(n_1) \triangleright \mathit{true} \quad \mathtt{isInt}(n_2) \triangleright \mathit{true} \quad \mathtt{mulOp}(n_1, n_2) \triangleright V}{\langle \mathsf{mul}(n_1, n_2), \sigma\rangle \to \langle V, \sigma\rangle}$$

It is a little uncomfortable that we need to write three rules for every operator. In the next chapter we shall see alternative styles of rules that allow a more compact description.

3.10.4 Boolean relations

As a variation on the above, here are four rules that define semantics for the greater-than relational operation, represented in the abstract syntax by constructor gt(). Here, instead of using a term variable to carry the result of the side condition that performs the computation, we create a rule for each of the possible outcomes. These rules could instead be written in the style of the arithmetic ones in the previous section.

[gtLeft] $$\frac{\langle E_1, \sigma\rangle \to \langle I_1, \sigma\rangle}{\langle \mathsf{gt}(E_1, E_2), \sigma\rangle \to \langle \mathsf{gt}(I_1, E_2), \sigma\rangle}$$

[gtRight] $$\frac{\langle E_2, \sigma\rangle \to \langle I_2, \sigma\rangle \quad \mathtt{isInt}(n) \triangleright \mathit{true}}{\langle \mathsf{gt}(n, E_2), \sigma\rangle \to \langle \mathsf{gt}(n, I_2), \sigma\rangle}$$

[gtFalse] $$\frac{\mathtt{isInt}(n_1) \triangleright \mathit{true} \quad \mathtt{isInt}(n_2) \triangleright \mathit{true} \quad \mathtt{isgt}(n_1, n_2) \triangleright \mathit{false}}{\langle \mathsf{gt}(E_1, E_2), \sigma\rangle \to \langle \mathit{false}, \sigma\rangle}$$

[gtTrue] $$\frac{\mathtt{isInt}(n_1) \triangleright \mathit{true} \quad \mathtt{isInt}(n_2) \triangleright \mathit{true} \quad \mathtt{isgt}(n_1, n_2) \triangleright \mathit{true}}{\langle \mathsf{gt}(E_1, E_2), \sigma\rangle \to \langle \mathit{true}, \sigma\rangle}$$

3.10.5 Sequential flow control

The simplest for of flow control is the sequencing of statements. The special value done plays an important role here: as we discussed earlier we use it to allow us to express the notion that a finished command followed by some command C has the same meaning as just C on its own:

[done] $$\langle \mathsf{seq}(\mathit{done}, C), \sigma\rangle \to \langle C, \sigma\rangle$$

Just as with arithmetic, we need rules that allow other rules to be activated so as to reduce components of a pattern to basic values: in the case of arithmetic operations, integers and in the case of commands, to done. These kinds of rules are sometimes called congruence rules for reasons that we shall discuss in the next chapter.

For sequencing, we need such a rule to ensure that the left argument is reduced until such time as rule [done] can be activated:

[sequence] $$\frac{\langle C_1, \sigma_1\rangle \to \langle C_1', \sigma_2\rangle}{\langle \mathsf{seq}(C_1, C_2), \sigma_1\rangle \to \langle \mathsf{seq}(C_1', C_2), \sigma_2\rangle}$$

3.10.6 Conditional flow control

An if statement is a language takes an expression as a predicate, and a command to be executed. If the predicate yields true then the command is executed;

otherwise it is skipped. We can express these semantics using two rules that operate on the degenerate predicates true and false. These rules are unconditional; that is they are *axioms*.

[ifTrue] $$\langle \mathsf{if}(\mathit{true}, C), \sigma \rangle \rightarrow \langle C, \sigma \rangle$$

[ifFalse] $$\langle \mathsf{if}(\mathit{false}, C), \sigma \rangle \rightarrow \langle \mathit{done}, \sigma \rangle$$

We also, of course, need to consider the semantics of if statements that have non-degenerate predicates. Again, we use a congruence rule to ensure that general predicates in if statementes are reduced to one of the base cases that we can handle directly.

[ifCongruence] $$\frac{\langle E, \sigma \rangle \rightarrow \langle E', \sigma \rangle}{\langle \mathsf{if}(E, C), \sigma \rangle \rightarrow \langle \mathsf{if}(E', C), \sigma \rangle}$$

3.10.7 Loops

The rules for a while loop are closely related to those for if; in fact [whileFalse] and [whileCongruence] are identical up to the constructor name.

[whileFalse] $$\langle \mathsf{while}(\mathit{false}, C), \sigma \rangle \rightarrow \langle \mathit{done}, \sigma \rangle$$

[whileCongruence] $$\frac{\langle E, \sigma \rangle \rightarrow \langle E', \sigma \rangle}{\langle \mathsf{while}(E, C), \sigma \rangle \rightarrow \langle \mathsf{while}(E', c), \sigma \rangle}$$

The [whileTrue] rule applies the rewrite that we introduced in the first chapter, mapping a while to an if followed by a while.

[whileTrue] $$\langle \mathsf{while}(\mathit{true}, C), \sigma \rangle \rightarrow \langle \mathsf{if}(E, \mathsf{seq}(C, \mathsf{while}(E, C))), \sigma \rangle$$

3.11 Using big steps to simplify the rules

We can avoid the need to explicitly reduce expressions in steps over the left and right arguments by simply specifying that a term can transition directly to its value. We use a big arrow $\Rightarrow$ to indicate such rules. There is a sense in which a whole sequence of congruence rule applications has been built-in to these bug step rules by applying the transitive closure of the rules in our original specification.

Here is a more compact version of the semantics for our language. We lose the link between individual rewrites and machine level operations, in that an entire expression will be rewritten in a single step. We also lose the ability to reason in detail about the effects of exceptions (such as a divide by zero) that might be raised during the evaluation of an expression. However, the specification, and the associated execution traces, become more compact.

$$\frac{\langle E, \sigma\rangle \Rightarrow \langle \mathsf{true}, \sigma\rangle}{\langle \mathsf{if}(E, C), \sigma\rangle \rightarrow \langle C, \sigma\rangle} \tag{3.1}$$

$$\frac{\langle E, \sigma\rangle \Rightarrow \langle \mathsf{false}, \sigma\rangle}{\langle \mathsf{if}(E, C), \sigma\rangle \rightarrow \langle \mathsf{done}, \sigma\rangle} \tag{3.2}$$

$$\frac{\langle E, \sigma\rangle \Rightarrow \langle \mathsf{true}, \sigma\rangle}{\langle \mathsf{while}(E, C), \sigma\rangle \rightarrow \langle \mathsf{if}(E, seq(C, \mathsf{while}(E, C))), \sigma\rangle} \tag{3.3}$$

$$\frac{\langle E, \sigma\rangle \Rightarrow \langle \mathsf{false}, \sigma\rangle}{\langle \mathsf{while}(E, C), \sigma\rangle \rightarrow \langle \mathsf{done}, \sigma\rangle} \tag{3.4}$$

$$\langle \mathsf{seq}(\mathsf{done}, C_2), \sigma\rangle \rightarrow \langle C_2, \sigma\rangle \tag{3.5}$$

$$\frac{\langle C_1, \sigma_1\rangle \rightarrow \langle C_1', \sigma_2\rangle}{\langle \mathsf{seq}(C_1, C_2), \sigma_1\rangle \rightarrow \langle \mathsf{seq}(C_1', C_2), \sigma_2\rangle} \tag{3.6}$$

$$\frac{\langle E_1, \sigma\rangle \Rightarrow \langle I_1, \sigma\rangle \quad \langle E_2, \sigma\rangle \Rightarrow \langle I_2, \sigma\rangle \quad \mathtt{isgt}(I_1, I_2) \triangleright \mathsf{false}}{\langle \mathsf{gt}(E_1, E_2), \sigma\rangle \Rightarrow \langle \mathsf{false}, \sigma\rangle} \tag{3.7}$$

$$\frac{\langle E_1, \sigma\rangle \Rightarrow \langle I_1, \sigma\rangle \quad \langle E_2, \sigma\rangle \Rightarrow \langle I_2, \sigma\rangle \quad \mathtt{isgt}(I_1, I_2) \triangleright \mathsf{true}}{\langle \mathsf{gt}(E_1, E_2), \sigma\rangle \Rightarrow \langle \mathsf{true}, \sigma\rangle} \tag{3.8}$$

$$\frac{\langle E_1, \sigma\rangle \Rightarrow \langle I_1, \sigma\rangle \quad \langle E_2, \sigma\rangle \Rightarrow \langle I_2, \sigma\rangle \quad \mathtt{addOp}(I_1, I_2) \triangleright V}{\langle \mathsf{add}(E_1, E_2), \sigma\rangle \Rightarrow \langle V, \sigma\rangle} \tag{3.9}$$

$$\frac{\langle E_1, \sigma\rangle \Rightarrow \langle I_1, \sigma\rangle \quad \langle E_2, \sigma\rangle \Rightarrow \langle I_2, \sigma\rangle \quad \mathtt{mulOp}(I_1, I_2) \triangleright V}{\langle \mathsf{mul}(E_1, E_2), \sigma\rangle \Rightarrow \langle V, \sigma\rangle} \tag{3.10}$$

$$\frac{\langle E, \sigma_1\rangle \Rightarrow \langle V, \sigma_1\rangle \quad \mathtt{put}(\sigma_1, X, V) \triangleright \sigma_2}{\langle \mathsf{assign}(X, E), \sigma_1\rangle \rightarrow \langle \mathsf{done}, \sigma_2\rangle} \tag{3.11}$$

$$\frac{\mathtt{get}(R, \sigma) \triangleright V}{\langle R, \sigma\rangle \Rightarrow \langle V, \sigma\rangle} \tag{3.12}$$

3.12 Interpretation traces for our language

In this section we give F_{SOS} traces for some examples using the rules we have developed. For convenience, we gather all of the rules together here in the order that F_{SOS} will examine them.

3.12.1 Example 1 – assignment to literal

Initial configuration

$$\langle \text{assign(X, 3)}, \{\} \rangle$$

Trace

```
1. Fsos(assign(X, 3), { })
  [4.11].C1 yields X
  [4.11].SC2 yields { X |–> 3 }
  [4.11] rewrites to done, { X |–> 3 }
```

3.12.2 Example 2 – assignment to variable

Initial configuration

$$\langle \text{assign(Y, X)}, \{X \mapsto 3\} \rangle$$

Trace

```
1. Fsos(assign(Y, X), { X |–> 3 })
  [4.11].C1 calls Fsos(X, { X |–> 3 })
      [4.12].SC1 yields 3
      [4.12] rewrites to 3, { X |–> 3 }
  [4.11].SC2 yields { Y |–> 3, X |–> 3 }
  [4.11] rewrites to done, { Y |–> 3, X |–> 3 }
```

3.12.3 Example 3 – sequence over assignments

Initial configuration

$$\langle \text{seq(assign(X, 3), assign (Y, X))}, \{\} \rangle$$

Trace

```
1. Fsos(seq(assign(X, 3), assign (Y, X)), { })
  [4.6].C1 calls Fsos(assign(X, 3))
     [4.11].C1 yields X
     [4.11].SC2 yields { X |–> 3 }
     [4.1] rewrites to done, { X |–> 3 }
  [4.6] rewrites to seq(done, assign (Y, X)) { X |–> 3 }
```

```
2. Fsos(seq(done, assign (Y, X)), { X |–> 3 })
  [4.5] rewrites to assign (Y, X)), { X |–> 3 }
```

```
3. Fsos(assign(Y, X), { X |–> 3 })
  [4.11].C1 calls Fsos(X, { X |–> 3 })
      [4.12].SC1 yields 3
      [4.13] rewrites to 3, { X |–> 3 }
  [4.11].SC2 yields { Y |–> 3, X |–> 3 }
  [4.11] rewrites to done, { Y |–> 3, X |–> 3 }
```

3.12.4 Example 4 - conditional assignment

Initial configuration — note that the store already includes a value for the variable X.

$$\langle \text{if(gt(X,0)), assign(Y, X))}, \{X \mapsto 2\}\rangle$$

Trace

```
1. Fsos(if(gt(X,0)), assign(Y, X)), { X |–> 2 })
  [4.1].C1 calls Fsos(gt(X,0), { X |–> 2 })
    [4.7].C1 calls Fsos(X, { X |–> 2 } )
      [4.12].SC1 yields 3
      [4.12] rewrites to 3
    [4.7]C2 yields 0 (already a value – no recursive call)
    [4.7]SC1 yields true – failure leading to backtrack
    [4.8].C1 calls Fsos(X, { X |–> 2 } )
      [4.12].SC1 yields 3
      [4.12] rewrites to 3
    [4.8]C2 yields 0 (already a value – no recursive call)
    [4.8]SC1 yields true
    [4.8] rewrites to true, { X |–> 2 }
  [4.1] rewrites to assign(Y, X)), { X |–> 2 }
```

```
2. Fsos(assign(Y, X), { X |–> 3 })
  [4.11].C1 calls Fsos(X, { X |–> 3 })
      [4.12].SC1 yields 3
      [4.12] rewrites to 3, { X |–> 3 }
  [4.11].SC2 yields { Y |–> 3, X |–> 3 }
  [4.11] rewrites to done, { Y |–> 3, X |–> 3 }
```

3.12.5 Example 5 - loops

Initial configuration — note that the store already includes a value for the variable X.

$$\langle \text{while(gt(X,0)), assign(X, -1))}, \{X \mapsto 2\}\rangle$$

Trace

```
1. Fsos(while(gt(X,0)), assign(Y, −1)), { X |−> 2 })
  [4.3].C1 calls Fsos(gt(X,0), { X |−> 2 })
    [4.7].C1 calls Fsos(X, { X |−> 2 } )
      [4.12].SC1 yields 3
      [4.12] rewrites to 3
    [4.7]C2 yields 0 (already a value − no recursive call)
    [4.7]SC1 yields true − failure leading to backtrack
    [4.8].C1 calls Fsos(X, { X |−> 2 } )
      [4.12].SC1 yields 3
      [4.12] rewrites to 3
    [4.8]C2 yields 0 (already a value − no recursive call)
    [4.8]SC1 yields true
    [4.8] rewrites to true, { X |−> 2 }
  [4.3] rewrites to if(gt(X,0), seq(assign(Y, −1), while(gt(X, 0), assign(Y, −1)))), { X |−> 2 }
```

...

4 Syntax

People like to communicate, and language in its broadest sense is the means by which we perform communication. The processes by which concepts in my internal world can be summoned up in another person's consciousness through the use of language is truly mysterious, and yet at the same time so fundamental to our lives that we rarely think deeply about it.

Each of us has some sense of *self*, an internal landscape populated by experiential memories, learned concepts and new ideas. We speak of consciousness as being the embodiment of being aware and alive, but in truth there is very little agreement as to what consciousness *is* even though this fundamental aspect of existence has been been examined and debated for millennia.

On a prosaic level, we communicate by making noises (speech), or perhaps with shapes that we make with our hands (sign language), or by creating signs on paper and other media (writing and symbols). Somehow we achieve agreement that certain noises, shapes or signs stand for certain simple things, and from that common core we can build joint understanding. I like to think that the base case of this inductive process is the human smile, which seems to be universally understood across all cultures as denoting and communicating happiness.

Human languages are extraordinarily diverse. My own native language, English, seems to evolve quite quickly, with each generation inventing its own idioms, jokes and aphorisms: perhaps to better define the new in opposition to the old. So somebody's speech patterns can give me clues as to their age. I can also guess whereabouts somebody grew up from their *accent*: a particular pattern of pronunciation. Clearly human language is not constant but varies from place to place and over time, and this evolution can become so extreme that different sub-communities lose the ability to understand each other directly. Presumably this is how the multitude of current human languages came into being.

Although languages are diverse and evolving, we are able to translate between them. This tells is that in some sense *meaning* is more fundamental than the particular language used to convey that meaning. If *A rose by any other name would smell as sweet* then we can be sure that the word *rose* is just a marker for a deeper experience that exists independently of whether we speak English, French or some long dead language like Babylonian. We should distinguish, then, between language *syntax* and language *semantics*. The syntax is the particular pattern of noises or signs that constitute valid communications, and the semantics is the set of meanings that we can associate with those

communications.

4.1 Syntax in natural languages

When learning a new human language, we might first concentrate on *vocabulary* (the set of generally understood words in that language) and perhaps try to match up words for objects in the new language with the equivalent words in our native language.

Taking French as an example, an English speaker might note that maison in French corresponds to house in English and that voiture corresponds to car. We are not limited to names for objects: attributes such as colour also have closely corresponding words such as rouge for red and noir for black.

In an emergency, and with some good will on both sides, it is possible to communicate simple ideas using just vocabulary, but most communication requires the construction of complete sentences by sequencing together words from the vocabulary. In typical human languages, the order of words within these sequences is significant.

Some sequences are 'wrong' in that a native speaker would not utter them. For instance, in English 'The car black.' sounds very odd: we expect 'The black car.' It seems as though we expect the attributes of an object (colour in this case) to be listed before uttering the word for the object. In French, however, the attributes typically appear after the name of the object: 'la maison rouge' corresponds to the 'the red house' in English. Interestingly, although 'The car black.' is not a valid English sentence, if we were speaking to somebody who was learning English we could probably guess what they meant.

The rules governing valid word orderings are called the *syntax* of the language, and ignoring the rules can cause deep confusion. For instance a word-by-word translation of 'La maison rouge.' yields 'The house red.', which is a valid colloquial phrase since *The house red* in a restaurant means their non-label (and usually cheaper) wine. Simply by putting the colour last we have completely changed the meaning of the sentence.

An important aspect of communication is *redundancy*. If every sequence of noises or sign

Search for universal exact scientific language - latin, arithmetic conventions etc

Syntax as word morphemes, work order, long range relationships such as agreement

4.2 Writing

The origins of spoken and written communication are necessarily obscure, but the archaeological record leaves us some clues. We know that bipedal apes emerged around seven million years ago; that 1.5–2 million years ago early hominids such as *Homo Habilis* used tools to scavenge but were often prey to large animals; and that by the time of *Homo Ergaster* and its descendant species (around 1–1.5 million years ago) there is evidence of the use of fire and

a physiology that might allow some form of speech. By 70–80,000 years ago there is clear evidence that the hunted had turned hunter, with group hunting of large game, which must surely have required coordination, planning and communication.

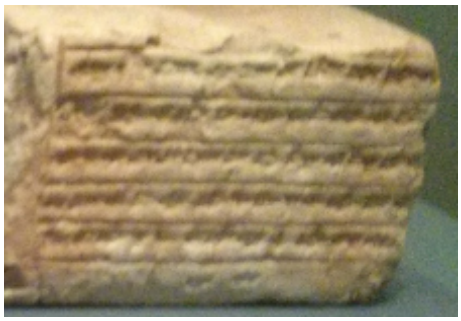
Many hundreds of sites containing cave paintings are known, the oldest of which are believed to be around 40,000 years old and clearly represent some form of *non-transient* communication. Images of animals abound, and images of humans are rare apart from hand stencils (pigment blown over a hand). In a few cases, hunting scenes are clearly represented. This painting, for instance, is from the Bhimbetka rock shelters in India; and shows mesolithic hunters using bows and arrows, perhaps 6–10,000 years ago.

We cannot know whether these are records of successful campaigns, or perhaps instructional in the sense that they played a part in preparing for a hunt and training new members of the group. To be a little whimsical; it would be pleasing if one could establish that some paintings formed a do-this then do-that progression, as this would surely constitute use of a recorded sequence of commands; the world's first program.

The transition to text based systems is also, naturally, obscure. Neolithic objects from Jiahu in China incorporating symbols have been dated to 6,500 BCE. The so-called Vinča symbols on the Tărtăria tablets found in Romania date to around 5,300 BCE (left, below). The Greek Displio tablet (`journals.uair.arizona.edu/index.php/radiocarbon/article/view/17456` is a wooden piece which has been carbon-14 dated to 5260 ± 40 BCE and includes symbols such as a triangle with a dot in it, and forms similar to our letters E, t, v and L (right, below). Scholars in this area call these sorts of symbols, which often appear in isloation as *proto-writing*.

In fully formed writing, strings of symbols (called *graphemes*) represent spoken sentences. Some writing systems are *logographic* in which individual concepts are represented by graphemes called *logograms* (small pictorial symbols representing, say, a class of objects such as house); modern examples include Chinese characters and Japanese Kanji. More commonly individual (or short sequences of) graphemes represent the primitive sounds or *phonemes* of spoken language, forming an *alphabet*. A string of alphabetic graphemes 'spell' out the sound of a spoken word.

Egyptian hieroglyphs combine both elements: of the 1,000 or so known hieroglyphics there are symbols corresponding to consonants as well as symbols that in themselves represent, for instance, sun. Some symbols, such as that for house, may be logographic in some contexts and but stand for a single consonant in other contexts. A vertical bar under the symbol indicates the logographic use. Hieroglyphics and Samarian cuneiform (which was made with cut reeds on clay tablets) are candidates for the earliest fully formed writing system dating back to about 3,000 BCE, but they involve large numbers of signs: cuneiform uses around 800 patterns which represent individual symbols.



Alphabets instead encode the *phonemes*, the individual sounds from which speech is composed which requires far fewer symbols, and in which syllables are represented by short strings of alphabetic letters.

Most Western scripts descend from the Phoenician alphabet which was in use by 1000 BCE, though that script only directly represents consonants (and is thus sometimes called an *abjad* rather than an alphabet). Txt wrttn tht wy cn b prfctl lgbl. Each Phoenecian letter was derived from the shape of the sign for some common syllable which started with that letter. This arrow head from 1100 BCE and now in the British Museum is inscribed *arrow of Ada, son of Bala.*

Extra characters representing vowels were later added by the Greeks, and that developed into the Latin alphabet in which this text was written.

4.3 The search for precision

Betrand Russell - little quote
- Notions and notations - Gauss
- Latin
- Mathematics
- Programming languages

4.4 Metalanguage

Linguistics
- Prefix style is a notation
- BNF
- EBNF
- maths conventions
- tool conventions

Up until now we have used simple parenthesised terms to capture the essential meaning of programs, and then written inference rules that express the semantics in a way that allows us to directly interpret the specification. Now, most real programming languages do not look like these simple terms: perhaps the closest real example would be the Lisp family languages such as Scheme. In practice, we would like to write something in an *outer* syntax like

```
a:=15;
b:=9;
while a != b do
  if a > b then
    a:=a-b else
    b:=b-a;

gcd := a
```

and then we would like to have it automatically translated into something in our *inner* syntax like

```
seq(seq(seq(assign(a, 15), assign(b, 9)),
   while(ne(deref(a), deref(b)),
      if(gt(deref(a), deref(b)),
        fcassign(a, sub(deref(a), deref(b))),
        assign(b, sub(deref(b), deref(a))))))),
assign(gcd, deref(a)))
```

4.5 Outer and inner syntax

In practice, real tools usually maintain internal representations that are optimised for the task in hand, and they might not be tree shaped, or they might not need all of the elements of the derivation tree. In the context of language based

software tools, we often call this internal form the *abstract* or *inner syntax* of a language. The terms *intermediate form*, *internal representation* and *model* also appear in different contexts. The term abstract syntax usually implies a formal (or at least semi-formal) relationship to the syntax of the user language, which is then called the *outer* or *concrete syntax*. Intermediate and internal forms are often understood to be rather *ad hoc*, and it often not easy to show that all possible concrete syntax programs have a valid internal form: forgetting certain cases is a common implementation error. When the internal form is called a *model* we usually understand that the concrete syntax is primarily being used to load the members of a set of classes with data, where the interrelationships between the classes can be specified with a UML diagram. Tools exist that allow an existing UML diagram to be 'decorated' with concrete syntax so as to produce a language tuned to that model. If we need to add new classes, then the language can be regenerated with extended syntax. An alternative approach is to derive the model from the concrete syntax, by annotating the nonterminals and terminals which are 'significant' and must be loaded into the internal form.

In this book we shall use the terms *inner* and *outer* syntax to distinguish the internal computer representation of a program and the external human-centric form. Most often, our inner representations will be trees represented textually as terms.

4.5.1 Syntactic sugar, redundancy and syntactic 'noise'

It is the notion of *significance* which ultimately distinguishes inner from outer syntax. The outer syntax is designed for humans, and often contains elements which protect against common error patterns without adding any semantics. For instance, we could design a concise Java conditional expression which allowed us to write expressions such as

```
1 x = a > b ? y + 2 z * 3
```

The real Java conditional operator requires a colon between the two expressions

```
1 x = a > b ? y + 2 : z * 3
```

Why is this? Well, it allows the concrete syntax analyser to detect the situation where the user mistypes the second expression, omitting the variable

```
1 x = a > b ? y + 2 : * 3
```

which would be rejected, because there is no monadic * operator in Java. In our reduced syntax, this would be

```
1 x = a > b ? y + 2 * 3
```

which is a valid expression (though not the one the user intended) and so would be accepted by the parser.

This use of syntactic elements to catch common errors also explains why in Java and C the predicated of if, switch and while statements must be surrounded with parentheses, even though they carry no semantic information.

Another aspect of concrete syntax that is redundant in the derivation tree is the use of parentheses to enforce operator execution order in expressions. We have seen how to write grammar productions that enforce associativity and priority rules for operators in the absence of parentheses, and we have also seen that a fully parenthesized expression requires no such rules. In a tree, we use the depth of a node to encode its execution priority under the rules that the tree will be traversed top down, left to right with operators being executed in post-order. It is clear, then, that parentheses in the user expression may be omitted from the tree, and by the same argument other grouping elements such as braces around compound statements may be suppressed without losing fidelity.

4.6 The legacy of non-general parsing

A further source of redundancy in concrete grammars as typically found in language standard documents such as that for ANSI-C is that they have been written so as to be admissable by traditional deterministic parsing algorithms, and as such they can contain complicated BNF constructs which could be simplified for use with a general parser. This problem also affects language *exposition* for human readers. For instance, the first version of the Java Language Specification contains two grammars which we call the *pedagogic* and the *near-deterministic* grammars. In the main body of the document, individual language constructs are introduced with a grammar fragment that describes their syntax, accompanied by an informal English-language description of the semantics. The union of all these grammar fragments specifies the language, but unfortunately simply concatenating the pedagogic grammar fragments does not yield a grammar that is admissable by traditional parser generators. As a result, the JLS authors provide a second grammar which would be admissable, and describe its relationship with the pedagogic grammar so as to convince the reader that they generate the same language. With a more powerful parsing technology, it might have been directly use the pedagogic grammar, reducing the scope for errors.

The JLS example reinforces our expectation that even informal semantics are conventionally defined over the compositional syntax of the language. In practice, this means that we need to specify semantics with respect to particular productions, and would be very convenient to have productions (and thus nonterminals) which in some sense reflect the semantic concepts within the language in as simple a way as possible so as to reduce the number of formal cases

that have to be specified. Abstract syntax for formal semantics systems emphasise this notion of *compressing* the concrete syntax into a concise form which matches *at least* all of the strings in the concrete language. In practice, inner grammars often match larger languages, and are typically highly ambiguous. When building tools for such systems, we use a concrete parser to produce an individual derivation tree in the concrete syntax, and then give rules for, say, discarding redundant terminals and merging the children of nonterminals so as to produce a derivation in our inner syntax. The formal semantics interpreter can then work on the simplified tree.

There are a variety of ways to specify these outer to inner mappings. We could simply write a program in a general purpose programming language to traverse the outer derivation and build the corresponding inner derivation. This rather misses the point for using formal approaches though, since it would usually be hard to ensure that the translator was complete (catering for all cases) and correct. Alternatively, we could use an *attribute grammar* to formalise the relationship between outer and inners suntax (see Chapter **??**), or we could use a set of equations to show how terms in the outer grammar should be rewritten to terms in the inner grammar, as we shall see in Chapterrewriting, or we could use some less general technique which rewrote the concrete tree under the control of a set of convenient tree annotations. We shall examine one such set of operations called the Gather-Insert-Fold-Tear (GIFT) formalism in Chapter **??**.

Whichever technique we use, we must first develop a derivation of our input in the outer syntax, and for that we shall need a parser.

4.7 Parsing by expanding the start symbol

**** Todo: Classical RD parsing; Backtrack RD parsing;GLL**

4.8 Parsing by reducing to the start symbol

**** Todo: NFA;DFA;Shift-reduce automaton**

4.9 Multiparsing and the lexer-parser interface

**** Todo: Lexer parser interface Regular expressions Thompson's algorithm Subset construction State minimisation Generating all lexicalisations (multi) Lexing**

4.10 OSBRD: Implementing a parser toolchain

When we implement a translator, we parse the source language into an intermediate form, and then traverse the intermediate form outputting the object language.

A *parser generator* is a program which reads specifications for a grammar Γ written in BNF (or EBNF) and outputs the source code for a parser. When

compiled the parser will test strings to see if they are in the language $L(\Gamma)$, and perhaps build a derivation tree.

Parser generators, then, can be thought of as processors for a DSL (BNF) which translates to, say, Java. Embedded within each parser will be a *parsing algorithm*. We shall illustrate this process using ordered singleton backtrack recursive descent parsing (OSBRD) which is a rather limited algorithm: it's advantage is that it is easy to understand and easy to generate. This means that it is possible to fully explain the internals not just of the parsers but of the program that writes out the parsers.

4.11 Ordered Singleton Backtrack Recursive Descent parsing

OSBRD is a long acronym for a very simple parsing technique. The parsers may be written by hand, and there is no reqiurement to compute properties of the grammar: in fact an OSBRD parser can be produced as a syntax-directed translation from BNF; it is in effect a pretty-printed version of the grammar.

OSBRD is not used for production parsers because (a) the performance of OSBRD parsers is exponential in the length o fth einput string for some 'nasty' grammars and (b) because OSBRD parsers fail to recognise some strings that are in the language of the grammar being parsed.

(b) sounds like a show stopper, but in fact the commonly-used parsing techniques such as LALR(1), SLR(1) and LL(1) all suffer from the same problem. In fact any non-general parsing technology will fail to accept some strings for some grammars. However, it is possible to compute in advance whether a grammar is LALR(1) (or SLR(1) or LL(1)...) and so the user is at least told that their grammar will not behave as they expect. Although we could do some processing to help the user (and in our OSBRD toolchain we do test for one obvious error condition as we shall see) a basic syntax driven translation produces a parser which silently misbehave. The user can write what appears to be a perfectly reasonable grammar, have a parser generated and then find that it does not work as it should: we call these situations *nasty suprises*.

4.11.1 The OSBRD algorithm

Consider a grammar $\Gamma = (N, T, X_S, P)$ where, as usual, N is a set of nonterminals $\{X_1, X_2, X_3, \ldots, X_k\}$, T is a set of terminals, X_S is the start nonterminal and P is a a set of productions $\{X \to \rho, \rho \in (N \cup T)^*\}$.

The OSBRD algorithm works on *ordered* grammar. In an ordered grammar the subset of productions $\{X_i \to \rho_1, X_i \to \rho_2, \ldots\}$ are ordered, and are tested in that order. There is no ordering associated with the nonterminals or the terminals; it is just the order or productions *within* a particular nonterminal X_i that is significant.

This seemingly innocuous change has a big impact on the languages that can be successfully parsed by OSBRD compared to a truly general technique such as GLL. We are highlighting this difference here because occasionally one

encounters parsing tools which use algorithms based on ordered grammars, and in my experience the authors often do not adequately explain the limitations of the technique.

Informally, an OSBRD parser is a set of (possibly recursive) functions, one per nonterminal. The functions take no parameters, and return a boolean. The input string is held in a buffer `String input` and there is a global variable `int cc` which holds the index of the *current character*.

At the start of the parse function for nonterminal X_i, the value of `cc` on entry is remembered in a local variable `int rc` which holds the index of the *restart character*. Each alternate production $X_i \rightarrow \rho_j$ is then laid out as a nest of `if` statements: for a terminal we test against a direct match; for a nonterminal we call the appropriate parse function and for ϵ we do nothing. If the nest evaluates true, then we have found a match against that alternate, and so the parse function returns true. If not, we proceed to alternate $X_i \rightarrow \rho_{j+1}$. If all alternate productions fail, the parse function returns false.

Each parse function also remembers which alternate (if any) succeeded. The running parser maintains a global array of integers called the *oracle* and a global variable `int co` which holds the index of the next free slot in the oracle.

An OSBRD parser explores the grammar by recursively calling the parse functions. Sometimes these exploration fail after severl layers of function call, and in that case the parser *backtracks* to the next level up so as to continue testings its alternate productions. In fact the backtracking can recursively unwind an arbitray number of levels. As a result, we need to remember where we were in the oracle when we entered the parse function so that we can rest `co` at the start of each alternate; the local variable `int ro` remembers this restart oracle index.

4.11.2 An OSBRD example in Java

Here is a small grammar.

```
S ::= 'b' | 'a' X '@'
X ::= 'x' X | #
```

We specify terminals within single quotes, and ϵ is written as `#`. Alternate productions are separated by a vertical bar, and each rule is terminated with a period. Repeated nonterminals are not allowed.

The language of this grammar is `{ b, a@, ax@, axx@ axxx@,... }`.

When processed by the OSBRD parser generator, the following two Java parse functions are produced:

```
boolean parse_S() {
  int rc = cc, ro = co;

  /* Nonterminal S, alternate 1 */
  cc = rc; co = ro; oracleSet(1);
```

```
6    if (match("b")) { return true; }
7
8    /* Nonterminal S, alternate 2 */
9    cc = rc; co = ro; oracleSet(2);
10   if (match("a")) {
11   if (parse_X()) {
12   if (match("@")) { return true; }}}
13
14   return false;
15 }
16
17 boolean parse_X() {
18   int rc = cc, ro = co;
19
20   /* Nonterminal X, alternate 1 */
21   cc = rc; co = ro; oracleSet(1);
22   if (match("x")) {
23   if (parse_X()) { return true; }}
24
25   /* Nonterminal X, alternate 2 */
26   cc = rc; co = ro; oracleSet(2);
27   /* epsilon */ return true;
28 }
```

Let us follow this code through whilst parsing the string `axx@`.

The parser initially loads `input` with the string `axx@`$\boxed{\$}$ where $\boxed{\$}$ is not the dollar character but rather stands for some special end-of-string marker. (In Java, we use the character containing zero or `'\0'`; regular expression processors and parsing texts conventionally use some variant of the dollar symbol.) The oracle does not need to be initialised, but the two global variables that index the input and the oracle are zeroed: `cc = co = 0`

We start the parse by calling the parse function for the start symbol `parse_S()`.

`parse_S()` remembers the entry values for the input and oracle indices before executing the clauses for the alternates in sequence.

Each clause begins by setting the gloal indices to these restart values before testing the input against the production using a nest of predicates each of which either call `match()` to test a terminal or call the relevant `parse_()` function. If none of the clauses succeed, then the `return false;` statement is executed.

By instrumenting the parse we can produce a trace of the function calls and terninal matches. Here is the output from one run.

```
Input: 'a x x @'
S() at rc = 0, cc = 0 'a'
  S() alternate 1 rc = 0, cc = 0 'a'
    At 0 'a' match b - reject
  S() alternate 2 rc = 0, cc = 0 'a'
    At 0 'a' match a - accept
```

```
X() at rc = 2, cc = 2 'x'
  X() alternate 1 rc = 2, cc = 2 'x'
    At 2 'x' match x - accept
X() at rc = 4, cc = 4 'x'
  X() alternate 1 rc = 4, cc = 4 'x'
    At 4 'x' match x - accept
X() at rc = 6, cc = 6 '@'
  X() alternate 1 rc = 6, cc = 6 '@'
    At 6 '@' match x - reject
  X() alternate 2 rc = 6, cc = 6 '@'
    At 6 '@' match @ - accept
Accepted
Oracle: 2 1 1 2
```

This shows us that a call to `S()` with `cc=0` tried alternates 1 and 2, before a call to `X()` with `cc=2` tries alternate 1 and then calls `X()` with `cc=4` which calls `X()` with `cc=6`. Alternate 1 cannot match `@` tp `x`, so alternate 2 is tried instead which must succeed because it is an ϵ-production.

All of the calls then unwind, and because the call to S() terminates with the current character `cc` pointing to the end of string marker, the string is `Accepted`.

The parser then prints out the oracle which (when combined with the grammar) encodes the successful derivation: we took alternates 2, 1, 1 and 2 as we went down through the nest of parse functions.

4.12 Engineering a complete Java parser

The parse functions in the previous section make use of auxilliary functions like `match()` and also require some initialisation code. In this section we shall look at how the generated parse functions are embedded into Java classes so as to make a standalone parser.

A minimal OSBRD parser comprises two classes in two source files:

1. `ARTOSBRDBase.java` which contains auxiliary methods.
2. `ARTGeneratedParser.java` which extends class `ATYOSBRDBase` with the parse member functions and a `main()` function which processes command line arguments.

Here is the contents of `ARTGeneratedParser.java` for a simple parse — in later sections we shall add more functions to support semantics processing and tree construction.

```
1 import java.io.File;
2 import java.io.PrintWriter;
3 import java.io.FileNotFoundException;
4 import java.util.Scanner;
5 import java.util.ArrayList;
```

```
6
7  class ARTGeneratedParser extends uk.ac.rhul.cs.csle.artosbrd.ARTOSBRDBase {
8    String input;
9    int cc, co, oracleLength, oracle[];
10   int ts, te;
11
12 ARTGeneratedParser() { oracleLength = 1000; oracle = new int[oracleLength]; cc = co = 0;}
13
14 String readInput(String filename) throws FileNotFoundException {
15   return new Scanner(new File(filename)).useDelimiter("\\Z").next() + "\0";
16 }
17
18 void oracleSet(int i) {
19   if (co == oracleLength) {
20     int oracleLengthOld = oracleLength;
21     oracleLength += oracleLength / 2;
22     int newOracle[] = new int[oracleLength];
23     System.arraycopy(oracle, 0, newOracle, 0, oracleLengthOld);
24     oracle = newOracle;
25   }
26   oracle[co++] = i;
27 }
28
29 boolean match(String s) {
30   if (input.regionMatches(cc, s, 0, s.length())) {
31     cc += s.length();
32     builtIn_WHITESPACE();
33     return true;
34   }
35   return false;
36 }
37
38 boolean builtIn_WHITESPACE() {
39   while(Character.isWhitespace(input.charAt(cc)))
40     cc++;
41   return true;
42 }
43 }
```

The constructor sets the initial oracle length 1000, creates the array and zeroes the `cc` and `co` indices. Loading the input string is the responsibility of the generated parse class `SBxyz`, though it makes use of function`readInput()` which automatically appends an end-of-string marker `'\0'`.

The `oracleSet(int i)` function resizes the oracle if necessary (adding 50% to its length at each resize operation) before loading the supplied alternate number `i` into the oracle and incrementing `co`, the current oracle index.

The `match(String s)` function checks to see whether a substring of `input`

starting at character `cc` matches parameter `s`. If so, then a helper function `builtin_WHITESPACE()` is called to absorb any blank spaces and line ends after the string. This allows generated parsers to treat the strings `axx@` and `a x x @` as equivalent.

If the example grammar is in file `test.sb`, the parser generator generates this file `SBtest.java`.

```
1  import java.io.FileNotFoundException;
2
3  class ARTGeneratedParser extends uk.ac.rhul.cs.csle.artosbrd.ARTOSBRDBase {
4  boolean parse_S() {
5    int rc = cc, ro = co;
6
7    /* Nonterminal S, alternate 1 */
8    cc = rc; co = ro; oracleSet(1);
9    if (match("b")) { return true; }
10
11   /* Nonterminal S, alternate 2 */
12   cc = rc; co = ro; oracleSet(2);
13   if (match("a")) {
14   if (parse_X()) {
15   if (match("@")) { return true; }}}
16
17   return false;
18 }
19
20 boolean parse_X() {
21   int rc = cc, ro = co;
22
23   /* Nonterminal X, alternate 1 */
24   cc = rc; co = ro; oracleSet(1);
25   if (match("x")) {
26   if (parse_X()) { return true; }}
27
28   /* Nonterminal X, alternate 2 */
29   cc = rc; co = ro; oracleSet(2);
30   /* epsilon */ return true;
31 }
32
33 SBtest(String filename) throws FileNotFoundException {
34   input = readInput(filename);
35
36   System.out.printf("Input: '%s'%n", input);
37   cc = co = 0; builtIn_WHITESPACE();
38   if (!(parse_S() && input.charAt(cc) == '\0'))
39     { System.out.print("Rejected%n"); return; }
```

```
40
41    System.out.print("Accepted%n");
42    System.out.print("Oracle:");
43    for (int i = 0; i < co; i++) System.out.printf(" %d", oracle[i]);
44    System.out.printf("%n");
45  }
46
47  public static void main(String[] args) throws FileNotFoundException{
48    if (args.length < 1)
49          new SBtest("");
50    else
51          new SBtest(args[0]);
52    }
53  }
```

The `main()` function collects the name of a string file from the input and then instances `SBtest`, passing the filename as an argument to the constructor.

The constructor loads the`input` string from the supplied filename; prints it out; and then calls `builtin_WHITESPACE()` to consume any leading blanks in the input string.

If the start symbol's parse function consumes the entire string up to but not including the end-of-string marker, the string is accepted and the oracle printed out.

4.13 Using built in matchers

The OSBRD base class provides a set of builtin matchers which can be used to efficiently parse identifiers, numeric literals, strings and so on. It is quite easy to add new builtins as required.

The parser generator translates pseudo-terminals such as `&ID` into calls to the corresponding builtin matcher. In detail, `&XYZ` is trabslated to a call to `builtin_XYZ()`. The generator does not check the name of the builtin, so just by adding a new builtin member function to class `uk.ac.rhul.cs.csle.artosbrd.ARTOSBRDBase` we can extend the repetoire of builtin matchers.

Here is a slightly modified version of our test grammar.

```
1  S ::= 'b' | 'a' X '@' .
2  X ::= &ID X | # .
```

It generates these parse functions.

```
1  boolean parse_S() {
2    int rc = cc, ro = co;
3
4    /* Nonterminal S, alternate 1 */
5    cc = rc; co = ro; ora cleSet(1);
```

```
6     if (match("b")) { return true; }
7
8     /* Nonterminal S, alternate 2 */
9     cc = rc; co = ro; oracleSet(2);
10    if (match("a")) {
11    if (parse_X()) {
12    if (match("@")) { return true; }}}
13
14    return false;
15  }
16
17  boolean parse_X() {
18    int rc = cc, ro = co;
19
20    /* Nonterminal X, alternate 1 */
21    cc = rc; co = ro; oracleSet(1);
22    if (builtIn_ID()) {
23    if (parse_X()) { return true; }}
24
25    /* Nonterminal X, alternate 2 */
26    cc = rc; co = ro; oracleSet(2);
27    /* epsilon */ return true;
28  }
```

which are identical to the previous versions except that the call to `match("x")` has been replaced by a call to `builtIn_ID()`.

Here is the source code for a set of builtins. Note how each of them calls `builtin_WHITESPACE()` after matching. Each builtin matcher remembers the start and end indices of the matched terminal in global variables `int ts` and `int te`.

```
1   boolean builtIn_ID() {
2     if (!Character.isJavaIdentifierStart(input.charAt(cc))) return false;
3     ts = cc++;
4     while (Character.isJavaIdentifierPart(input.charAt(cc)))
5       cc++;
6     te = cc;
7     builtIn_WHITESPACE();
8     return true;
9   }
10
11  boolean isxdigit(char c) {
12    if (Character.isDigit(c)) return true;
13    if (c >= 'a' && c<='f') return true;
14    if (c >= 'A' && c<='F') return true;
15    return false;
16  }
```

```
17
18 boolean builtIn_INTEGER() {
19   if (!Character.isDigit(input.charAt(cc))) return false;
20   ts = cc;
21   /* Check for hexadecimal introducer */
22   boolean hex = (input.charAt(cc) == '0' &&
23                  (input.charAt(cc + 1) == 'x' ||
24                   input.charAt(cc + 1) == 'X'));
25   if (hex) cc += 2; // Skip over hex introducer
26   /* Now collect decimal or hex digits */
27   while (hex ? isxdigit(input.charAt(cc)) :
28                Character.isDigit(input.charAt(cc)))
29     cc++;
30   te = cc;
31   builtIn_WHITESPACE();
32   return true;
33 }
34
35 boolean builtIn_REAL() {
36   if (!Character.isDigit(input.charAt(cc))) return false;
37   ts = cc;
38   while (Character.isDigit(input.charAt(cc)))
39     cc++;
40   if (input.charAt(cc) != '.')
41     return true;
42   cc++; // skip .
43   while (Character.isDigit(input.charAt(cc)))
44     cc++;
45   if (input.charAt(cc) == 'e' || input.charAt(cc) == 'E') {
46     cc++;
47   while (Character.isDigit(input.charAt(cc)))
48     cc++;
49   }
50   te = cc;
51   builtIn_WHITESPACE();
52   return true;
53 }
54
55 boolean builtIn_CHAR_SQ() {
56   if (input.charAt(cc) != '\'') return false;
57   cc++;
58   ts = cc;
59   if (input.charAt(cc) == '\\')
60     cc++;
61   cc++;
62   if (input.charAt(cc) != '\'') return false;
63   te = cc;
```

```
64    cc++; // skip past final delimiter
65    builtIn_WHITESPACE();
66    return true;
67  }
68
69  boolean builtIn_STRING_SQ() {
70    if (input.charAt(cc) != '\'') return false;
71    ts = cc + 1;
72    do {
73      if (input.charAt(cc) == '\\')
74        cc++;
75
76      cc++;
77    }
78    while (input.charAt(cc) != '\'');
79    te = cc;
80    cc++; // skip past final delimiter
81    builtIn_WHITESPACE();
82    return true;
83  }
84
85  boolean builtIn_STRING_DQ() {
86    if (input.charAt(cc) != '"') return false;
87    ts = cc + 1;
88    do {
89      if (input.charAt(cc) == '\\')
90        cc++;
91      cc++;
92    }
93    while (input.charAt(cc) != '"');
94    te = cc;
95    cc++; // skip past final delimiter
96    builtIn_WHITESPACE();
97    return true;
98  }
99
100 boolean builtIn_ACTION() {
101   if (!(input.charAt(cc) == '[' &&
102       input.charAt(cc + 1) == '*'))
103     return false;
104   cc += 2;
105   ts = cc;
106   while (true) {
107     if (input.charAt(cc) == 0)
108       break;
109     if (input.charAt(cc) == '*' && input.charAt(cc) == ']') {
110       cc += 2;
```

```
111         break;
112       }
113       cc++;
114     }
115     te = cc - 2;
116     builtIn_WHITESPACE();
117     return true;
118 }
```

4.14 Using attributes and inline semantics

Our generated OSBRD parsers will execute embedded semantic actions which may also use synthesized attributes. The current version does not support inherited attributes, but it is not hard to extend the parser generator to allow that. Only a single pass is made over the input string which significantly limits the kinds of behaviour which may be generated. However, the generated parsers can also generate explicit derivation trees which may be passed to a back end for arbitrary processing: we shall look at tree generation in the next section.

An embedded action is delimited bt `{ }` brackets and must be written in the implementation language for the generated parser (in our case Java, although an ANSI C++ versions exist for which actions must be written in C++).

Here is our example grammar extended with an action to report the location of matching x characters.

```
1 S ::= 'b' | 'a' X '@' .
2 X ::= 'x' [* System.out.printf("Matched an x at location %d%n", cc); *] X | # .
```

The generated parser running on the string `a x x @` displays:

```
Input: 'a x x @ '
Accepted
Oracle: 2 1 1 2

Semantics phase
Matched an x at location 4
Matched an x at location 6
```

The parse is as before. After parsing is completed, a second pass is made during which the semantics are executed.

4.14.1 Attributes

Simply printing out messages showing where we are in a parse is interesting, but limited. If we want to perform useful computations, it turns out that we

need to pass information *between* parse functions or, equivalently, around the derivation tree.

Recursive descent parsers provide a natural built-in mechanism for passing information around: we can use the parse function parameters to pass information down the derivation tree and the function return values to pass information up.

The formal underpinnings for this approach are part of the theory of *attribute grammars*. Attributes are classified as *synthesized* which means that move up the tree (like a return result) or *inherited* which means that they pass down the tree (like a parameter). In a general attribute grammar information can move round the derivation tree in arbitrary ways by making use of inherited and synthesized atteibutes, and the calculation of the fial result requires an analysis of the dependency relationships between attribute definitions and their users. An *attribute evaluator* is a general tool for doing just that.

Two useful classes of attribute grammar are the *L-attributed* class in which attributes must be resolvable in a single top-down left to right pass and *S-attributed* grammars which may only contain synthesized arributes. Recursive descent parsers naturally support L-attribited grammars whilst bottom up parsing techniques such as LALR(1) (that is, Bison and YACC) naturally support S-attributed grammars. (In detail, tools often also make use of global attributes which extends their power a little.)

Here is a grammar that uses synthesized attributes to add up the number of 1's seen in a binary string:

```
1 S ::= 'b' | 'a' X:result [* System.out.printf("Result is %d\n", result); *] '@' .
2 X:int ::= '1' X:sum [* rv = sum + 1; *] |
3           '0' X:rv | # .
```

The language of this grammar is

```
{ b, a@, a1@, a0@, a11@, a10@, a01@, a00@, a111@, ...}.
```

The attributes and associated semantic actions implement a recursive function that runs along the string of 1's and 0's maintaining a count of the number of 1's seen.

The return value attribute `rv` is automatically defined for any parse function that has an associated type, and is used to carry the synthesized information back up the tree, or equivalently to pass it back to the calling function. At the top level, the accumulated value is printed out.

Sandbox decides on the type of attributes by looking at the type annotation for the left hand side of the associated nonterminal. In this case, since nonterminal `X` is declared as being of type `int`, the `sum` and `result` attributes will also be of type `int`.

The result of running this parser on the string `a101@` is

```
Input: 'a 1 0 1 @ '
Accepted
Oracle: 2 1 2 1 3
```

```
Semantics phase
Result is 2
```

4.14.2 A four function calculator

Let us now extend our example to a more general computing language: a four function calculator for integer constants of one, two or three digits. Warning: Sandbox parsers do not allow left recursion, so all of the operators have been implemented in right associative form, whereas they should really be left associative.

```
S ::= exprs:val [* System.out.printf("Final result: %d\n", val); *] .

exprs:int ::= add:val ';' [* System.out.printf("Result: %d\n", val); *] exprs:rv |
              add:rv [* System.out.printf("Result: %d\n", rv); *] .

add:int ::= mul:l '+' add:r [* rv = l + r; *] |
            mul:l '-' add:r [* rv = l - r; *] |
            mul:rv .

mul:int ::= op:l '*' mul:r [* rv = l * r; *]|
            op:l '/' mul:r [* rv = l / r; *]|
            op:rv .

op:int ::= integer:rv |
           '(' exprs:rv ')' .

integer:int ::= digit:hi digit:mid digit:lo
                    [* rv = hi*100 + mid*10 + lo; *] |
                digit:mid digit:lo [* rv = mid*10 + lo; *] |
                digit:rv .

digit:int ::= '0' [* rv = 0; *] |
              '1' [* rv = 1; *] |
              '2' [* rv = 2; *] |
              '3' [* rv = 3; *] |
              '4' [* rv = 4; *] |
              '5' [* rv = 5; *] |
              '6' [* rv = 6; *] |
              '7' [* rv = 7; *] |
              '8' [* rv = 8; *] |
              '9' [* rv = 9; *] .
```

When the generated parser is run on the string `3  4; 10; (7*2)+1+` we get the following output

```
Input: '3 + 4; 10; (7*2)+1 '
Accepted
Oracle: 1 1 1 3 1 3 4 3 3 1 3 5 1 3 3 1 2 2 1 2 1 3 2 2 3 1 1 3
        8 3 1 3 3 3 3 1 3 2

Semantics phase
Result: 7
Result: 10
Result: 14
Result: 15
Final result: 15
```

4.15 Implementing inline semantics

OSBRD parsers explore the grammar in a way that may require tentative matches that are subsequently rejected. Whenever a parser backtracks, some decisions are being unmade.

As a result of this retry behaviour, we cannot simply execute inline semantics during the parse, even though when we design grammars and their semantic actions we tend to think of the action being executed as a side-effect of parsing. Instead, we need to complete the searching associated with parsing and only then run through the grammar the 'correct' way to execute the actions. This is the purpose of the oracle: during parsing we construct the oracle as we go, adjusting it as necessary when we backtrack. By the end of the parse we have a map of where the parser *should* have gone. We call the control data structure an oracle because it is as if we had a parser which instead of guessing where to go could simply ask an all-powerful oracle for advice.

To execute the semantics, we use a modified set of parse functions (the *semantics* functions) that (a) contains the embedded semantic actions and (b) look in the oracle to see where to go rather than searching and backtracking.

Recall the attributed grammar that adds up the 1's in a string:

```
1 S ::= 'b' | 'a' X:result [* System.out.printf("Result is %d\n", result); *] '@' .
2 X:int ::= '1' X:sum [* rv = sum + 1; *] |
3           '0' X:rv | # .
```

Here are the associated semantics functions.

```
1 void semantics_S() {
2 int result;
3   switch(oracle[co++]) {
4     case 1:
5       match("b");
6       break;
7
8     case 2:
```

```
9           match("a");
10          result = semantics_X();
11      System.out.printf("Result is %d%n", result);
12
13          match("@");
14          break;
15      }
16  }
17  int semantics_X() {
18      int rv = 0;
19  int sum;
20      switch(oracle[co++]) {
21          case 1:
22              match("1");
23              sum = semantics_X();
24      rv = sum + 1;
25
26              break;
27
28          case 2:
29              match("0");
30      rv = 0;
31
32              rv = semantics_X();
33              break;
34
35          case 3:
36              /* epsilon */
37              break;
38      }
39      return rv;
40  }
```

Note that the semantics functions here are void functions taking no parameters unless we declare a type for their associated nonterminals. Functions with a type T automatically have a local variable `rv` declared of type T which holds the return value; in addition the statement `return rv;` is inserted at the end of the corresponding function.

It is the user's responsibility to ensure that `rv` is loaded with a suitable value. There are two ways to get a value into `rv`: (i) by explictly assigning to it using a semantic action as in the first alternate of nonterminal `X` and (ii) implictly assigning to it by naming `rv` as the attribute receiving a synthesized result from a nonterminal, as in the second alternate of nonterminal `X`.

The overall control flow is *via* switch statements selecting on the current oracle index; as each element of the oracle is consumed, the index is incremented by one. The use of switch statements is fast compared to the sequential testing requirted in the parse functions.

4.16 Making explicit trees

The oracle combined with the semantics functions encode the derivation of a string, but in a rather implicit way that lends itself only to the evaluation of L-attributed grammars. If our semantics specification mandates multiple passes over the tree, or random access into the tree, then the semantics functions are not helpful. For these kinds of applications it is preferable to construct the explicit tree as a datastructure in memory that we can traverse in any way we see fit. (General formal attribute evaluators work this way, as well as the rather informal translators that we design on this course.)

Sandbox makes trees by building a specialised set of semantics functions whose sole actions are to construct trees. Here are the tree construction functions for the previous example grammar:

```
1  TreeNode tree_S() {
2    TreeNode leftNode = null, rightNode = null;
3    switch(oracle[co++]) {
4      case 1:
5        /* 'b' */ leftNode = rightNode =
6        new TreeNode("b", null, rightNode, TreeKind.TREE_TERMINAL,
7                     TIFKind.TIF_NONE, null);
8        match("b");
9        break;
10
11     case 2:
12       /* 'a' */ leftNode = rightNode =
13       new TreeNode("a", null, rightNode, TreeKind.TREE_TERMINAL,
14                    TIFKind.TIF_NONE, null);
15       match("a");
16       /* X */ rightNode =
17       new TreeNode("X", tree_X(), rightNode, TreeKind.TREE_NONTERMINAL,
18                    TIFKind.TIF_NONE, null);
19       /* '@' */ rightNode =
20       new TreeNode("@", null, rightNode, TreeKind.TREE_TERMINAL,
21                    TIFKind.TIF_NONE, null);
22       match("@");
23       break;
24
25   }
26   return leftNode;
27 }
28 TreeNode tree_X() {
29   TreeNode leftNode = null, rightNode = null;
30   switch(oracle[co++]) {
31     case 1:
32       /* '1' */ leftNode = rightNode =
33       new TreeNode("1", null, rightNode, TreeKind.TREE_TERMINAL,
```

```
34                          TIFKind.TIF_NONE, null);
35          match("1");
36          /* X */ rightNode =
37          new TreeNode("X", tree_X(), rightNode, TreeKind.TREE_NONTERMINAL,
38                          TIFKind.TIF_NONE, null);
39          break;
40
41        case 2:
42          /* '0' */ leftNode = rightNode =
43          new TreeNode("0", null, rightNode, TreeKind.TREE_TERMINAL,
44                          TIFKind.TIF_NONE, null);
45          match("0");
46          /* X */ rightNode =
47          new TreeNode("X", tree_X(), rightNode, TreeKind.TREE_NONTERMINAL,
48                          TIFKind.TIF_NONE, null);
49          break;
50
51        case 3:
52          /* # */ leftNode = rightNode =
53          new TreeNode("#", null, rightNode, TreeKind.TREE_EPSILON,
54                          TIFKind.TIF_NONE, null);
55          break;
56
57      }
58      return leftNode;
59  }
```

Each parse function constructs a list of sibling tree nodes corresponding to the elements of the derivation tree, and returns the leftmost element to its parent. Treenodes are labelled with either (i) the name of the nonterminal for nonterminals, or (ii) the name of the terminal for terminals, or (iii) `#` for epsilon nodes. Now, we could in principle have both a nonterminal called `adrian` and a terminal `'adrian'` and we need to be able to distinguish between them which the names alone will not do. (we have the same possible clash between a terminal `'#'` and the epsilon symbol. The solution is to additionally label each node with an element of a `TreeKind` enumeration.

There are two other bits of information that we might want to put into a tree node, neither of which is in use in this example: (i) we might wish to add a TIF operator and (ii) for builtins we might want to know not only the name of the builtin, but the substring that it matched. A instance of `&ID` that matched `adrian` needs to store the pair `(ID, adrian)`. The `null` parameter in the above example is a placeholder for this attribute information: since this example does not use builtins, the parameter is always null.

4.16.1 The TreeNode class

The operation of the tree bundling functions is dependent on the behaviour of class `TreeNode` which is a nested class of `Sandbox`. The core design issue is that we wish to efficiently represent trees of arbitrary out-degree. Now, there are three main ways to represent tree-like structures.

- ◇ Decide a maximum out-degree n and create a TreeNode class that contains members that represent the node's label and n references to other nodes. This model is memory inefficient for nodes with low out-degree, and in any case has a fixed upper bound on out degree. (For derivation trees of BNF grammars, we can at least measure the maximum required outdegree because it would be the length of the longest right hand side; for EBNF and for grammars with TIF annotations it is not in general possible to precompute a maximum length.)

- ◇ Create a TreeNode class that contains the node's label and one reference to a linked list of TreeEdge objects; the TreeEdge class contains a reference to the rest of the list and a reference to a TreeNode. In this model, then a tree node points to a list of edge nodes; each edge node points to a tree node. This model is sufficiently general to model arbitrary directed graphs (which of course include trees) but is memory inefficient in that each edge needs two references though each node needs only one reference.

- ◇ Create a TreeNode class that contains the node's label along with a reference to the first child node and a reference to the rightmost sibling node.

Of these three, the last one is the best choice for us since we do not need the generality of the second scheme and the first scheme is fatally flawed. This design decision explains the idiom used in the tree construction functions. Here is a fragment

```
1  TreeNode tree_X() {
2    TreeNode leftNode = null, rightNode = null;
3    switch(oracle[co++]) {
4      case 1:
5        /* '1' */ leftNode = rightNode =
6        new TreeNode("1", null, rightNode, TreeKind.TREE_TERMINAL,
7                     TIFKind.TIF_NONE, null);
8        match("1");
9        /* X */ rightNode =
10       new TreeNode("X", tree_X(), rightNode, TreeKind.TREE_NONTERMINAL,
11                    TIFKind.TIF_NONE, null);
12       break;
13
14     ...
15
16   }
```

```
17    return leftNode;
18  }
```

Tree funcion `tree_X()` has to create a sequence of children corresponding to one of the alternates of nonterminal `X`. Each `TreeNode` instance is created with a `new TreeNode()` operation, and these are formed into a list by remembering the most recently created (rightmost) `TreeNode` in local variable `rightNode`. We also remember the head of the list in local variable `leftNode`; this is then returned at the end of the tree function.

The constructor takes as paramaters the node's label, a reference to the first child node and a reference to the left sibling. Inside the constructor, the left sibling's `sibling` reference is updated to point to the newly created `TreeNode`. That part of the code is perhaps the most subtle element of sandbox's implementation: it repays study. The other fields are the 'kind' of the node (nonterminal, terminal, builtin terminal or epsilon), a TIF operator and the substring matched by a builtin.

Here is the source for the `TreeNode` class constructors, along with some helper methods from Sandbox that are used to render enumeration elements. (Note that these helper methods should really be implemented as `asString` methods in the enumeration classes: this is a hangover from the original C++ implementation of Sandbox — C++ does not treat enumerations as full blown objects.)

```
1  enum TreeKind {TREE_EPSILON, TREE_TERMINAL, TREE_BUILTIN, TREE_NONTERMINAL};
2  enum TIFKind {TIF_NONE, TIF_FOLD_UNDER, TIF_FOLD_OVER, TIF_FOLD_ABOVE};
3
4  class TreeNode{
5    String label; int nodeNumber; TreeNode child; TreeNode sibling;
6    TreeKind kind; TIFKind tifOp; String attribute;
7
8    TreeNode(String label, TreeNode child, TreeNode previousSibling,
9             TreeKind kind, TIFKind tifOp, String attribute) {
10     if (previousSibling != null) previousSibling.sibling = this;
11     this.label = label; this.child = child; this.sibling = null;
12     this.kind = kind; this.tifOp = tifOp; this.attribute = attribute;
13     nodeNumber = nextNode++;
14   };
15
16   TreeNode(TreeNode old) {
17     label = old.label; kind = old.kind; tifOp = old.tifOp;
18     child = sibling = null;
19     attribute = old.attribute;
20     nodeNumber = nextNode++;
21   };
22
23 };
```

4.16.2 Cloning trees

```
1  TreeNode clone(TreeNode parent, TreeNode previousSibling) {
2    TreeNode ret = new TreeNode(this);
3    if (previousSibling != null) previousSibling.sibling = ret;
4    else if (parent != null) parent.child = ret;
5    TreeNode rightNode = null;
6    for (TreeNode srcNode = child; srcNode != null; srcNode = srcNode.sibling)
7      rightNode = srcNode.clone(ret, rightNode);
8    return ret;
9  }
```

4.16.3 Visualising trees on the console

In addition to the fields described above, every `TreeNode` object also contains a unique *node number* that can be very useful when debugging since it enables us to distinguish between otherwise-identical tree nodes that carry the same label.

```
1  void print(int indent) {
2      System.out.printf("%d: ", nodeNumber);
3      for (int temp = 0; temp < indent; temp++) System.out.printf(" ");
4      System.out.printf("%s%s%s", labelPreString(kind), label,
5                                  labelPostString(kind));
6      if (attribute != null) System.out.printf(":%s", attribute);
7      System.out.printf("%s\n", tifString(tifOp));
8
9      if (child != null) child.print(++indent);
10     if (sibling != null) sibling.print(indent);
11   };
```

4.16.4 Visualising trees with the GraphViz tools

OSBRD parsers write out files in the `.dot` format which can then be displayed graphically using the tools in the `GraphViz` toolset that is commonly available on Un*x systems and is available for Windows.

4.16.5 Implementing TIF operators

```
1  TreeNode evaluateTIF(TreeNode parent, TreeNode previousSibling,
2                       boolean parentSuppressed) {
3
4    // Special case: don't promote root node
5    if (parent != null && (tifOp == TIFKind.TIF_FOLD_UNDER ||
```

```
6                                   tifOp == TIFKind.TIF_FOLD_OVER))
7       {
8         /* Link the children in to the previousSibling's chain */
9         TreeNode rightNode = null;
10        if (previousSibling != null) rightNode = previousSibling;
11        else if (parent != null) rightNode = parent.child;
12
13        boolean suppress = tifOp == TIFKind.TIF_FOLD_UNDER ||
14              (parentSuppressed && tifOp == TIFKind.TIF_FOLD_OVER);
15
16        for (TreeNode srcNode = child; srcNode != null; srcNode = srcNode.sibling)
17          rightNode = srcNode.evaluateTIF(parent, rightNode, suppress);
18
19        if (tifOp == TIFKind.TIF_FOLD_OVER && !parentSuppressed) {
20          parent.label = label; /* What about tifOp? */
21          parent.kind = kind;
22          parent.attribute = attribute;
23        }
24
25        return rightNode;
26      }
27      else { /* make a new node and scan our children */
28        TreeNode ret = new TreeNode(this); ret.tifOp = TIFKind.TIF_NONE;
29        if (previousSibling != null) previousSibling.sibling = ret;
30        else if (parent != null) parent.child = ret;
31        TreeNode rightNode = null;
32        for (TreeNode srcNode = child; srcNode != null; srcNode = srcNode.sibling)
33          rightNode = srcNode.evaluateTIF(ret, rightNode, false);
34        return ret;
35      }
36    }
37
38    void foldunderEpsilon(){
39      if (kind == TreeKind.TREE_EPSILON)
40        tifOp = TIFKind.TIF_FOLD_UNDER;
41      for (TreeNode srcNode = child; srcNode != null; srcNode = srcNode.sibling)
42        srcNode.foldunderEpsilon();
43    }
```

4.17 A Sandbox grammar for Sandbox

```
1 grammar ::= ruleOrActions^ .
2
3 ruleOrActions ::= action ruleOrActions^ | rule ruleOrActions^ | # .
```

```
4
5  rule ::= nonterm '::='^ cats^ '.'^ .
6
7  cats ::= cat catTail^ .
8  cat::= element cat^ | # .
9  catTail ::= '|'^ cat catTail^ | # .
10
11 element ::= action^^ | subrule^^ | nonterm^^ tif | term^^ tif |
12                 builtIn^^ tif | epsilon^^ tif .
13
14 action ::= &ACTION .
15
16 subrule ::= subruleWrapper .
17 subruleWrapper ::= '('^ cats^ subruleKind^^ .
18 subruleKind ::= ')'^^ | ')?'^^ | ')+'^^ | ')*'^^ .
19
20 nonterm ::= nontermWrapper .
21 nontermWrapper ::= &ID^^ optionalAttribute .
22
23 term ::= termWrapper .
24 termWrapper ::= &STRING_SQ^^ optionalAttribute .
25
26 tif ::= '^^^'^^ | '^^'^^ | '^'^^ | #^^ .
27
28 builtIn ::= '&'^ &ID .
29
30 epsilon ::= '#'^.
31
32 optionalAttribute ::= ':'^ &ID^^ | #^^ .
```

4.18 The Gather-Insert-Fold-Tear formalism

It is useful to be able to compress derivation trees into trees which carry only such information from the derivation that we wish to carry forward into other stages of the translation process. Common transformations include:

- ◇ the suppression of recursion-scaffolding nodes,
- ◇ the construction of expression trees made up solely of nodes labeled with terminals,
- ◇ the suppression of entire sub-trees,
- ◇ the local reordering of sub-trees,
- ◇ the insertion of new pieces of tree.

The GIFT formalism provides a small set of operations with postfix annotations that specify their application to the tree nodes associated with grammar elements. We specify them by writing them into the grammar, but it is helpful to think of them being attached to tree nodes.

GIFT stands for Gather-Insert-Fold-Tear. The ART tool presently only implements the two Fold operations, but we shall discuss applications of the other operators. Collectively, the GIFT operations may be viewed as special cases of a more general approach called term rewriting, which allows tree to be rewritten using tree-to-tree rewrite rules.

The best way to think about the GIFT operators is that they are annotations that are loaded into the derivation tree, and that a GIFT rewriting phase then rewrites the derivation tree under the control of those operators into a Rewritten Derivation Tree (RDT).

4.18.1 Fold operators

The fold operators can only be applied to a node which has a parent: that is the root node may not be folded.

There are two kinds of fold: fold-under (^) and fold-over (^^).

A rule such as

```
X ::= `a `b `c^ `d
```

will generate an (as-yet-unrewritten) derivation subtree of the form

```
     X
  / / \ \
 a b c^ d
```

and rule such as

```
Y ::= `a `b `c^^ `d
```

will generate derivation subtree of the form

```
     X
  / / \ \
 a b c^^ d
```

The idea of the fold operators is that the edge joining the annotated node to its parent is folded in half so that the child node and the parent node are coincident. If we fold under (^) then the child goes under the parent; if we fold over then the child goes over the parent. Alternatively, you can see that for a fold under we delete the child node and keep the parent node; if we fold over then we delete the parent node and replace it with the child node.

For fold-under, then, we have

```
X ::= a b c^ d
```

gives

```
    X => X
 / / \ \ / / \
a b c^ d a b d
```

and

```
X ::= a b c^^ d
```

gives

```
    X => c
 / / \ \ / / \
a b c^^ d a b d
```

Note that this allows us to build trees which have *terminals* as internal nodes.

So far, we have only considered fold operators on terminal nodes, which have no children. If we apply a fold operator to a nonterminal instance, then we must explain how the children are to be treated. The metaphor of edge folding helps here: the children of the annotated node are inserted as a group into the siblings of the annotated node. We can think of this as the children being dragged up a level in the tree.

For fold-under

```
X ::= a b Y^ d
Y ::= y z
```

```
    X => X
 / / \ \ / / | \ \
a b Y^ d a b y z d
     /\
    y z
```

For fold-over

```
X ::= a b Y^^ d
Y ::= y z
```

```
    X => Y
 / / \ \ / / | \ \
a b Y^^ d a b y z d
     /\
    y z
```

4.18.2 The Tear operator

We can suppress an entire subtree by attaching the Tear (^^^) annotation.

```
X ::= a b Y^^^ d
Y ::= y z
```

```
      X => X
 / / | \ / | \
a b Y^^^ d a b d
     /\
    y z
```

4.18.3 Insertions

Nodes can be named by appending a colon and an identifier, and named tear nodes can be inserted elsewhere in the tree:

```
X ::= a b Y:t^^^ d [t]
Y ::= y z
```

```
      X => X
 / / | \ / | | \
a b Y^^^ d a b d Y
     /\ /\
    y z y z
```

4.18.4 The Gather operator

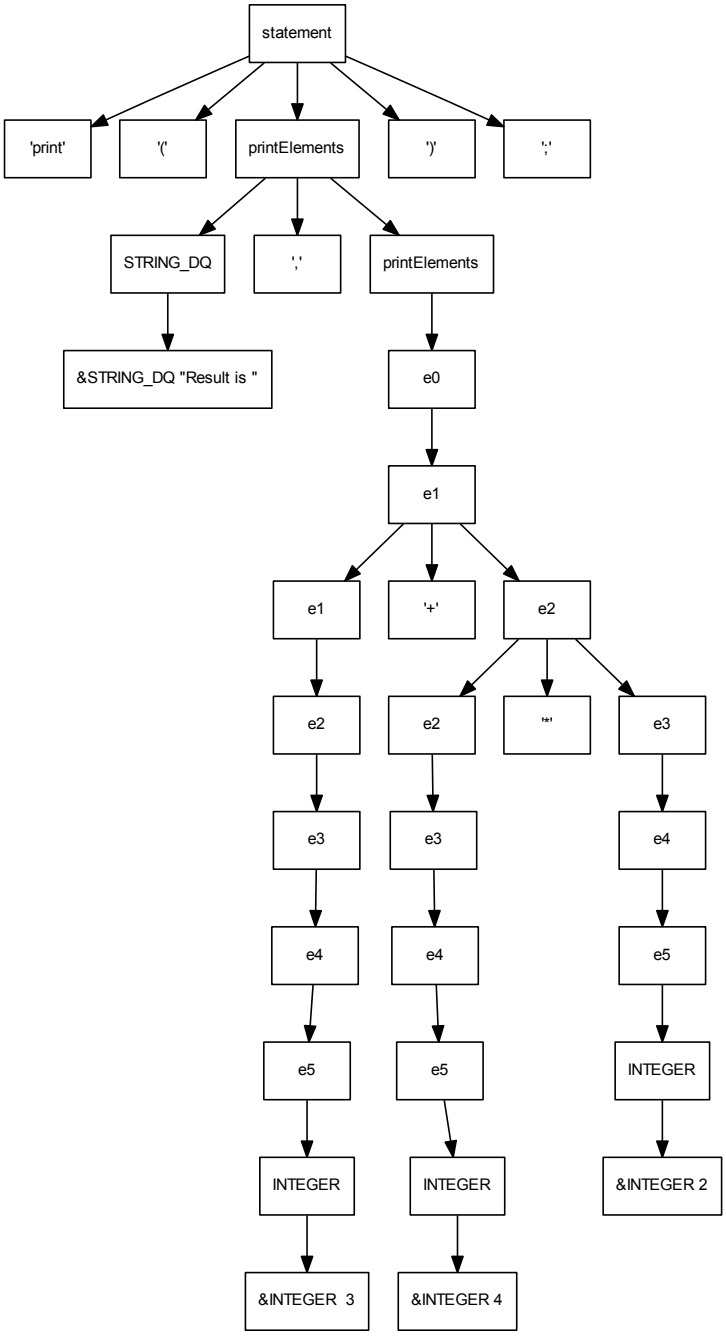
Sometimes we want to bring together nonterminal subtrees under a new parent.

4.19 GIFT applications

It is often convenient to be able to represent expression trees as being made up of operators and operands. Operators such as + and × are grammar terminals, and we can achieve this affect by promotin the operator symbols over their parent nonterminals.

Here is a first attempt, in which the `minisyntax` grammar has been modified so that every operator symbol has had a ^^ annotation applied to it.

```
statement ::= 'print' '(' printElements ')' ';' ; (* print statement *)

printElements ::= STRING_DQ |
                  STRING_DQ ',' printElements |
                  e0 | e0 ',' printElements ;

e0 ::= e1 |
       e1 '>'^^ e1 | (* Greater than *)
       e1 '<'^^ e1 | (* Less than *)
       e1 '>='^^ e1 | (* Greater than or equals*)
       e1 '<='^^ e1 | (* Less than or equals *)
       e1 '=='^^ e1 | (* Equal to *)
       e1 '!='^^ e1 ; (* Not equal to *)

e1 ::= e2 |
       e1 '+'^^ e2 | (* Add *)
       e1 '-'^^ e2 ; (* Subtract *)

e2 ::= e3 |
       e2 '*'^^ e3 | (* Multiply *)
       e2 '/'^^ e3 | (* Divide *)
       e2 '%'^^ e3 ; (* Mod *)

e3 ::= e4 |
       '+'^^ e3 | (* Posite *)
       '-'^^ e3 ; (* Negate *)

e4 ::= e5 |
       e5 '**'^^ e4 ; (* exponentiate *)

e5 ::= INTEGER | (* Integer literal *)
       '(' e1 ')'; (* Parenthesised expression *)

STRING_DQ ::= &STRING_DQ ;

INTEGER ::= &INTEGER ;
```



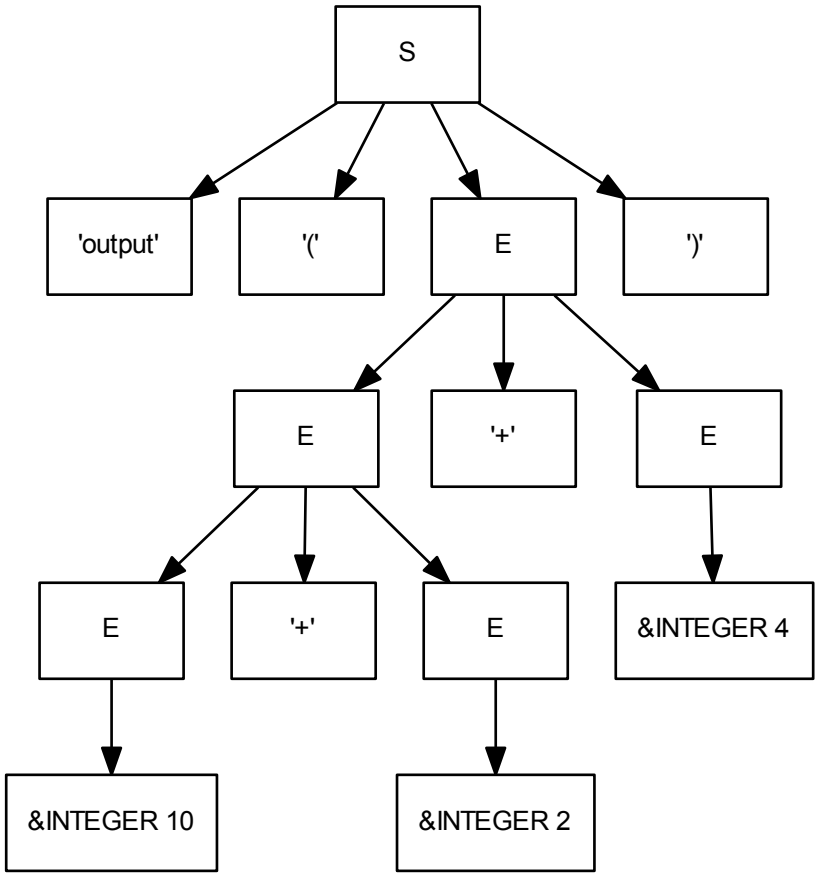

```
statement ::= 'print'^^ '('^ printElements^ ')'^ ';'^ ; (* print statement *)

printElements ::= STRING_DQ |
                  STRING_DQ ','^ printElements^ |
                  e0 | e0 ','^ printElements^^ ;
```

```
e0 ::= e1^^ |
        e1 '>'^^ e1 | (* Greater than *)
        e1 '<'^^ e1 | (* Less than *)
        e1 '>='^^ e1 | (* Greater than or equals*)
        e1 '<='^^ e1 | (* Less than or equals *)
        e1 '=='^^ e1 | (* Equal to *)
        e1 '!='^^ e1 ; (* Not equal to *)

e1 ::= e2^^ |
         e1 '+'^^ e2 | (* Add *)
         e1 '-'^^ e2 ; (* Subtract *)

e2 ::= e3^^ |
         e2 '*'^^ e3 | (* Multiply *)
         e2 '/'^^ e3 | (* Divide *)
         e2 '%'^^ e3 ; (* Mod *)

e3 ::= e4^^ |
         '+'^^ e3 | (* Posite *)
         '-'^^ e3 ; (* Negate *)

e4 ::= e5^^ |
         e5 '**'^^ e4 ; (* exponentiate *)

e5 ::= INTEGER^^ | (* Integer literal *)
         '('^ e1^^ ')'^; (* Parenthesised expression *)

STRING_DQ ::= &STRING_DQ ;

INTEGER ::= &INTEGER ;
```

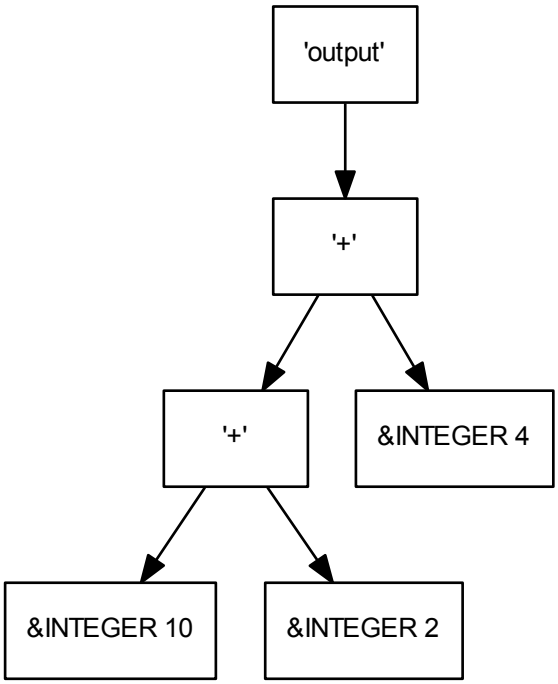
A *term* is a well formed formula (wff) is some formal language Γ. Our formal view of semantics will be a game in which terms representing program fragments and semantic entities such as the store and output will evolve through transitions. We shall move between three different ways of representing terms: (a) as the formula itself, as a derivation tree of the formula in an abstract syntax Γ', and as a fully parenthesized representation of the that derivation tree. So, for instance, if Γ has these productions

```
S ::= 'output' '(' E ')'
E ::= E '+' E
E ::= E '*' E
E ::= INTEGER
```

then the well formed formula output(10+2+4) has this derivation tree



Using GIFT annotations, we could map this to a more compact trees:



and we would then represent the derivation tree term as output(+(+(10, 2), 4))

5 Attributes

In this chapter we look at a variety of formalisms and engineering methodologies for implementing language semantics. As we have seen, we have well-developed theory and tools for syntax analysis, and for many years it has been conventional for the front end of compilers and translators to be automatically generated from concise specifications in a BNF style notation.

Semantic analysis certainly has a well-developed theory, but in contrast to syntax analysis, rather few real applications make direct use of formal approaches to semantic description. The reasons seem to be three-fold.

Firstly, a formal approach to programming is uncomfortable for many software engineers. This is a shame, because in fact the mathematics required for a user-level understanding of formal semantics is straightforward and extends only to sets, relations, rewriting as substitution and the use of logical inference. Nevertheless, it is true that very concise mathematical notation can be off-putting to the casual reader. Attempts have been made to bridge this gap by wrapping the notions and notations of formal semantics in a more programming-language like style; most notably in Peter Mosses' Action Semantics and Action Notation.

Secondly, currently used formal models of semantics do not have efficient natural implementations. One can achieve a great deal with paper-only analyses of programming language constructs, but engineers who want to get a translator running need a way to get from descriptions of programming languages to actual interpreters and compilers for their language, and in most cases need the memory consumption and speed of these to be acceptable to their users.

Thirdly, software projects rely on re-use. Constructing a fully featured programming language and its processors from scratch is a very substantial task, and most developers want to build on pre-existing tools. The most common styles of formal semantics result in specifications whose elements are richly entwined, and adding a new language feature typically requires adjustment to many existing rules. This high level of interdependence militates against a mix-and-match approach to programming language design. Of course, some aspects of programming languages (in particular the type system) are so fundamental that modifications are very likely to require changes that ripple across the whole language, but we might hope that we could hybridise most programming language features without needing an *ab initio* rewrite of their formal semantics, especially when dealing with languages that are ina subset relationship (such as C and C++, or new version 1.8 and 1.5 of Java) or when dealing with languages which are conceptually similar, such as Java and C#.

One of the goals of this book is to encourage the use of more formal approaches to language design even for small domain specific languages, so as to promote clean design and reduce the probability of implementation errors. Our approach is to 'operationalise' the mathematics of formal semantics by explaining how specifications may be directly (though inefficiently) interpreted, and by developing routes to the generation and compilation of formal descriptions that support automatic generation of more efficient translators. We begin by considering use-cases for different styles of language.

5.1 Language styles

Languages are designed for particular purposes, and the language designer often seeks conciseness of expression for their chosen domain and conceptual elegance by providing a particular set of facilities that match the application domain.

5.1.1 Data-centric languages

Our focus has mostly been on executable languages with a runtime model that includes state and control flow. However there are many languages which are in essence data description languages: we call these *data-centric* languages. Of course, data structures can display repetition (think of arrays and lists) and even conditional fields (for instance, discriminated unions in C and Pascal) and so conditonality and repetition specified by control-flow like constructs can be useful even in pure data-description languages.

Examples of domain specific data-centric languages include scene description languages such as PoV-ray, and the simulator languages used to describe electronic circuits. At their core, these languages describe graphs of objects which can then be exercised by, respectively, ray tracers and simulators.

General purpose data-centric languages based on human-readable textual descriptions have come to dominate data interchange, especially in networked and service-oriented computing. Probably the best known example is XML, which although primarily targeting document description is also widely used to describe data formats. XML is hierarchical, which means that the described structures are naturally trees, but object references in principle allow the construction of arbitrary relationships within the data structures. Languages such as JSON may be viewed as lightweight replacements for XML which are particularly easy to read and parse.

5.1.2 General purpose programming languages

A multitude of languages for general purpose programming have been developed since the 1950's. **** Todo:**

5.1.3 Domain specific languages and requirements analysis

**** Todo:**

5.2 Approaches to implementation

At some point, language processors make contact with the hardware, since the only mechanism we have for generating observable behaviour from specifications written in some notation N is to 'run' it on a real computer. We can distinguish a spectrum of implementation techniques.

At one end are simple applications which work directly with the results of syntax analysis: in general a set of derivations for our input string; most often a single derivation that has been obtained by disambiguating the results of our parser. We call these *derivation traversers*. At the other, are tools which transform the input string through a sequence of translations resulting in efficient machine code which can be subsequently loaded for execution by some processor: traditionally these tools are called *compilers*.

A complete compiler for a general purpose programing language is a challenging project, and many successful systems have instead been implemented as *preprocessors* in which a string in N is translated to a semantically-equivalent string in some language L which is usually a high level language. For instance, the first C++ compilers were implemented as a tool called Lfront which translated from C++ to C. This use case, in which $L \subset N$ is particularly common when researchers are seeking to develop new versions of existing languages. A related task is the development of translators for *mingled* languages, for instance allowing the use of SQL database language statements within an existing high level language. We can think of this as needing multiple preprocessors operating on a source syntax which is the union of the original languages, the main task being to separate out phrases belonging to each of the mingled languages, and then passing them to an appropriate conventional language processors.

In between are tools called *interpreters* which construct an internal representation from the important elements of the derivation and then traverse that form performing actions as they go. Historically, interpretation has been viewed as the poor-cousin of compilation, since the overhead of mapping language phrases to machine actions during execution means that interpretation of general purpose programs can be slow. However, in the modern era of fast processors and abundant memory, interpretive systems are often sufficiently fast, and if our interpreter is written in a high level language for which good compilers already exist, then adopting an interpretive style gives a high degree of portability without the need to maintain multiple translators for multiple architectures. For instance, the interpreted language Python has a reference implementation called CPython, the core of which is written in C and is therefore portable to any machine with a C compiler (which is practice means nearly all commercially available architectures). There is also a JPython implmentation which is implemented in Java, and thus can be compiled on any system with a Java compiler, and the resulting code run wherever there is a JVM implementation. A further advantage of interpreted systems is that they may require less memory: a stream of high level language tokens is often smaller than the equivalent compiled code, and this can be very helpful in highly memory-constrained systems: for instance the microPython interpreter is practical for systems with as little as 32K of rewritable memory.

The modern trend, led by the development of high performance Java Virtual Machines, is to produce hybrid systems which have interpreters that monitor their own execution, and which can stop and locally compile to native machine code phrases which have been executed many times. This so-called Just-In-Time (JIT) execution model has allowed JVM performance to approach that of compiled code once the interpreter has detected appropriate regions for compilation (a process referred to as 'warm up') whilst retaining code compatibility with highly portable JVM interpreters.

These styles — traverser, preprocessor, interpreter and compiler — form a loose hierarchy, since all language processors will perform some derivation tree traversal; most translators perform some normalisation actions (such as translating different kinds of loop statement into a single mormalised form) which may be viewed as pre-processing; and simple compilers could be viewed as interpreters whose actions leave behind a trace of machine-level actions to be executed.

5.2.1 Derivation traversers

There are a few applications for which the tree *is* the semantics, and no further processing is required. Consider, for instance, the task of deciding whether two student programs are identical up to variables names. The derivation tree over language tokens typically captures this information if we ignore the individual lexemes associated with identifier leaf nodes. In principle, we could output a textual form of the derivation tree and use an operating-system level utility to compare them with no other programming required.

We might be interested in software metrics of various kinds, for instance, such as the average number of statements in a method, or the maximum nesting depth of control flow statements. These sorts of applications require simple computations over the tree such as counting the number of nodes in a subtree, or counting the maximum depth of a tree. We might also want to write a tool that enforced some coding standards: for instance a software company might require all control flow statements in Java to have braces around their bodies even when they are single statements. These kinds of applications require straightforward tree traversals.

Now consider code refactoring. A common requirement is to rename all instances of a variable X, say, to Y in a program. A correct implementation must not simply change all of the X identifiers to Y because we may be using the same name for several different variables. For instance, may Javamethods operating on strings might have an argument called `string`; a refactoring centred on one of those methods must leave the others untouched. Clearly we need a *semantics-aware* replacement which only updates instances which are within the same scope region. At this point, the derivation tree is no longer sufficient in itself: we additionally need some representation of the scope semantics of our language so that we can distinguish indepndent variables that happen to have the same name.

For a data-centric language, again the tree itself might be a near-sufficient representation. The derivation for an XML description of a document contains

all of the information in the document, along with styling information, and one could build, say, a word processing application around routines which traversed the tree to compose an on-screen representation of the document, and which modified the tree in response to insertion, deletion and styling requests from a graphical user interface. The XML derivation tree is thus being directly used as the *internal form* for the word processor.

5.3 Attribute Grammars

It is natural to think of the leaves of a derivation tree as being associated with values: for instance an INTEGER token matching the string `"0123"` is associated with the value 123 and so on. When implementing arithmetic expressions, it is useful to think of these values as percolating up through the tree, being transformed by operators as we go.

Many early compilers used these sorts of ideas, and in the late 1960's Donald Knuth formalised these ideas by associating attributes with nonterminals in a grammar, such that every (a) instance in a derivation tree of some nonterminal X would have the same attribute set; (b) the values of attributes would be specified by equations; and that (c) if an attribute of X were defined in a production of X, then it must be defined in *all* productions of X.

Knuth distinguished between *inherited* and *synthesized* attributes. Conceptually, the use f inherited attributes causes information to be passed down the tree, and uses of synthesized attributes represent upwards data flow. It turns out that formally we can write equivalent specifications that use either only inherited or only synthesized attributes, but in practice it is convenient to use both. We have already seen that expression evaluation uses upwards propogation of values, and indeed expression evaluators typically use synthesized attributes. Context information, such as the declared types of variables often needs to be propogated down into sections of the tree, and inherited attributes are the appropriate means to do so.

5.3.1 The formal attribute grammar game

Let $\Gamma = (T, N, S, P)$ where T is a set of terminals, N is a set of nonterminals $(T \cap N = \emptyset)$, $S \in N$ is the start nonterminal which must not appear on any RHS (and so S must not be recursive), and$P = (T \times (N \setminus S))^*$ is a set of productions.

Each symbol $X \in V$ has a finite set $A(X)$ of attributes partitioned into two disjoint sets, synthesized attributes $A_S(X)$ and inherited attributes $A_I(X)$.

The inherited attributes of the start symbol (elements of $A_I(S)$) and the synthesized attributes of terminal symbols (elements of $A_S(t \in T)$) are preinitialised before attribute evaluation commences: they have constant values.

Annotate the CFG as follows: if Γ has m productions then let production p be

$$X_{p_0} \to x_{p_1} x_{p_2} \ldots x_{p_{n_p}}, \qquad n_p \geq 0, \qquad X_{p_0} \in N, \qquad X_{p_j} \in V,\ 1 \leq j \leq n_p$$

A semantic rule is a function f_{pja} defined for all $1 \leq p \leq m, 0 \leq j \leq n_p$; if $j = 0$ then $a \in A_S(X_{p,0})$ and if $j > 0$ then $a \in A_I(X_{p,j})$.

The functions map $V_{a_1} \times V_{a_2} \times \ldots \times V_{a_t}$ into V_a for some $t = t(p, j, a) \geq 0$

The 'meaning' of a string in $L(\Gamma)$ is the value of some distinguished attribute of S, along with any side effects of the evaluation.

5.3.2 Attribute grammars in practice

When we want to engineer a translator using attribute grammars we have to do two things: (a) consider the fragments of data that need to reside at each node (the attributes) and (b) the manner in which those attributes will be assigned values. Let us construct by example some concrete syntax for attribute grammars which incorporates the abstract syntax represented by the definitions in the previous section.

Consider a rule of the form

```
X ::= Y Z Y X.a = add(Y1.v, Y2.v) Z1.v = 0
```

The BNF syntax is as usual. The equation is written as an attribute X.a followed by a = sign and then an expression involving other attributes. The scope of an equation is just a single production which means that the only grammar elements (and thus attributes) that may be referenced in an equation are the left hand side noterminal, and the terminals and nonterminals on the right hand side. In Knuth's definition, the LHS nonterminal has suffix zero, but typically in real tools we drop the suffix and just use the nonterminal name as here. The right hand side instances are numbered in a single sequence; in tools we often maintain separate sequences for each unique nonterminal name as here.

One can tell syntactically whether an attribute is inherited or synthesized by examining the left hand side of its equation: if the LHS of an equation is an attribute of the left hand side of the rule, then in tree terms we are putting information into the parent node, and thus information is flowing up the tree and this must be a synthesized attribute. If the LHS of an equation references one of the right hand side production instances then we are putting information into one of the children nodes, and information is flowing down the tree (so this must be an inherited attribute). In the example above, X.a is synthesized and Z1.v is inherited.

It is perfectly possible to completely define a real translator or compiler using attribute grammars, and many tools exist to support this methodology. Pure attribute grammars are a declarative way of specifying language semantics using just BNF rewrite rules and equations, and it is the job of the attribute evaluator to find an efficient way to visit the tree nodes and perform the required computations.

5.3.3 Attribute grammar subclasses

A variety of attribute grammar subclasses have been defined, mostly in an attempt to ensure that equations may be evaluated in a single pass on-the-fly by near deterministic parser generators. For instance, the LR style of parsing used by Bison is bottom up, that is the derivation tree is constructed from the leaves upwards. If we are to do attribute evaluation at the same time, then we must restrict ourselves to equations that propogate upwards: hence all of the attributes must be synthesized (and equations are usually written at the end of the production to ensure that all values are available). Such attribute grammars are called *S-attributed*.

For top-down recursive descent parsers we can handle a broader class of attribute grammars. As with bottom up, the requirement is that attribute values be computable in an order which matches the construction order of the derivation tree. Such attribute grammars are called *L-attributed*: in an L-attributed grammar, in every production

$$X \to y_0 y_1 y_2 \dots y_k \dots y_n$$

every inherited attribute of y_k depends only on the attributes of $y_0 \dots y_k - 1$ and the inherited attributes of X. This definition reflects the left-to-right construction order of the derivation tree.

5.4 Semantic actions in ART

Semantic actions and attributes in ART use the parser's implementation language to model the decalarative, equational attribute grammar formalism. Back end languages for ART include C++ and Java, which are languages that do not enforce referential transparency. As a result, it is possible to write attribute grammar specifications in ART which are not equational: specifically, attributes in ART are procedural language variables to which we make assignments, and so in principle we can have several 'equations' in a rule all of which target the same attribute, which means that the value of an attribute is no longer a once-and-for-all thing, but instead may evolve during the parse.

From a formal point of view, this is very ugly. From a software engineer's perspective, it is an opportunity to introduce efficiencies. Which camp you are in rather depends on your primary concerns.

Although ART attributes are in some senses more powerful than true AG attributes, the evaluator in ART is definitely less powerful than would be required for a true AG evaluator, as we shall see.

5.5 Syntax of attributes in ART

In ART, user attributes must be declared for each nonterminal. A rule such as

```
X < value:String number:int> ::= 'x'
```

specifies that an instance of X has two attributes: one of type `String` called value and another of type `int` called number.

An action in ART is specified on the right hand side of the rule within curly braces `{` and `}`. Any syntactically and semantically valid fragment of Java may appear within the braces. It is important to understand that ART treats material within these braces as a simple string — ART does not understand the syntax or semantics of Java or any of the other backend languages, and so cannot test for errors within the string. If you write an action which is ill-formed, you will only find out when either (a) the compiler for the back end language attempts to process the output from ART or (b) when the evaluator actually runs. This can make debugging semantic actions somewhat challenging. This is in the nature of meta-programming: the ART specification is effectively a specification for a program that ART will write, so you are one step removed compared to the normal software engineering process.

So, for instance,

```
X < value:String number:int> ::= 'x' { X.value = 3; }
```

is a valid ART specification which generates a compile-time-invalid piece of Java because the expression `3` is not type compatible with the attribute `value` which is of type `String`. Similarly, if an attribute was an array and an action tried to acccess an element which was out or range, then the error in the action would not be picked up by either ART or the Java compiler, but instead generate a run-time exception.

5.5.1 Special attributes in ART

ART recognises two special attribute names: `leftExtent` and `rightExtent`. When the user declares attributes with those names and type `int` they are treated just as for ordinary attributes except that the parser initialises them with the start and end positions of the string matched by that nonterminal instance. As a result, one would not usually expect to find attribute equations in the actions that had either `leftExtent` or `rightExtent` on their left hand sides.

The use of these special attributes allows us to create attributes at the lowest level of the tree. In ART, attributes cannot be defined for builtin terminals or terminals that are created literally. However, we can wrap an instance of a terminal in a nonterminal, and then use these special attributes to extract the substring matched by some terminal. For instance, here is the definition of a nonterminal `INTEGER` which uses the `&INTEGER` lexical builtin matcher to match a substring, and then extracts a value using the lextExtent and rightExtent attributes

```
INTEGER <leftExtent:int rightExtent:int v:int> ::= &INTEGER
{INTEGER.v = artLexemeAsInteger(INTEGER.leftExtent, INTEGER.rightExtent);}
```

ART provides a set of methods for converting substrings of the input to values: `artLexemeAsInteger()`, `artLexemeAsDouble()` and so on.

5.6 Accessing user written code from actions in ART generated parsers

It would be cumbersome to have to put the entire functionality of a translator into semantic actions. Instead, we would like to parcel complex operations up into functions or class methods, and simply call them from the semantic actions.

Back end languages for ART vary in their requirements, but for the Java backend we can imagine both wanting to access objects of classes outside of ART's generated parser class, and also the addition of members to the ART generated parser class itself. ART provides two mechanisms to help.

The `prelude{...}` declaration specifies that the material within the braces be copied into the generated code at the top of the file. This enables us to add, for instance, `import` declarations to the Java generated parser, and thus to access objects and static methods of other classes within our semantic actions.

The `support{...}` declaration specifies that material within the braces be copied into the generated code within the ART generated parser class itself, allowing us to declare methods and variables which are visible throughout the generated parser's actions.

5.7 A naïve model of attribute evaluation

How would we build a (not very efficient) general attribute evaluator for ART? Let us begin by giving each node in the tree a unique *instance* number. Then each attribute may be uniquely named as (instance number, name). Make a set U which will contain the subset of attributes which are presently undefined. Make a map V from attribute names to attribute values which is initially empty.

We begin by handling the special attributes `leftExtent` and `rightExtent`. For each attribute in $u_i \in U$ with a name of the form (k, leftExtent) or (k, rightExtent), remove u_i from U and add an element to map V which maps (k, left(right)Extent) to the first(last) index position of the substring matched by instance k.

Now, while U is nonempty, traverse the entire tree and examine, all of the equations for productions used in the derivation sequence and perform these actions

1. If the attribute on the LHS of the equation is not in U, then continue. (The attribute has already been computed.)

2. If the attribute on the LHS is in U and any attribute on the RHS is in U, then continue. (The attribute is not ready to be computed.)

3. If the attribute (k, n) on the LHS is in U and no attribute on the RHS is in U, remove (k, n) from U and add an element to V mapping (k, n) to the result of computing the right hand side expression.

Recall that a well-formed attribute grammar must be (a) non-circular and (b) must have an equation in every production defining the value of all attributes in its RHS nonterminal. As a result, there must be some ordering over the equations that allows them to be resolved. This algorithm finds an ordering by brute force: it simply continually traverses the tree looking for so-far undefined LHS attributes whose right hand side attributes *are* defined, at which point it computes the new value and removes the attribute from the undefined set.

This algorithm is simple, but inefficient because in worst case we might only be able to compute one equation per entire pass of the tree. In practice, real general attribute evaluators perform *dependency analysis* on the equations to find much more efficient schedules.

5.8 The representation of attributes within ART generated parsers

ART provides an abstract class `ARTGLLAttributeBlock`. Inside ART, nonterminals are named $M.N$ where M is a module name and N is the name of a nonterminal defined in module M. The default module name is `ART` so in specifications with explicit module handing, a nonterminal called `X` by the user is called `ART_X` internally.

For each attributed nonterminal $M.X$, ART creates a concrete subclass of `ARTGLLAttributeBlock` called `ART_AT_M_N`, so for instance the ART rule

```
X <p: int q:double> ::= 'x'
```

in module M generates the class

```
1  public static class ART_AT_M_X extends ART_GLLAttributeBlock {
2    protected double q;
3    protected int p;
4  }
```

A separate instance of this class is created for each instance of nonterminal $M.X$ in the derivation. Each instance effectively has two names within the attribute evaluator: $M.X$ for left hand side attributes and $M.X_k$ for right hand side instances, where k is an integer. When we write an action like `M_X.v = 3;` we mean, locate the attribute block for my left hand side which is called `M_X` and then access the field called `v`. When we write an action like `M_X.v = M_X1.v` we are asking for the `v` value from the attribute block for the first instance of $M.X$ on the right hand side of our rule to be copied to the left hand side instance.

5.9 The ART RD attribute evaluator

Whilst we could implement an attribute evaluator based on the general model above, it would be inefficient. Instead we implement *syntax directed translation.*

Rather than seeking a schedule which resolves all of the data dependencies in the attribute grammar, we instead assert a particular schedule and require the writer of the attribute grammar to not write equations which violate its constraints. We say that an AG specification is *admissable* if it may be computed by our predefined schedule, and *inadmissable* otherwise.

The ART attribute evaluator correctly evaluates *L-attributed* grammars. In an L-attributed grammar, in every production

$$X \rightarrow y_0 y_1 y_2 \dots y_k \dots y_n$$

every inherited attribute of y_k depends only on the attributes of $y_0 \dots y_k - 1$ and the inherited attributes of X.

There is quite a strong parallel here with parsing: a general parsing algorithm such as GLL (the algorithm ART implements) can handle any specification, but with the risk of poor performance on some grammars. A non-backtracking Recursive Descent parser, on the other hand, can only handle deterministic LL(1) grammars (or ordered grammars which are nearly LL(1)) but will run in linear time.

Our evaluator is essentially a recursive descent evaluator. It will only traverse the tree once. As long as the equations may be fully resolved in a single pass, all will be well. The ART evaluator is limited to attribute schemes that are essentially the L-attributed schemes. However, we can do a lot with such schemes, and the evaluation time is linear in the size of the tree.

In detail, the ART evaluator recurses over the datastructure constructed by the GLL parser. This is not a single derivation, but a (potentially infinite) set of derivation trees embedded within a structure called a Shared Packed Parse Forest (SPPF). However, prior to starting the evaluator, we will have marked some parts of the SPPF as suppressed, and some parts as selected, and the net effect is that the evaluator can assume that it is recursing over a single derivation tree.

As the evaluator enters a node labeled X, it creates the attribute block for each nonterminal child below it (corresponding to the nonterminal instances in the derivation step $X \Rightarrow \alpha$ encoded in this height-1 sub-tree). These newly-created attribute blocks are assigned to variables with names like $Y1$ and $Z2$ corresponding to the first and second instances of Y and Z in some production like `X::= Y Z Z`

The evaluator is a nest of functions, one for each nonterminal, in a way that is isomorphic with our OSBTRD parser functions. In our example above, the evaluator function for X will be called and make attribute blocks for the children $Y1$, $Z1$ and $Z2$. It will then call the evaluator function for Y passing block $Y1$ as an argument. The evaluator functions all take a single parameter block whise name is the same as that of the nonterminal. By this means, the block for Y allocated in X is called $Y1$ in the evaluator for X but called Y in the evaluator for Y.

Just like an RD parser, the evaluator functions call each other in the same order as instances are encountered within the grammar, and the semantic actions are inserted directly into the evaluator functions.

5.10 Higher order attributes

There is a well-developed theory of higher order attributes, which are attributes that represent parts of derivation trees rather than simple values. There are essentially two classes of HO attributes: attributes which capture part of an existing derivation tree, and attributes which contain new pieces of tree which can be used to extend a derivation tree from the parser. In ART, we support the former, but not yet the latter. This means that the shape and labelling of a derivation tree in ART cannot be modified by an attribute grammar: only the attribute values associated with tree nodes can be modified. In a later section we shall describe ART's GIFT operators which do allow trees to be modified. In the present implementation, the attribute evaluator works on the full derivation and completes evaluation before the GIFT rewriter changes the tree.

ART's notion of higher order attributes requires only two things: a way of marking tree nodes as having a higher-order attribute associated with them, and a way to allow the user to activate the evaluator function under the control of semantic actions.

The first is achieved by adding an annotation `<` to any right hand side instance of a nonterminal in a grammar rule. The second is achieved by providing a method `artEvaluate()` which takes as an argument a higher order attribute.

When the evaluator function arrives at a node with a higher-order attribute, it does not descend into it (although it will construct the attribute block for it). The idea is that instead of automatically evaluating a subtree, the outer evaluator will ignore it, but the user may specify semantic actions to trigger its evaluation on demand.

Why is this useful? Well one application is to allow our recursive evaluator to interpret flow-control constructs. Consider an `if` statement. It comprises a predicate, and a statement which is only to be executed if the predicate is true. We can specify this as follows:

```
ifStatement ::= 'if' e0 'then' statement<
        { if (e01.v != 0) artEvaluate(ifStatement.statement1, statement1); } ;
```

The `<` character after the instance of statement creates a higher order attribute called `statement1` in the attribute block for `ifStatement`. ART will also have created an attribute block called `statement1`. The evaluator will automatically descend into the subtree for `e0`, but will not descend into the subtree for `statement`: instead it loads a refernce to the subtree for this instance of `statement` into the attribute statement1 in `ifStatement`.

In the action, we look at the result that was computed within `e0`, and if it is not zero (signifying false) we call the evaluator on the the subtree root node held in the attribute `ifStatement.statement1` and pass in parameter block `statement1`. This effectively emulates what would have happened automatically if we had left off the `<` annotation, but under the control of the result of `e0`. Hence the evaluation order of the tree is being dictated by the attributes and semantic actions themselves! This is exactly the sense in which our attributes are higher order. However, we can only traverse bits of tree that were built by

the parser: we cannot make new tree elements and call the evaluator on them. Full higher order attributes do allow that. We call our restricted form *delayed* attributes so as to distinguish them from the more general technique.

We can use these delayed attributes to build interpreters for languages with conditionals, loops and function calls, as we shall see in the laboratory exercises.

6 Pragmatics

In this chapter we look at the origins and basic building blocks of programming languages.

6.1 Icons, letters and phrases

The origins of human communication are necessarily obscure, but the archaeological record leaves us some clues. We know that bipedal apes emerged around seven million years ago; that 1.5–2 million years ago early hominids such as *Homo Habilis* used tools to scavenge but were often prey to large animals; and that by the time of *Homo Ergaster* and its descendant species (around 1–1.5 million years ago) there is evidence of the use of fire and a physiology that might allow some form of speech. By 70–80,000 years ago there is clear evidence that the hunted had turned hunter, with group hunting of large game, which must surely have required coordination, planning and communication.

Many hundreds of sites containing cave paintings are known, the oldest of which are believed to be around 40,000 years old and clearly represent some form of *non-transient* communication. Images of animals abound, and images of humans are rare apart from hand stencils (pigment blown over a hand). In a few cases, hunting scenes are clearly represented. This painting, for instance, is from the Bhimbetka rock shelters in India; and shows mesolithic hunters using bows and arrows, perhaps 6–10,000 years ago.

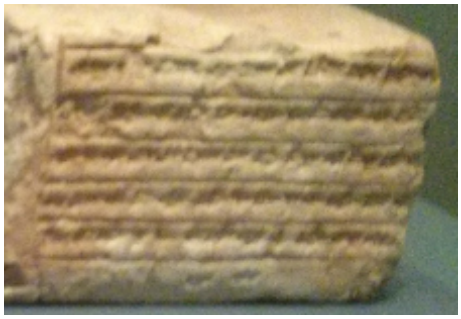
We cannot know whether these are records of successful campaigns, or perhaps instructional in the sense that they played a part in preparing for a hunt and training new members of the group. To be a little whimsical; it would be pleasing if one could establish that some paintings formed a do-this then

do-that progression, as this would surely constitute use of a recorded sequence of commands; the world's first program.

The transition to text based systems is also, naturally, obscure. Neolithic objects from Jiahu in China incorporating symbols have been dated to 6,500 BCE. The so-called Vinča symbols on the Tărtăria tablets found in Romania date to around 5,300 BCE (left, below). The Greek Displio tablet (`journals.uair.arizona.edu/index.php/radiocarbon/article/view/17456` is a wooden piece which has been carbon-14 dated to 5260 ± 40 BCE and includes symbols such as a triangle with a dot in it, and forms similar to our letters E, t, v and L (right, below). Scholars in this area call these sorts of symbols, which often appear in isloation as *proto-writing*.

In fully formed writing, strings of symbols (called *graphemes*) represent spoken sentences. Some writing systems are *logographic* in which individual concepts are represented by graphemes called *logograms* (small pictorial symbols representing, say, a class of objects such as house); modern examples include Chinese characters and Japanese Kanji. More commonly individual (or short sequences of) graphemes represent the primitive sounds or *phonemes* of spoken language, forming an *alphabet*. A string of alphabetic graphemes 'spell' out the sound of a spoken word.

Egyptian hieroglyphs combine both elements: of the 1,000 or so known hieroglyphics there are symbols corresponding to consonants as well as symbols that in themselves represent, for instance, sun. Some symbols, such as that for house, may be logographic in some contexts and but stand for a single consonant in other contexts. A vertical bar under the symbol indicates the logographic use. Hieroglyphics and Samarian cuneiform (which was made with cut reeds on clay tablets) are candidates for the earliest fully formed writing system dating back to about 3,000 BCE, but they involve large numbers of signs: cuneiform uses around 800 patterns which represent individual symbols.



Alphabets instead encode the *phonemes*, the individual sounds from which speech is composed which requires far fewer symbols, and in which syllables are represented by short strings of alphabetic letters.

Most Western scripts descend from the Phoenician alphabet which was in use by 1000 BCE, though that script only directly represents consonants (and is thus sometimes called an *abjad* rather than an alphabet). Txt wrttn tht wy cn b prfctl lgbl. Each Phoenecian letter was derived from the shape of the sign for some common syllable which started with that letter. This arrow head from 1100 BCE and now in the British Museum is inscribed *arrow of Ada, son of Bala.*

Extra characters representing vowels were later added by the Greeks, and that developed into the Latin alphabet in which this text was written.

Software languages are primarily alphabetic, but there have been many attempts to construct primarily logographic systems, such as the Scratch and Lego Wedo programming environments for neophyte programmers. *Graphical User Interfaces* (GUIs) are heavily oriented towards logographic representations, making extensive use of mnemonic icons instead of text labels: an immediate advantage is interationalisation since pictorial representations are independent of spoken language; immediate disadvantages include the proliferation of sometimes hard to understand images. GUI's increasingly use touch gestures as inputs (such as pinch to zoom) and in future GUI input graphemes might include future facial expressions and hand gestures. For one-off actions, clicking a button labeled with an icon can be appropriate, but as soon as we want to record sequences of actions then logographic specification can become unwieldy. Most *programs* are intended to be read as well as executed and a textual representation is therefore convenient, so shall focus on systems where the graphemes are limited to characters input via a keyboard.

Even in conventional languages, we extend the Latin alphabet with standard mathematical symbols such as $>$ and $=$. Mathematics publications use many more such symbols and the meaning associated with a symbol, say ϵ will commonly vary with sub-discipline. There are programming languages which use large alphabet, most notably APL `en.wikipedia.org/wiki/APL_syntax_and_symbols` which pre-defines a very large number of operator symbols. For instance, this APL expression from the Rosetta code website (`rosettacode.org/wiki/Greatest_common_divisor#APL`) is an APL expression which uses Euclid's algorithm to compute the GCD of 49,865 and 69,811 which is 9.972.

```
⌈/(^/0=A∘.|X)/A←⍳⌊/X←49865 69811
```

At the other extreme, languages in the Lisp family tend to be composed mostly of identifiers composed from letters: a recursive version of GCD in Scheme (also from the Rosetta code site) is:

```
(define (gcd a b)
  (if (= b 0)
      a
      (gcd b (modulo a b))))
```

The design of a comfortable syntax is unlikely to ever be formalised: the tension between conciseness and readability means that it is very hard to find a form that suits all programmers. Some languages even provide alternative forms. For instance, in Algol-68 our iterative implementation of Euclid's algorithm may be written as

```
ref int a = loc int;
ref int b = loc int;

 while a != b do
   if a > b
     a := a - b
   else
     b := b - a
   fi
  od
```

or as

```
ref int a = loc int;
ref int b = loc int;

 while a != b (
   ( a > b |
     a := a - b |
     b := b - a
   )
  )
```

in which the if statement is written in a concise bracketed form. Programs may freely intermix if statements written in the two styles (although each individual if statement must be completely in one of these styles).

Another interesting approach is represented by Donald Knuth's *literate programming* system [**?**], which encourages the construction of programs as literary objects which allow a narrative to be built in a way that is independent of the conventions of any particular programming language and which allows mathematical notation; these specifications may then be converted to compilable programs and to documentation. In some of his writings, Knuth discusses this approach as a software engineering paradigm:

> I had the feeling that top-down and bottom-up were opposing methodologies: one more suitable for program exposition and the other

> more suitable for program creation. But after gaining experience with WEB, I have come to realize that there is no need to choose once and for all between top-down and bottom-up, because a program is best thought of as a web instead of a tree. A hierarchical structure is present, but the most important thing about a program is its structural relationships. A complex piece of software consists of simple parts and simple relations between those parts; the programmer's task is to state those parts and those relationships, in whatever order is best for human comprehension not in some rigidly determined order like top-down or bottom-up. [**?**]

We shall explore more deeply the design and implementation aspects of syntax in the next chapter. Our focus here is on semantics, and we need to first look at the semantic components that are found in typical programming languages.

6.2 Semantics at machine level

At the raw machine code level, we have a store comprising a fixed set of cells, each of which has a unique index number called its *address* and a fixed set of values which may be assigned to it. On a modern computer, this set of values is almost always the set of binary digit strings of length 8. There is no typing information associated with these cells: the contents could be a small integer, part of a high precision floating point number, part of a character string, or part of an instruction[1].

The contents of memory cells are interpreted under the control of a special variable called the *program counter* which holds an address P. The behaviour of the computer reduces to an sequence of *fetch-execute* cycles.

During a fetch, an instruction is fetched from P, and the program counter is updated to hold the address of the first store location P' past that instruction. This updating ensure that normally the computer executes store-contiguous instructions in sequence.

During the execute part of the cycle, the instruction's bit-pattern is interpreted as, for instance an addition of operands held at two different store locations with the result being written to a third. Other instruction forms might generate output, on some specified channel, or read input from a keyboard.

Some instruction modify the flow of control by, during the execute part, writing some new value T called a *branch target* to the program counter. The next fetch cycle will then read an instruction starting at T instead of that at P', and the effect is that the program will branch rather than executing sequentially. Typically branches are *conditional*: the loading of the the program counter with

[1]There have been experimantal machines which associated a tag with the data, thus allowing some type checking in hardware, the most well known example of which is the ICL 2900 which requireded a 64-bit descriptor to be associated with some memory accesses. The contents of the descriptor could specify, for instance, the size of an operand and whether it was code or data

T depends on whether, say, the value of a store location is zero. In this way, the program can include data dependent actions.

Since store locations are not tagged with any sort of type information, each location is simply a bit string, and will be interpreted according to context. If, for instance, the program counter is accidentally set to an address in the middle of some data, then the computer will still attempt to interpret those data bit strings as instructions, with (in general) unpredictable effects. On the other hand, a von Neumann computer exploits this property to allow programs to be loaded from storage as data, and then executed by passing control to their start address A which simply requires A to be loaded into the program counter.

Although cell contents are not tagged, the computer does impose a notion of type onto those bit patterns, in that particular machine instructions expect particular types of operands. For instance, a modern machine will have separate instructions to add 32 bit integers and 64 bit floating point numbers called, say ADD32 and FADD64. The fetch cycle will collect 32-bit long operands for the ADD32 instruction and 64-bit long operands for the FADD instruction. If an integer add specifies operands that have most recently been assigned the results of a floating point operation, then we have a typing error.

Hardware instructions rarely allow heterogeneous operands: for instance adding an integer to a floating point number. Instead, conversion instructions are supplied which directly implement semantics-preserving conversions, such as converting the floating point number 3.0 to the integer 3. These conversions are called *casts* and usually include some 'sensible' non-semantics preserving transformations. In the case of floating to integer conversions, the detailed implementation of the instruction will govern whether 3.6 is converted to integer 3 by truncation, or integer 4 by rounding.

A particularly difficult aspect of cast operations arises when the range of the target type is less than that of the input type, as in the integer cast above. Programmers are used to thinking about the decimal places being lost on conversion but sometimes caught out by the inability of, say, a 32-bit integer to directly represent even the integer elements of the range of a 64-bit float; that is an overflow on cast. Probably the most well known failure of this type is the destruction of the prototype Ariane 5 rocket in 1996. This flight used hardware and software that had operated successfully on earlier Ariane 4 flights, but the new vehicle's flight path included a higher horizontal acceleration component. The software performed a 64-bit floating to 16-bit integer conversion, and on the maiden Ariane 5 flight this led to a 16-bit overflow around 50 seconds after lunch. The autopilot responded to the rsulting sign change in the acceleration vector and attempted to correct for it by making a large adjustment to the booster and main engine nozzles which caused the vehicle to disintegrate. The estimated one-off financial loss from this bug was around one third of a billion dollars.

6.2.1 Translation to machine level

The set of instruction available on a typical modern computer will include basic arithmetic and logic operations, casts, copy instructions and flow control

including conditional branches and function calls. The raw datatypes encoded into these instructions semantics will typically include signed and unsigned integers of various sizes, characters, and floating point numbers of various sizes.

Programming languages, even very simple ones such as assembly languages, then additionally offer identifiers which may bind to store addresses and values. Whatever the level of the programming language, all of the semantic machinery of that languages must have translations down into the machine code level if a program is to be executed; the purpose of a formal semantics is ultimately to specify those translations in an unambiguous way and thus aid in the avoidance of problems such as the Ariane 5 failure.

Before looking at a taxonomy of programming language semantic features, we shall produce a concrete example of a translation from our high level IL description of Euclid's algorithm into a low level machine code.

We shall use eight bit integers, which restricts our algorithm to finding the GCD of two numbers between 0 and 255. Our store will be large enough to hold the program and the program variables; it will turn out that a 256 location store suffices, so our program counter need be only 8 bits wide.

Here is our GCD implementation again:

```
1  int a, b;
2
3  a := input(); b := input();
4
5   while a != b
6     if a > b
7       a := a − b;
8     else
9       b := b − a;
10
11 output(a);
```

The only arithmetic operation required is subtraction. It will turn out that we shall use three kinds of branch instruction, and we shall also need input and output instructions. Here is a table of instructions for plausible although non-existent computer architecture.

op	opcode	operands	length	Effect
exit	0	0	1	return from program
inp i	1	1	2	$\sigma_i \leftarrow I$
outp i	2	1	2	$P \leftarrow [i : P]$
sub8 i j k	3	3	4	$\sigma_i \leftarrow \sigma_j -_8 \sigma_k$
bra t	15	1	2	$C \leftarrow t$
beq i t	16	2	3	$i = 0 \Rightarrow C \leftarrow t$
ble i t	17	2	3	$i \leq 0 \Rightarrow C \leftarrow t$

The first column gives a human-readable name to each instruction; the second column gives the *opcode* for that instruction, that is the unique 8 bit-string that

encodes the instruction represented as a decimal number. The third and fourth columns are the number of operands and the overall length of the instruction, respectively, and the fifth column gives the semantics. We use the notationσ_k to represent the bit string in the store indexed by address k, I to represent the input, P the output and C the program counter.

Note that type information is effectively encoded into the instruction: we are using a subtract operation called sub8 represented as $-_8$ in the semantics column.

We shall now give human readable, and then machine code level translations of the IL program into instructions for this hypothetical machine. Conventionally, machine level programming languages (called assembly languges) allow programmer to write programs using syntax like that in the first column above, and then translate those lines into the binary representation. It is an easy, though rather clerical and error prone, task to do this by hand; usually a program called an *assembler* is used.

The assembler reads the program sequentially, and maintains an *assembly pointer* A. Typically the machine code is assembled into an array representing the memory of the computer at run time. In addition to instructions like those above, the assembler will provide directives for, for instance, reserving and initialising memory. The assembler will also provide *labels*. Syntactically, a label is usually an alphanumeric identifier followed by a colon; when the assembler encounters a label it binds the current value of A to that label.

Here is an assembly program corresponding to our GCD IL program. The IL program lines have been appended as comments so that the relationship between machine instructions and IL statements is clear. We begin by reserving space (lines 3–5) for our variables using the assembler directive var k which reserves k locations for later use.

```
1  var 2 // leave unused space for two byte variable
2  // int a, b;
3  a: var 1 // reserve one byte for a
4  b: var 1 // reserve one byte for b
5  tmp: var 1 // reserve one byte for internal temporary variable
6
7  var 4 // leave four bytes of unused space
8
9  // start of code
10
11 inp a // a := input();
12 inp b // b := input();
13
14 do1: // while a != b
15 sub8 tmp, a, b
16 beq tmp od1
17
18 if2: // if a > b
```

```
19 sub8 tmp, a, b
20 ble tmp else2
21
22 sub a, a, b //a := a − b;
23 bra fi2
24
25 else2: // else
26 sub8 b, b, a // b := b − a;
27
28 fi2:
29 bra do1:
30
31 od1:
32
33 outp a // output(a);
34 exit
```

The executable code begins at line 11. We first use the inp instruction to load variables a and b. Note how the location of these variables has been bound to the labels a and b on lines 3 and 3, and the way in which the labels are being used to provide operands to the inp instruction on lines 11 and 12.

We use data labels as operand addresses, and we use code labels as jump targets. Control flow constructs such as while ... do loops and if ... then .. else constructs. We shall use the convention that each control construct in a program will be given a number, and we shall use labels with names like if3 to mark the first instruction in the third construct and fi3 to label the *successor* to the last instruction in the third construct. We shall use if ... fi around if statements and do ... od around while do statements.

This next listing expands the assembly language program by preceding each line with the output that the assembler produces. The concatenation of these numbers forms the initial state of the store for th erunning program.

```
1  0: − var 2 // leave unused space for two byte variable
2  // int a, b;
3  2: 0 − a: var 1 // a is at location 2
4  3: 0 − b: var 1 // b is at location 3
5  4: 0 − tmp: var 1 // tmp is at location 4
6
7  4: − var 4 // leave four bytes of unused space
8
9  // entry point for code is location 8
10 8: 1 2 − inp a // a := input();
11 10: 1 3 − inp b // b := input();
12
13 12 − do1: // while a != b
14 12: 3 4 2 3 − sub8 tmp, a, b
15 16: 16 4 38 − beq tmp od1
```

```
16
17 19 – if2: // if a > b
18 19: 3 4 2 3 – sub tmp, a, b
19 23: 17 4 32 – ble tmp else2
20
21 26: 3 2 2 3 – sub8 a, a, b //a := a – b;
22 30: 15 36 – bra fi2
23
24 32: else2: // else
25 32: 3 3 3 2 – sub8 b, b, a // b := b – a;
26
27 36: fi2:
28 36: 15 12 bra do1:
29
30 38: od1:
31
32 38: 2 2 – outp a // output(a);
33 40: 0 – exit
```

So this sequence of numbers, then, is our program:

0,0,0,0,0,0,0,0,0,1,2,1,3,3,4,2,3,16,4,38,3,4,2,3,17,4,32,3,2,2,3,15,36,3,3,3,2,15,12,2,2,0

6.3 The semantic facets of programming languages

We shall group our discussion of semantic components of high level languages into five *facets*: (a) values, types and expressions; (b) storage, assignment and commands; (c) identifiers, bindings and scope; (d) control flow and (e) abstraction mechanisms, both procedural and data.

6.3.1 Values, types and expressions

The ultimate purpose of a program is to compute *values*; those values might be numeric solutions to equations, textual outputs, visual effects on a screen or movements of a robot arm. At machine level, all of these correspond to patterns of binary digits in memory, but programming languages provide a range of abstractions which enable us to reason more effectively about program execution. Grouping the available values by class of abstraction naturally leads to the notion of *type*.

6.3.2 Storage, assignment and commands

In a von Neumann view of computing, values are stored in reusable cells, and in that world, storage is also fundamental. Most program texts are dominated by identifiers which stand for, for instance, constant values, locations in store,

6.3.3 Identifiers, scope and binding

Nested scope rules

6.3.4 Control flow

D-structures
- Concurrency
- Jumps
- Exceptions

6.3.5 Procedural and data abstraction

Procedures
- Higher order functions
- Abstract data types, classes and packages
- Generics

6.4 Interpretation, compilation and runtime rework

6.5 Four early language traditions

6.5.1 FORTRAN – numeric processing and portable programs

6.5.2 COBOL – data processing

6.5.3 LISP – the accidental language

6.5.4 Algol – user defined data and algorithmic elegance

6.6 New ideas

6.6.1 Programming in the large

6.6.2 Object orientation

6.6.3 Concurrency

6.6.4 Generics, and types as values

6.7 General purpose and domain specific software languages

6.8 The music domain

In this section we lay the foundations for project work in music by looking at elementary aspects of western music, and exploring the capabilities of Java's built in synthesizer. Our goal is to build a domain specific language which allows convenient specification, performance, and display of musical pieces.

When working with the Java sound API, please use headphones or earbuds so as not to disturb other people in the lab.

6.8.1 Musical instruments

Broadly speaking, music is a form of structured sound. Sound itself is our perception of vibrations in the air which create sympathetic vibrations of our ear drums, and via the workings of the inner ear into changes in brain activity. A young human can perceive frequencies between about 20 and 20,000 cycles per second (written 20-20kHz) though maximum sensitivity is between 2kHz and 5kHz. We perceive different frequencies as different pitches: an increase or decrease in frequency is perceived as an increase or decrease in pitch.

A musical instrument is a device for creating structured sound. Physical objects display frequencies at which they 'want' to vibrate: we call these their resonant frequencies. If you have ever pushed somebody on a swing you will understand resonance: the swing has a frequency at which it naturally arcs back and forth, and if you give a little push just at the top of the arc then you can maintain steady smooth motion with little effort. If on the other hand you shove the swing before it reaches the high point, then the motion can become very irregular, and even cause the person on the swing to be thrown off. It turns out that you can change the resonant frequency of a swing by changing the length of the ropes holding the seat: longer ropes give a slower swing frequency.

It is a general rule that large physical objects resonate at lower frequencies than small physical objects, and that if you want to keep an object vibrating whilst using as little energy as possible, then you should stimulate it at its resonant frequency. This is, perhaps, why tiny humming birds' wings move so quickly that we perceive the disruption in the air as a hum, whilst an albatross with a 3m wingspan beats its wings slowly.

If we tension a string and pluck it, then it will vibrate at its resonant frequency. If we shorten the string, or increase its tension. or replace it with a lighter string, then the resonant frequency will increase. These are the principles behind stringed instruments including guitars, violins and pianos.

If we take an open bottle and blow across the top of it then the air in it will resonate. By using a smaller bottle (or perhaps two identical bottles with one half full of water) we can try smaller mass of air which will resonate at a higher frequency. This is the operating principle of woodwind instruments.

There are many variations on these basic themes: strings may be plucked, hit with hammers or excited with a bow. Air resonators may be fed with pumped air (as in a pipe organ) or blown into; their resonant frequencies may be modified by opening and closing valves between air spaces or by deforming the resonator, as in a trombone. For stringed instruments in particular, it is common to provide a coupled mass of air in a hollow *sound box* that will resonate in sympathy with the primary resonator, strengthening the amplitude of the air vibrations. Some modern instruments (such as the electric guitar) directly convert resonator vibrations into electrical signals which can be amplified, transmitted long distances and fed to a loudspeaker; the microphone is a general device for converting air pressure waves into an electrical signal.

Even the highest perceivable sound frequency is very low compared to the instruction execution frequencies of current computers. At 20kHz, each sound cycle lasts for $50\mu s$. A modern desktop processor can achieve instruction execution frequencies of around 2×10^9 Hz, and can thus execute around 100,000 instructions during each audio cycle. This makes it feasible to use software to generate quite complicated musical waveforms in real time. Such a device is called a digital music synthesizer, and the standard Java distribution ships with libraries for music synthesis.

Ensuring the correct synchronisation between multiple synthesized instruments and input devices such as keyboards requires careful coding and protocol design: in 1982 the *Musical Instrument Digital Interface* standard was published, and this has become a very broadly implemented system for controlling musical instruments. The Java distribution contains classes which may be used with MIDI controllers (such as keyboards) to make entirely electronic music. We shall give examples of the use of this library; and we shall write a very simple Domain Specific Language is to give the non-Java programmer access to these facilities using a small self-contained notation.

6.8.2 The perception of pitch

When presented with a complex repetitive waveform, the ear resolves it into multiple pitches, that is we can perceive the separate frequencies individually. In this respect, the ear is fundamentally different to the eye which 'averages' frequencies: when presented with a patch of green light overlaying a patch of red light, we perceive yellow. The separation of a waveform into its constituent frequencies is called Fourier Analysis: the ear effectively performs Fourier Analysis whereas the eye merges frequencies.

As we increase frequency, we hear an ascending pitch. Interestingly, and fundamentally, the ear perceives frequencies which are integer multiples of one another as 'the same but different' and nearly all music systems use this observation to split the frequency spectrum up into a sequence of 'octaves': one octave is the range of musical pitches between some frequency f and the frequency $2f$. This perception of frequency doubling naturally leads to a log-style description of pitch.

Some instruments offer a continuous range of frequencies (examples include the Theramin, fretless stringed instruments and the human voice) but most instruments provide a finite set of discrete pitches which may be based on individual tuned resonators (like the strings of a piano) or by discrete adjustments of an otherwise-continuous resonator (like the frets used in a guitar). When using a computer to generate audio waveforms, there are no constraints at all, but in practice we often use the computer to make sounds like traditional instruments.

6.8.3 The physics and psychology of pitch

Music appreciation is highly culturally conditioned, and perceptions of 'satisfying' music vary geographically and over time. As computer scientists, we understand how to systematise and implement behaviours which 'make sense'

to our users by hiding low-level complexity and only providing control over high level concepts: for instance, when we connect our laptops to a network, we do not expect users to understand the complexities of the network protocol, or even to know the numeric address of the machine they are connecting to. Similarly, musicians organise the continuum of available frequencies into a set of conventions which 'feel right'; and growing up within a particular musical culture these conventions we may come to feel in some sense natural and fundamental. However, we must never lose sight of the fact that alternative conventions may be just as natural to people growing up in other cultures.

Most western music is organised around the twelve-tone-equal-temperament scale (12-TET) in which an octave is divided into twelve discrete frequencies. The ratio between octaves is 2; the ratio between two successive notes (called a *semitone* is thus $\sqrt[12]{2}$. Each discrete frequency is called a note. Western keyboard instruments such as the piano have individual resonators tuned to each note, and a key which when pressed causes that note to sound. Typically, fretted string, brass, reed and woodwind instruments provide only a subset of the available notes in an octave, so to avoid these complications we shall use the keyboard as a reference instrument.

We noted before that the ear is particularly sensitive to frequencies up to about 5kHZ. If we start at, say, 20Hz and generate tones by multiplying by $\sqrt[12]{2}$, we get to 6kHz in around 100 steps, so we shouldn't be surprised to find that large concert pianos have 88 keys, and that nonstandard pianos have been constructed with 102 keys. Of course, we could have started at 25Hz or 19Hz. The particular mapping between the discrete notes and the continuum of frequencies is called the *tuning* of an instrument. For an equal tempered tuning such as 12-TET, it suffices to pick one particular frequency for one particular note: the other notes are then fixed by the $\sqrt[12]{2}$ ratio between semitones.

Current orchestral practice is to use 440Hz as a tuning standard and to tune the 49th key of a standard 88 note keyboard to to that frequency. This then results in the leftmost key generating 27.5Hz, and rightmost key generating 4186.01Hz. (A 102 key keyboard, rarely implemented, ranges from 16.3516Hz to 5587.65Hz.)

The MIDI standard defines 128 notes numbered from zero to 127, with twenty notes below and twenty notes above the standard piano keyboard. Key zero generates 8.17Hz and key 127 12,543.85Hz. Key 69 gives the 440Hz concert tuning standard.

A note on alternative tunings

There is nothing inherently perfect about this particular tuning. The 440Hz standard was internationally agreed in 1939, and became an ISO standard in 1955. However frequencies from 400Hz to 460hz are known to have been used historically, and those frequencies are more more than $\sqrt[5]{2}$ apart, which is greater than two semitones: playing an historical piece to modern tuning may therefore yield a performance that differs significantly from the composer's intent.

Apart from these shifts in reference frequency, equal temperament (the di-

vision of the octave using a constant ratio) is not the only way of mapping discrete notes on to the frequency continuum. Many musical traditions instead use ratios of small integers, since frequencies related in his way form harmonious combinations: these tunings are collectively called *just intonation* and arise naturally with certain classes of instrument. The ratio 3:2 forms the basis of the so-called *Pythagorean tuning*; many other systems exist.

The division of the octave into 12 notes is also merely a convention. Divisions into 15, 19, 34 and even 53 notes have been studied; one way of thinking about this is that by having more notes we can more closely approach the small-integer-ratio harmonics of just intonation. There is a vast literature on tunings which you can begin to explore through online articles: in the rest of this section we shall restrict ourselves to 12-TET.

6.8.4 Pure tones and instrument voices

The resonators used in musical instruments generate more than one frequency. Roughly speaking, the note that we hear is the lowest frequency produced, but there will usually be many related *overtones* present, usually integer multiples of the lowest frequency. The lowest frequency is called the fundamental; the integer-multiple frequencies are the harmonics. For a fundamental frequency f, the first harmonic has frequency $2f$, the second harmonic $3f$ and so on. Notice how these harmonics are at the octave intervals: hence a fundamental and its overtones together sound like a single note rather than resolving into several independent notes.

A pure single tone sounds rather other-worldly: a tuning fork or the human whistle is probably the closest thing to a pure tone resonator that most people hear. Of course, we can use the computer to generate pure tones, once we know the waveform.

Fourier analysis and synthesis

In 1822, Jean-Baptiste Joseph Fourier published a claim that any periodic waveform could be decomposed into a (possibly infinite) set of sinusoidal waveforms, which when added together would reconstruct the original waveform. This observation has turned out to have many applications in physics and engineering. It is of direct relevance to sound synthesis, since it suggests that if we can write a program that generates sine waves at different frequencies and then add them together in various proportions we can numerically construct any audio waveform. This is the principle behind music synthesizers. Interestingly, there is a procedure by which we can take an arbitrary waveform and numerically decompose it into its constituent sine waves. We call a display of the strength of each frequency a *spectrum analysis*.

If our musical instrument resonator produces only a fundamental and harmonics, then we can characterise the sound of, say, a piano string by listing the proportions of sine waves of frequencies f, $2f$, $3f$,... kf where k is chosen to be beyond the limit of human hearing. Even for a very low fundamental note such as 20Hz, the tenth harmonic exceeds 20kHz. Hence we have the prospect of

being able to accurately encode the detailed sound of a piano note into eleven real numbers, and then reconstituting the sound in realtime using software.

The conversion of a waveform to a spectrum display of frequencies is called Fourier Analysis; the reverse process of converting a series of proportions of sine waves to a single waveform is called Fourier Synthesis. We sometimes refer to operations on waveforms as 'working in the time domain' and operations on frequency proportions as 'working in the frequency domain'.

The Nyquist-Shannon criterion and making recordings

An ability to encode a single note of a single instrument accurately is clearly important for synthesis. However, it does not tell us how to capture the behaviour of an entire orchestra, or indeed how to encode non-musical sounds.

We can use a computer as a sound recorder by connecting a microphone and then measuring its output, say every microsecond. We use an Analogue to Digital Converter (ADC) to convert the voltage developed by the microphone into a number. Typical high quality digital audio systems use around 16 bits to represent each sample. By storing the resulting sequence of integer numbers, we can have a permanent record of a sound experience which may be reconstructed by converting the numbers back into voltages and applying the resulting waveform to a loudspeaker.

We can see that if we sample the original sound too slowly, then we may lose information. On the other hand, if we sample at very high speed then we shall require extra storage. A fascinating result which arises from Fourier analysis is that we can capture the full detail of any waveform, no matter how complex, up to some bounding frequency f by sampling the waveform at no more than $2f$. This is why CD quality audio samples waveforms at 44.1kHz: since the human ear can perceive sounds up to 20kHz, a sample rate of greater then 40kHz ensures that no information is lost. (In detail, the figure of 44.1kHz arises as a result of early experiments using video recorders to store audio information: you can read the story online.) This $2f$ requirement is called the Nyquist-Shannon criterion.

6.8.5 Tempo, rhythm and articulation

It is unusual for a note to be played continuously (although bagpipes and some other instruments have a *drone* which sounds continuously during a performance). Instead, the playing time is divided up into discrete beats which set the duration of the basic note.

In the western tradition, a piece of music will have an indicated *tempo*, sometimes expressed as *beats per minute* or bpm. As computer scientists, we might prefer to use Hz to specify the tempo, that is, beats per second. A very slow piece would be below 30bpm and a very fast piece above 200bpm, from which we can see that beat frequencies range from around 0.5 to 3.5Hz.

**** Todo: Rhythms**

Much of the character of a performance is embedded in the detailed way in which a performer uses the tempo. A straightforward approach is to leave a very

short silence at the end of each beat period, and to sound each note uniformly throughout the rest of the beat period. This very simplistic approach is easy to program but sounds, well, synthetic.

A human performer even when attempting uniformity will display some small variations in the length of notes. More significantly, humans players deliberately vary the details of note timing within the basic rhythm framework, a technique known as *articulation*. For instance, some notes may be run together into a smoothly connected frequency shift whereas other are deliberately shortened so as to create a jumpy effect. In wind instruments articulation is achieved by controlling airflow with the tongue; in stringed instruments by dampening the vibrations with the hand. Other forms of articulation include rapid periodic changes in amplitude (called tremolo) and rapid periodic changes in pitch (called vibrato). A slower shift in pitch is often called a *pitch bend*: some electric guitars (such as the fender Stratocaster) have an arm which allows the tension and length of the strings to be varied – this device is often called a tremolo bar (although really it is a vibrato bar).

6.8.6 Musical terminology for pitch

Musicians use a very large number of technical terms, and this can be rather overwhelming at a first encounter. However, we are interested in Domain Specific Languages, and musician's terminology certainly represents a very widely used language which is extremely domain specific, and as such can be the basis of some interesting case studies.

Musical nomenclature has grown up over a long historical period, and can seem rather arbitrary to outsiders even though there is usually an underlying logic. For instance, one might imagine that the divisions of an octave into 12-semitones might be represented by twelve unique names. In fact, in the western tradition there are seven unique names (the letters A through G inclusive), and two modifiers ♯ and ♭ (spoken sharp and flat) which raise (lower) by a semitone the note represented by a name.

This initially surprising naming convention arises from the observation that certain sequences of seven notes sound harmonious, and that the majority of western musical melodies are *mostly* constructed around seven note selections.

By appending a ♯ or a ♭ symbol to the seven basic names we can name the 'missing' five semitones. A conventional way of writing an ascending sequence of 12 semitones in an octave is:

A A♯ B C C♯ D D♯ E F F♯ G G♯

and a conventional way to write a descending sequence is:

A A♭ G G♭ F E E♭ D D♭ C B B♭

Using the 12-TET tuning (but not necessarily for other tunings), A♯ and B♭ represent the same frequency, and we can enumerate the full set of notes in an octave as

A A♯/B♭ B C C♯/D♭ D D♯/E♭ E F F♯/G♭ G G♯

**** Todo: Octave numbers**

6.8.7 Major and minor scales

Our basic pitch palette, then, comprises octaves of 12 fundamental notes each separated by a semitone. When played, notes come with a variety of harmonics which allow us to distinguish, say, a violin note from a guitar note.

Music can generate emotional responses in humans. It is clear that these responses are culturally conditioned, but nevertheless within a culture such as our own in which individuals are exposed to many musical pieces, strong associations between particular musical progressions and particular emotional states seem to be almost universally recognised — in the western tradition the difference between a joyous and a sad piece is well understood by most listeners.

The first component of mood is harmony. Some sequences of notes sound harmonious and some do not: we say they are discordant (which literally means that they disagree with each other). We can test our response to sequences by playing subsets of the 12 tones in an octave in ascending or descending order: it turns out that some sound good and some are unpleasant. Such a sequenced subset of the tones is called a *scale*.

If we think of the 12 semitones laid out as a 12-bit vector representing the presence or absence of a note within some scale, it is easy to see that there are $2^{12} = 4096$ scales. The one with all twelve notes in it is called the chromatic scale; its dual with no notes in at all is simply silence (and therefore of no musical utility).

We have already noted that western music focuses on scales with seven notes. It turns out that only two families of such seven note scales find wide application in popular music. The *major* scales start with any of the twelve notes and then include notes according to the increments

$$+2\ +2\ +1\ +2\ +2\ +2\ +1$$

The *minor* scales begin with any note, and then include notes according to the increments:

$$+2\ +1\ +2\ +2\ +1\ +3\ +1$$

Scales are usually played so as to finish one octave above the root. Hence, we play a major scale rooted on keyboard key k by playing the keys

$$k, k+2, k+4, k+5, k+7, k+9, k+11, k+12$$

and a minor scale with

$$k, k+2, k+3, k+5, k+7, k+8, k+11, k+12$$

As a further example, consider the scale made up of two equally spaced notes. These are three whole tones apart and thus called a tritone. When played, this creates, at best, a sense of tension: some might even say it sounds wrong.

6.8.8 Chords

A chord is a set of notes played simultaneously. Just as with scales, particular combinations sound harmonious, and the most common way of forming a chord for a root note k is to take the first, third and fifth elements of either the major or minor scale rooted on k. We write CM (spoken C-major) for the chord rooted on C using the major scale, and Cm (C-minor) for the chord rooted on C using the minor scale.

Chords formed of notes $k, k+6, k+12$ sound particularly inharmonious, made up as they are of a pair of tritones.

We can play a simple melody by picking out single notes. If we replace each note by the major chord rooted at that note then we get a fuller sound. We can do the same with the minor chords; and in general a melody and chords based on the minor scale will sound darker and perhaps more gentle: the major scale sounds brighter.

6.8.9 Synthesizing music with Java and MIDI

**** Todo: Overview of MIDI**

```
1  package uk.ac.rhul.cs.csle.artmusic;
2
3  import javax.sound.midi.MidiChannel;
4  import javax.sound.midi.MidiSystem;
5  import javax.sound.midi.Synthesizer;
6
7  public class ARTMiniMusicPlayer {
8    private Synthesizer synthesizer;
9    private MidiChannel[] channels;
10   private int defaultOctave = 5;
11   private int defaultVelocity = 50;
12   private int bpm;
13   private double bps;
14   private double beatPeriod;
15   private double beatRatio = 0.9;
16   private int beatSoundDelay = (int) (1000.0 * beatRatio / bps);
17   private int beatSilenceDelay = (int) (1000.0 * (1.0 - beatRatio) / bps);
18
19   public ARTMiniMusicPlayer() {
20     try {
21       System.out.print(MidiSystem.getMidiDeviceInfo());
22       synthesizer = MidiSystem.getSynthesizer();
23       synthesizer.open();
24       channels = synthesizer.getChannels();
25     } catch (Exception e) {
26       System.err.println("miniMusicPlayer exception: " + e.getMessage());
27       System.exit(1);
```

```
28        }
29
30        setBeatRatio(0.9);
31        setBpm(100);
32        setDefaultVelocity(50);
33    }
34
35    public int getDefaultOctave() {
36        return defaultOctave;
37    }
38
39    public void setDefaultOctave(int defaultOctave) {
40        this.defaultOctave = defaultOctave;
41    }
42
43    public int getDefaultVelocity() {
44        return defaultVelocity;
45    }
46
47    public void setDefaultVelocity(int defaultVelocity) {
48        this.defaultVelocity = defaultVelocity;
49    }
50
51    public int getBpm() {
52        return bpm;
53    }
54
55    public void setBpm(int bpm) {
56        this.bpm = bpm;
57        bps = bpm / 60.0;
58        beatPeriod = 1000.0 / bps;
59        beatSoundDelay = (int) (beatRatio * beatPeriod);
60        beatSilenceDelay = (int) ((1.0 - beatRatio) * beatPeriod);
61    }
62
63    public void setBeatRatio(double beatRatio) {
64        this.beatRatio = beatRatio;
65        beatSoundDelay = (int) (beatRatio * beatPeriod);
66        beatSilenceDelay = (int) ((1.0 - beatRatio) * beatPeriod);
67    }
68
69    private int noteNameToMidiKey(String n, int octave) {
70    // @formatter:off
71    int key = octave * 12 +
72              ( n.equals("C") ? 0
73              : n.equals("C#") ? 1
74              : n.equals("Db") ? 1
```

```
75            : n.equals("D") ? 2
76            : n.equals("D#") ? 3
77            : n.equals("Eb") ? 3
78            : n.equals("E") ? 4
79            : n.equals("F") ? 5
80            : n.equals("F#") ? 6
81            : n.equals("Gb") ? 6
82            : n.equals("G") ? 7
83            : n.equals("G#") ? 8
84            : n.equals("Ab") ? 8
85            : n.equals("A") ? 9
86            : n.equals("A#") ? 10
87            : n.equals("Bb") ? 10
88            : n.equals("B") ? 11
89            : -1);
90    // @formatter:on
91
92        if (key < 0 || key > 127) {
93          System.err.println("miniMusicPlayer exception: attempt to access out of range MIDI key " + n + oct
94          System.exit(1);
95        }
96        return key;
97      }
98
99      // Silence
100     public void rest(int beats) {
101       try {
102         Thread.sleep((long) (beats * beatPeriod));
103       } catch (InterruptedException e) {
104         /* ignore interruptedException */ }
105     }
106
107     // Single notes
108     public void play(int k) {
109       try {
110         channels[0].noteOn(k, defaultVelocity);
111         Thread.sleep(beatSoundDelay);
112         channels[0].noteOn(k, 0);
113         Thread.sleep(beatSilenceDelay);
114       } catch (InterruptedException e) {
115         /* ignore interruptedException */ }
116     }
117
118     public void play(String n) {
119       play(noteNameToMidiKey(n, defaultOctave));
120     }
121
```

```
122   public void play(String n, int octave) {
123     play(noteNameToMidiKey(n, octave));
124   }
125
126   // Arrays of notes
127   public void play(int[] k) {
128     try {
129       for (int i = 0; i < k.length; i++)
130         channels[1].noteOn(k[i], defaultVelocity);
131       Thread.sleep(beatSoundDelay);
132       for (int i = 0; i < k.length; i++)
133         channels[1].noteOn(k[i], 0);
134       Thread.sleep(beatSilenceDelay);
135     } catch (InterruptedException e) {
136       /* ignore interruptedException */ }
137   }
138
139   public void playSequentially(int[] k) {
140     try {
141       for (int i = 0; i < k.length; i++) {
142         channels[i].noteOn(k[i], defaultVelocity);
143         Thread.sleep(beatSoundDelay);
144         channels[i].noteOn(k[i], 0);
145         Thread.sleep(beatSilenceDelay);
146       }
147     } catch (InterruptedException e) {
148       /* ignore interruptedException */ }
149   }
150
151   // Scales
152   public void playScale(String n, ARTScale s) {
153     playScale(noteNameToMidiKey(n, defaultOctave), s);
154   }
155
156   public void playScale(String n, int octave, ARTScale s) {
157     playScale(noteNameToMidiKey(n, octave), s);
158   }
159
160   public void playScale(int k, ARTScale s) {
161     int[] keys;
162     switch (s) {
163     case CHROMATIC:
164       keys = new int[] { k, k + 1, k + 2, k + 3, k + 4, k + 5, k + 6, k + 7, k + 8, k + 9, k + 10, k + 1
165       break;
166
167     case MAJOR: // TTSTTTS
168       keys = new int[] { k, k + 2, k + 4, k + 5, k + 7, k + 9, k + 11, k + 12 };
```

```
169         break;
170
171       case MINOR_NATURAL: // TSTTSTT
172         keys = new int[] { k, k + 2, k + 3, k + 5, k + 7, k + 8, k + 10, k + 12 };
173         break;
174       case MINOR_HARMONIC: // TSTTS3S
175         keys = new int[] { k, k + 2, k + 3, k + 5, k + 7, k + 8, k + 11, k + 12 };
176         break;
177       case MINOR_MELODIC_ASCENDING: // TSTTS3S – harmonic with with sixth sharpened
178         keys = new int[] { k, k + 2, k + 3, k + 5, k + 7, k + 9, k + 11, k + 12 };
179         break;
180       case MINOR_MELODIC_DESCENDING: // TSTTS3S – harmonic with seventh flattened making it the
181         keys = new int[] { k + 12, k + 10, k + 8, k + 7, k + 5, k + 3, k + 2, k };
182         break;
183
184       default:
185         keys = new int[] { 0 };
186         break;
187       }
188       playSequentially(keys);
189   }
190
191   // Programmed chords
192   public void playChord(String n, ARTChord type) {
193     playChord(noteNameToMidiKey(n, defaultOctave), type);
194   }
195
196   public void playChord(String n, int octave, ARTChord type) {
197     playChord(noteNameToMidiKey(n, octave), type);
198   }
199
200   public void playChord(int k, ARTChord type) {
201     int[] keys;
202     switch (type) {
203     case NONE:
204       keys = new int[] { k };
205       break;
206     case MAJOR:
207       keys = new int[] { k, k + 4, k + 7 };
208       break;
209     case MAJOR7:
210       keys = new int[] { k, k + 4, k + 7, k + 11 };
211       break;
212     case MINOR:
213       keys = new int[] { k, k + 3, k + 7 };
214       break;
215     case MINOR7:
```

```
216         keys = new int[] { k, k + 4, k + 7 };
217         break;
218       default:
219         keys = new int[] { 0 };
220         break;
221       }
222       play(keys);
223     }
224
225     private void tune() {
226       int base = 47;
227       play(base + 14);
228       play(base + 12);
229       play(base + 11);
230       play(base + 7);
231       play(base + 5);
232       play(base + 7);
233       play(base + 2);
234       rest(2);
235     }
236
237     private void tuneChordMajor() {
238       int base = noteNameToMidiKey("C", 5);
239       playChord(base + 14, ARTChord.MAJOR);
240       playChord(base + 12, ARTChord.MAJOR);
241       playChord(base + 11, ARTChord.MAJOR);
242       playChord(base + 7, ARTChord.MAJOR);
243       playChord(base + 5, ARTChord.MAJOR);
244       playChord(base + 7, ARTChord.MAJOR);
245       playChord(base + 2, ARTChord.MAJOR);
246     }
247
248     private void tuneChordMinor() {
249       int base = noteNameToMidiKey("C", 5);
250       playChord(base + 14, ARTChord.MINOR);
251       playChord(base + 12, ARTChord.MINOR);
252       playChord(base + 11, ARTChord.MINOR);
253       playChord(base + 7, ARTChord.MINOR);
254       playChord(base + 5, ARTChord.MINOR);
255       playChord(base + 7, ARTChord.MINOR);
256       playChord(base + 2, ARTChord.MINOR);
257     }
258
259     public void close() {
260       synthesizer.close();
261     }
262
```

```
263     public static void main(String[] args) {
264       System.err.println("miniMusicPlayer test routine");
265       ARTMiniMusicPlayer mp = new ARTMiniMusicPlayer();
266
267        mp.playScale("C", ARTScale.CHROMATIC);
268        mp.rest(2);
269        String note = "C";
270        int octave = 6;
271        mp.play(note, octave);
272        mp.rest(2);
273        mp.playScale("C", ARTScale.MAJOR);
274        mp.rest(2);
275        mp.playScale("C", ARTScale.MINOR_NATURAL);
276        mp.rest(2);
277        mp.playScale("C", ARTScale.MINOR_HARMONIC);
278        mp.rest(2);
279        mp.playScale("C", ARTScale.MINOR_MELODIC_ASCENDING);
280        mp.playScale("C", ARTScale.MINOR_MELODIC_DESCENDING);
281        mp.rest(2);
282        mp.playChord("C", ARTChord.MAJOR);
283        mp.rest(2);
284        mp.playChord("C", ARTChord.MINOR);
285        mp.rest(2);
286        mp.tune();
287        mp.rest(2);
288        mp.tuneChordMajor();
289        mp.rest(2);
290        mp.tuneChordMinor();
291        mp.rest(2);
292   // Tritone scale and scale
293       mp.playSequentially(new int[] { 50, 56, 62 });
294       mp.rest(2);
295       mp.play(new int[] { 50, 56, 62 });
296       mp.rest(2);
297
298       mp.close();
299     }
300   }
```

6.8.10 `minimusic` – a DSL to access `MiniMusicPlayer`

```
1  melody sanctuary {
2
3  D+M C+M B+ G F G m D m7
4  }
```

```
5
6  x = 3;
7  while x > 0 do { print("x is ", x, "\n"); x = x -1; }
8
9  play sanctuary;
```

```
1  (*****************************************************************************
2  *
3  * miniMusic.art - Adrian Johnstone 18 Februrary 2017
4  *
5  *****************************************************************************)
6  prelude { import java.util.HashMap; import uk.ac.rhul.cs.csle.artmusic.*; }
7
8  support {
9  HashMap<String, Integer> variables = new HashMap<String, Integer>();
10 HashMap<String, ARTGLLRDTHandle> melodies = new HashMap<String, ARTGLLRDTHandle>();
11 ARTMiniMusicPlayer mp = new ARTMiniMusicPlayer();
12 }
13
14 whitespace &WHITESPACE
15 whitespace &COMMENT_NEST_ART
16 whitespace &COMMENT_LINE_C
17
18 statements ::= statement | statement statements
19
20 statement ::= ID '=' e0 ';' { variables.put(ID1.v, e01.v); } | (* assignment *)
21
22                 'if' e0 'then' statement< elseOpt< (* if statement *)
23                 { if (e01.v != 0)
24                       artEvaluate(statement.statement1, statement1);
25                   else
26                       artEvaluate(statement.elseOpt1, elseOpt1);
27                 } |
28
29                 'while' e0< 'do' statement< (* while statement *)
30                 { artEvaluate(statement.e01, e01);
31                   while (e01.v != 0) {
32                       artEvaluate(statement.statement1, statement1);
33                       artEvaluate(statement.e01, e01);
34                   }
35                 } |
36
37                 'print' '(' printElements ')' ';' | (* print statement *)
38
39                 'melody' ID statement< { melodies.put(ID1.v, statement.statement1); } |
40                 'play' ID ';'
```

```
41                      { if (!melodies.containsKey(ID1.v))
42                          artText.println(ARTTextLevel.WARNING, "ignoring request to play undefined melod
43                        else
44                          artEvaluate(melodies.get(ID1.v), null);
45                      } |
46
47                  '{' statements '}' | (* compound statement *)
48
49                  bpm | defaultOctave | note | chord | rest
50
51  elseOpt ::= 'else' statement | #
52
53  bpm ::= 'bpm' INTEGER { mp.setBpm(INTEGER1.v); }
54
55  beatRatio ::= 'beatRatio' REAL { mp.setBeatRatio(REAL1.v); }
56
57  defaultOctave ::= 'defaultOctave' INTEGER
58   { if (INTEGER1.v < 0 || INTEGER1.v > 10)
59         artText.println(ARTTextLevel.WARNING, "ignoring illegal MIDI octave number " + INTEGER1.v);
60       else
61          mp.setDefaultOctave(INTEGER1.v);
62   }
63
64  note ::= simpleNote chordMode { mp.playChord(simpleNote1.v.trim(), chordMode1.v ); } |
65              simpleNote shifters chordMode { mp.playChord(simpleNote1.v.trim(),
66                 mp.getDefaultOctave() + shifters1.v, chordMode1.v); } |
67              simpleNote INTEGER chordMode { mp.playChord(simpleNote1.v.trim(), INTEGER1.v, chordMod
68
69  chordMode <v:ARTChord> ::= # { chordMode.v = ARTChord.NONE; } |
70                             'm' { chordMode.v = ARTChord.MINOR; } | 'm7' { chordMode.v = ARTC
71                             'M' { chordMode.v = ARTChord.MAJOR; } | 'M7' { chordMode.v = ART
72
73  simpleNote<leftExtent:int rightExtent:int v:String> ::=
74     simpleNoteLexeme { simpleNote.v = artLexeme(simpleNote.leftExtent, simpleNote.rightExtent).trim(); }
75
76  simpleNoteLexeme ::= 'A' | 'A#' | 'Bb' | 'B' | 'C' | 'C#' | 'Db' | 'D' | 'D#' | 'Eb' | 'E' | 'F' | 'F#' | 'Gb' | '
77
78  shifters<v:int> ::= '+' {shifters.v = 1;} | '−' {shifters.v = −1;} |
79                      '+' shifters {shifters.v = shifters1.v + 1; } |
80                      '−' shifters {shifters.v = shifters1.v − 1; }
81
82  chord ::= '[' notes ']'
83
84  notes ::= note | note notes
85
86  rest ::= '.' { mp.rest(1); } | '..' { mp.rest(2); } | '...' { mp.rest(3); } | '....' { mp.rest(4); }
87
```

```
printElements ::= STRING_DQ { artText.printf("%s", STRING_DQ1.v); } |
                  STRING_DQ { artText.printf("%s", STRING_DQ1.v); } ',' printElements |
                  e0 { artText.printf("%d", e01.v); } | e0 { artText.printf("%d", e01.v); } ',' printEleme

e0 <v:int> ::= e1 { e0.v = e11.v; } |
               e1 '>' e1 { e0.v = e11.v > e12.v ? 1 : 0; } | (* Greater than *)
               e1 '<' e1 { e0.v = e11.v < e12.v ? 1 : 0; } | (* Less than *)
               e1 '>=' e1 { e0.v = e11.v >= e12.v ? 1 : 0; } | (* Greater than or equals*)
               e1 '<=' e1 { e0.v = e11.v <= e12.v ? 1 : 0; } | (* Less than or equals *)
               e1 '==' e1 { e0.v = e11.v == e12.v ? 1 : 0; } | (* Equal to *)
               e1 '!=' e1 { e0.v = e11.v != e12.v ? 1 : 0; } (* Not equal to *)

e1 <v:int> ::= e2 { e1.v = e21.v; } |
               e1 '+' e2 { e1.v = e11.v + e21.v; } | (* Add *)
               e1 '-' e2 { e1.v = e11.v - e21.v; } (* Subtract *)

e2 <v:int> ::= e3 { e2.v= e31.v; } |
               e2 '*' e3 { e2.v = e21.v * e31.v; } | (* Multiply *)
               e2 '/' e3 { e2.v = e21.v / e31.v; } | (* Divide *)
               e2 '%' e3 { e2.v = e21.v % e31.v; } (* Mod *)

e3 <v:int> ::= e4 {e3.v = e41.v; } |
               '+' e3 {e3.v = e41.v; } | (* Posite *)
               '-' e3 {e3.v = -e41.v; } (* Negate *)

e4 <v:int> ::= e5 { e4.v = e51.v; } |
               e5 '**' e4 {e4.v = (int) Math.pow(e51.v, e41.v); } (* exponentiate *)

e5 <v:int> ::= INTEGER {e5.v = INTEGER1.v; } | (* Integer literal *)
               ID { e5.v = variables.get(ID1.v); } | (* Variable access *)
               '(' e1 { e5.v = e11.v; } ')' (* Parenthesised expression *)

ID <leftExtent:int rightExtent:int lexeme:String v:String> ::=
  &ID {ID.lexeme = artLexeme(ID.leftExtent, ID.rightExtent); ID.v = artLexemeAsID(ID.leftExtent, ID.righ

INTEGER <leftExtent:int rightExtent:int lexeme:String v:int> ::=
  &INTEGER {INTEGER.lexeme = artLexeme(INTEGER.leftExtent, INTEGER.rightExtent);
    INTEGER.v = artLexemeAsInteger(INTEGER.leftExtent, INTEGER.rightExtent); }

REAL <leftExtent:int rightExtent:int lexeme:String v:double> ::=
  &REAL {REAL.lexeme = artLexeme(REAL.leftExtent, REAL.rightExtent);
    REAL.v = artLexemeAsInteger(REAL.leftExtent, REAL.rightExtent); }

STRING_DQ <leftExtent:int rightExtent:int lexeme:String v:String> ::=
  &STRING_DQ {STRING_DQ.lexeme = artLexeme(STRING_DQ.leftExtent, STRING_DQ.rightExtent);
    STRING_DQ.v = artLexemeAsString(STRING_DQ.leftExtent, STRING_DQ.rightExtent); }
```

6.9 The image processing domain

6.10 The 3D object domain

A Using ART

ART (Ambiguity Resiliant Translation) is a tool for specifying the syntax and semantics of language processors.

Traditional language implementation tools use restricted algorithms which remove ambiguity in ways that are not conveniently controllable by the language designer. ART takes a different approach, employing general algorithms which allow ambiguous interpretations of language rules to be maintained through various stages of translation, and supporting principled resolution of ambiguity.

ART supports unrestricted context-free lexing and parsing, term rewriting, and computation of tree attributes. For ambiguous rules, the general lexing and parsing algorithms will construct all derivations: *chooser relations* may optionally be used to suppress some derivations in a controlled manner.

There are two styles of attribute computation: *Syntax Directed Translation (SDT)* in which programming language fragments are embedded within rules and executed as a side-effect of the parsing process; and *Syntax Directed Definition (SDD)* which uses a set of equations over the attributes to define language semantics. An SDD with no side-effects is called an *Attribute Grammar*.

The term rewriter may be used to directly implement the *Structured Operational Semantics* style of formal language specification as well as performing general purpose language specifications.

A.1 Installing and running ART

ART is written in Java, and distributed as a single file `art.jar` which requires an installed Java Runtime Environment but which has no other dependencies. ART may be run from the command line as

```
java -cp path/art.jar uk.ac.rhul.cs.csle.art.ART restOfLine
```

where *path* is the directory containing `art.jar` and *restOfLine* is any valid ART specification.

Environment variables

ART uses environment variable `artpath` to search for specification files, and ART's convenience scripts use environment variable `arthome` to locate `art.jar`.

Convenience scripts

On Windows, ART is conveniently activated via this batch file `art.bat`

```
java -cp .;%arthome%/art.jar uk.ac.rhul.cs.csle.art.ART %*
```

Using the bash shell, ART is conveniently activated via this script file `art.sh`

```
#!/bin/bash
java -cp ".:$arthome/art.jar" uk.ac.rhul.cs.csle.art.ART "$@"
```

Convenience shortcuts

The most common ART usage is to run the tool on a single specification file with an optional single input file and there are two special cases of *restOfLine* for these.

1. If *restOfLine* comprises exactly one space delimited field *file1* that does not begin with a `!` character, then it is rewritten to

   `!merge` *file1* `!try`

   Typing `art mySpec.art` at the command line then has the effect of running ART on the specification `myspec.art`.

2. If *restOfLine* comprises exactly two space delimited fields *file1 file2* neither of which begin with a `!` character, then it is rewritten to

   `!merge` *file1* `!input` *file2* `!try`

   Typing `art mySpec.art myInput.str` at the command line then has the effect of running ART on the specification `myspec.art` with string input file `myInput.str`.

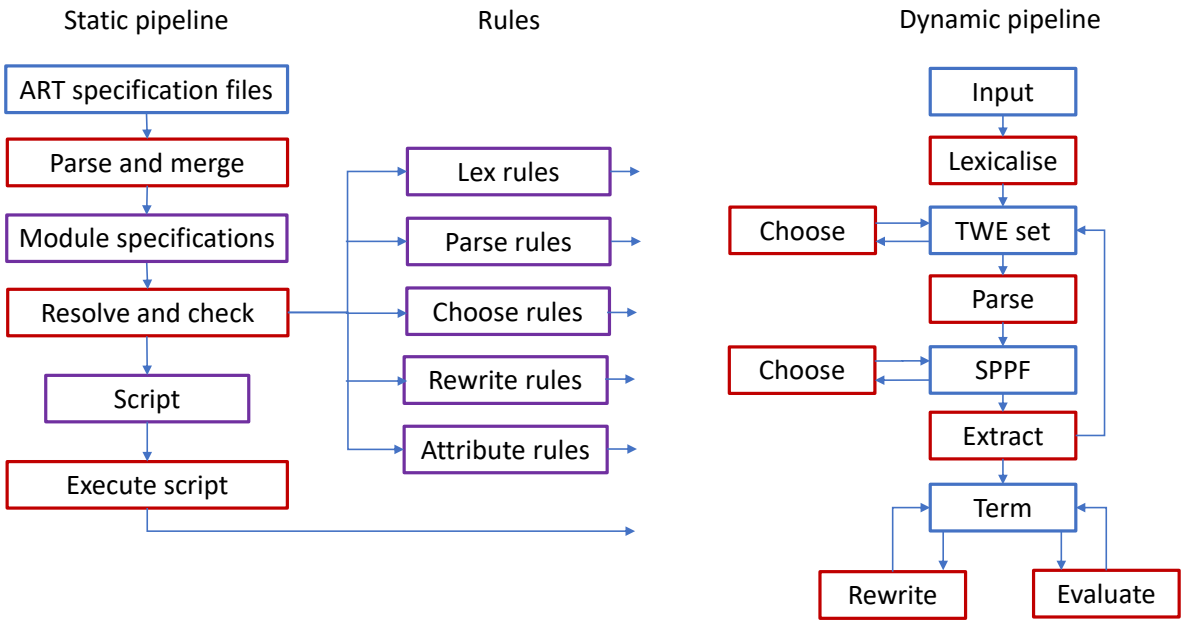
A.2 The ART pipelines

ART passes language inputs through a set of processes which include lexicalisation, parsing, ambiguity reduction *via* choice relations, term extraction, term rewriting and term attribute evaluation.

These various subsystems form the *dynamic pipeline* which takes language inputs and processes them according to rules given in the language specification.

The rules themselves are constructed by the *static* pipeline from files written in the ART specification language which contain rules and directives grouped into *modules*.

Each module contains five (possibly empty) sets of rules (lex, parse, choose, rewrite and evaluate) and a script. Modules may use rules and directives from other modules: the intention is to allow both standard libraries of common idioms to be built up, and to allow large language specifications to be segmented into managable pieces.



ART specifications comprise a mix of rule definitions and directives.

Rules are *declarative* and static: that is the rule definitions are used to construct the rules by which parsing, rewriting and evaluation will proceed *independently of any input*, and the ordering of rules in the specification is not significant.

ART directives (the names of which all start with an exclamation mark (!) form a program script to be executed by the dynamic program and should be read sequentially.

A.3 First examples

A.4 The static and dynamic pipelines in detail

A.4.1 Static pipeline directive summary

```
!merge
!module
!use
!paraterminal
!cfgElements
```

A.4.2 Dynamic pipeline directive summary

Standalone subsystems

```
!termTool
!grammarWrite
!lexerData
```
— Deprecated: instead switch on the TWE set analyses

Logging

```
!verbosity <n>
!trace <n>
!statistics <n>
!parseCounts
```

Datastructure visualisation

```
!inputPrint
!twePrint !tweWrite !tweShow
!gssPrint !gssWrite !gssShow
!sppfPrint !sppfWrite !sppfShow
!treePrint !treeWrite !treeShow
!termPrint !termWrite !termShow
```

Lexer control

```
!lexDFA
```
Unavailable

```
!lexGLL
!lexHardCoded
!lexWSSuffix
!whitespace <nonterminal>
!absorb <nonterminal>
!absorb #
!injectInstance <nonterminal>
!injectInstance #
!injectProduction <nonterminal>
!injectProduction #
```

TWE set analysis

```
!tweFromSPPF
!tweTokenWrite
!tweExtents
!tweSegments
!tweRecursive
```

Chooser enabling

```
!tweLonges !twePriority !tweDead
!sppfLongest !sppfPriority
```

Generated parser control

```
!outputDirectory <plainstring>
!namespace <plainstring>
!lexerName <plainstring>
!parserName <plainstring>
!generateDynamic !generateStatic !generateState !generateFragment
!generatePool
!generateJava !generateC++ !generateML
```

Native language insertions for generated parsers

```
!prelude <action>
!support <action>
```

GLL template control

```
!GLLPredictivePops
!GLLFIFODescriptors
!GLLSuppressPopGuard !GLLSuppressProductionGuard !GLLSuppressTestRepeat

!GLLSuppressSemantics
```

Parse algorithms and implementations

```
!earley2007LinkedAPI
!earley, !earleyLinkedAPI, !earleyIndexedAPI, !earleyIndexedPool, !earleyIndexedData

!earleyTable !earleyTableLinkedAPI !earleyTableIndexedAPI !earleyTableIndexedPool
!earleyTableIndexedData
!cnp !cnpLinkedAPI !cnpIndexedAPI !cnpIndexedPool !cnpGeneratorPool

!lcnp !lcnpLinkedAPI !lcnpIndexedAPI !lcnpIndexedPool !lcnpGeneratorPool

!gll !gllGeneratorPool
!gllTWEGeneratorPool
```

```
!gllClusteredGeneratorPool
!mgll !mgllGeneratorPool
!osbrd !osbrdGenerator
!sml97Parser
```

Rewrite rule elisions

```
!relation <relation> <entities...>
```

Term rewriter strategy selection

`!strategyRoot` — Set initial term rewriter strategy
`!strategyPostOrder` —
`!strategyPreOrderOneShot` —

Pipeline activation

`!main <moduleName>` ==
`!start <nonterminal>` — Set parser start nonterminal
`!start <relation>` — Set term rewriter start relation
`!input <input>` — Set input string or term
`!result <term>` — Set test term
`!try` — Run pipeline
`!try <input>` — Run pipeline on ¡input¿
`!try <input> = <term>` — Run pipeline on `<input>`; compare result to `<term>`

A.5 The ART specification language reference manual

Lexical structure

An ART specification is a normal text file which contains a string of characters. As is conventional with many software languages, this stream of characters is initially *lexicalised* into tokens. A token might always correspond to a single substring of characters, such as the sign for addition `+` or it might correspond to a multitude of possible substrings, such as the token for integers which could represent 1 or 2 or 3, and so on.

We call the substring corresponding to an instance of a token the *lexeme* of that token. The set of possible lexemes for a token is called the *pattern* of the token.

Each lexeme has a *lexical value*. For instance, the integer lexems `07`, `007` and `7` all have the same value: the number seven in its natural representation in ART's implementation language (which is Java). For string-like lexemes such as `"abc"` the value is the string with the delimiters removed: `abc`, and with any escape sequences (see below) replaced by their single character interpetation.

As is conventional in many programming languages, the lexicalisation of ART specifications discards all comment and whitespace tokens. The effect of this is that arbitrary comments and whitespace may appear at the boundaries of other tokens, and thus the layout of an ART specification has no significance (although an ART specification may create parsers that do *not* themselves suppress whitespace: see section A.6).

In the rest of this section we specify the pattern and values for each token in the ART specification language.

String-like tokens

Delimited string are used in four ways: the single character literal introduced by a back quote `` ` ``, and the full string literals delimited by `'...'` or `"..."` and the special identifier form `$...$`

All four styles share the property that their lexemes respect Java-style escape sequences: within the body of a string-like lexeme: a single backslash `\` introduces an escape sequence that will be interpreted as a single character. The available escape sequences are as follows.

`\u`*`abcd`*	Unicode character where *`abcd`* are hexadecimal digits
`\b`	backspace
`\f`	formfeed
`\n`	newline
`\r`	carriage return
`\t`	tab
`\`*`x`*	Any other character *`x`*

The lexical value of a string-like lexeme is the substring formed by removing the delimiters from the lexeme and then replacing embedded escape sequences according to this table. So for instance, the value of lexeme `'ART\'s'` is `ART's` since the `\'` escape sequence yields the character `'`.

Comments

Comments within ART specifications have two forms.

1. An ART *line comment* is introduced by a double slash `//` and continues to next end of line.

2. An ART *block comment* is delimited by `(*...*)` where ... denotes any character string without an unbalanced embedded `*)`. ART block comments nest, so blocks of specification may be 'commented outeven if they include comments (as long as balanced nesting is observed).

Whitespace

ART specification language whitespace tokens have pattern

```
( ` ` \n ` \t )*
```

that is zero or more space, newline or tab characters.

Identifiers

An ART identifier is either a simple identifier, or a dollar-delimited special identifier.

Simple identifiers follow the conventions of the Java language with pattern

```
(`\_ `a..`z `A..`Z) (`_ `a..`z `A..`Z `0..`9)*
```

that is they begin with an alphabetic character or underscore, and continue with zero or more alphabetic characters, digits or underscores. The lexical value of a simple identifier is the same as its lexeme.

Special identifiers are arbitrary dollar-delimited strings `$...$`. As a string-like token, the lexical value is the substring formed by removing the delimiters and replacing escape sequences. Special identifiers are intended for exceptional use. Although the lexical value of identifiers `_x1` and `$_x1$` are identical, the simple form is preferred wherever it may be used.

Boolean literals

ART has separate tokens for boolean true and false, with patterns `True` and `False` respectively and corresponding lexical values represented internally as Java primitive type `bool`.

Numeric literals

Numeric literals may be decimal integers, hexadecimal integers or decimal reals.

1. An integer literal has pattern `(`0..`9)+`, that is a non-empty string of decimal digits in the range zero to nine; lexical values are represented internally as Java primitive type `int`.

2. A hexadecimal integer literal has pattern

```
`0 (`x`X) (`0..`9  `a..`f  `A..`F)+
```

that is `0x` or `0X` followed by a non-empty string of hexadecimal digits in the range zero to nine or the first six alphabetic characters, upper or lower case; lexical values are represented internally as Java primitive type `int`.

3. A real literal has pattern

```
(`0..`9)+ `. (`0..`9)+ ( (`e  `E) (`0..`9)+ )?
```

that is two non-empty strings of decimal digits in the range zero to nine separated by a decimal point and followed by an optional *exponent*; lexical values are reprsented internally as Java primitive type `double`.

Other ART tokens

There are a large number of other ART tokens which are used as operators and keywords. These simple tokens all have patterns that contain a single lexeme, and for each their lexical value is the lexeme itself.

`module` *more to follow*

A.6 String rewrite rules and parsing

String rewrite rules are used to specify grammars for parsing from strings to derivation trees. The general form of a string rewrite rule is one of

```
stringRewrite ::= ID (\dirsection{<nativeAttribute+' >)?  '::= srRHS
srRhs ::= srCat  srCat '' srRhs
```

where `ID` is an identifier and `sRHS` is sequence of extended Backus-Naur Form (EBNF) expressions over string rule elements, separated by `|` tokens.

EBNF operators

concatenation

alternation

group

optional

Kleene closure

positive closure

range

not

String rule elements

nonterminals

case-insensitive terminals

case-sensitive terminals

character terminals

builtin terminals

Native attribute declarations and actions

Value expressions

Whitespace management in string rewrite rules

String rewrite rules may be rewritten to display one of three modes of automatic whitespace handline:

`none`

`inject` add an instance of the WS nonterminal after each terminal

`absorb` add an instance of the WS terminal at the end of

Paraterminalise tokens in the phrase level grammar.
Paraterminals from ‘x in the phrase level grammar do not have whitespace injection or absorption applied to them.

A.7 Terms and term rewrite rules

Term rewrite rules are used to rewrite terms represented as prefix expressions. Equivalently, term rewrite rules may be thought of as tree to tree rewrites. The general form of a term rewrite rules is

```
premise* --- conclusion
```

that is, zero or more premises followed by a `---` token followed by a conclusion.

A.8 RAG rewrite rules

This section intentionally blank

A.9 Directives

A.9.1 `try` clauses

A.10 Value types and operations

A.11 An overview of ART's implementation

A.12 ART package and class documentation

A.13 ART concisely

The ART static pipeline processes ART specifications to construct a set of modules containing rules, and a single list of *dynamic* directives:

1. Parse initial specification
2. Recursively process `!merge` directives to form final specification
3. Construct modules using `!module`, `!import` and `!export` directives
4. Construct an imperative program by concatenating dynamic directives.

The ART dynamic pipeline connnects a lexer, a parser, a term rewriter and an attribute evaluator to process an input string. The pipeline is parameterised by the rules and by dynamic directives which specify, for instance, the particular parsing algorithm to be used. the `!try` directive triggers a single pipeline run over an input string or term.

Stage	Process		Result
1.	Read input	→	Character string or term
2.	Lexicalise	→	TWE set
3.	Lexical choose	→	TWE set
4.	Parse	→	ESPPF
5.	Derivation choose	→	Term
6.	Rewrite	→	Term
7.	Evaluate	→	Value

‘

The ART command line interface constructs constructs an initial specification F from the space delimited command line arguments $a_1 \ldots a_k$ as follows:

assign the empty string to F
if a_1 does not begin with a shriek (`!`) **then for** i from 1 to k append a_i␣
else {
    append `!merge`␣`'`a_1`'`␣
    if a_2 does not begin with a shriek (`!`) **then for** i from 2 to k append a_i␣
    else {
        append `!input`␣`'`a_1`'`␣
        for i from 3 to k append a_i␣
    }}
if F does not contain the substring `!try` **then** append `!try`

The effect of this is that a command line such as

```
java -jar art.jar spec.art input.str !option1
```

yields the initial ART specification

```
!merge 'spec.art' !input 'input.str' !option1 !try
```

which will process the contents of file `input.str` using the rules in `spec.art` with directive `!option1`.

Static pipeline directives control the process of constructing modules

`!merge` hjgjgh

`!module`
`!use`

Dynamic pipeline directives

Tracing, creating and accessing pipeline data structures

`!verbosity n` set the console verbosity threshold to n (DEFAULT 5)
`!trace n` set the console trace threshold for processes (DEFAULT 0)
`!log n` set the logging threshold to n

For x in input, TWE, ESPPF, GSS, tree, term: !xCountsprint summary statistics !xPrint print contents !xPrintFull print detailed contents !xWrite 'f'write detailed contents as text to file f !xShowcreate a graphical visialisation !xShowFullcreate a detailed graphical visialisation !xDump 'f'write binary representation to file 'f' !xUndump 'f'read binary representation from file 'f'

`!main m` select m as main module
`!start N` set the start nonterminal for the parser to nonterminal N
`!start R` set the initial rewrite relation to relation R
`!try` trigger pipeline execution

Input control

`!input "s"` set current input to the literal string s
`!input 'f'` set current input to the string contents of file named f
`!inputPrint` print the current input
`!inputCounts` print the number of lines and the overall number of characters for current input

Lexer/parser interface control

`!paraterminal n` declare nonterminal n as a paraterminal (n may also be a comma delimited list of nonterminals)
`!wsAbsorbLeft` absorb prefix whitespace into left extent
`!wsAbsorbRight` absorb postfix whitespace into right extent (DEFAULT)
`!wsAbsorb` absorb default whitespace (' — 't — 'n — 'r)* after each paraterminal is recognised
`!wsAbsorb N` absorb whitespace defined by nonterminal N after each paraterminal is recognised
`!wsInject` inject (' — 't — 'n — 'r)* after each righ hand side instance of a paraterminal
`!wsInject N` inject nontermonal N after each right hand side instance of a paraterminal

Lexer control

`!lexCounts` print final lexer statistics
`!lexDisable` disable lexing, using current TWE set for parser input
`!lexDFA` select DFA based recogniser (DEFAULT)
`!lexGLL` select GLL recogniser
`!lexPlugin` select hand-crafted recognisers defined by lexer plugin code

TWE set management

`!tweShortest` enable suppression of TWE element ¡t,i,j¿ if s¡¡t and there exists ¡s,i,k¿ with k¡j (t, s tokens; i, j extents)
`!tweLongest` enable suppression of TWE element ¡t,i,j¿ if s¿¿t and there exists ¡s,i,k¿ with k¿j (t, s tokens; i, j extents)
`!twePriority` enable suppression of TWE element ¡t,i,j¿ if s¿t and there exists ¡s,i,j¿ (t, s tokens; i, j extents)
`!tweDead` enable suppression of TWE element if it is not on a path that spans the input string
`!tweSelectOne` arbitrarily select a single lexicalisation within the TWE set if there is one after chooser application
`!twePrint` print non-suppressed contents of TWE set
`!twePrintFull` print all TWE set elements
`!tweCounts` print TWE set cardinalities
`!tweAmbiguityClasses` collect and print current TWE set's ambiguity classes
`!tweLexicalisations` compute lexicalisation counts and report: the next three directives enable different aspects of lexicalisation counting
`!tweExtents` in tweCounts, use extents
`!tweSegments` in tweCounts, use segments
`!tweRecursive` in tweCounts, exhastively count lexicalisations (only useful for small TWE sets)
`!tweDump` dump TWE set to ARTTWE.twe in a format that can be loaded by V3 MGLL parsers
`!tweTokenWrite 'f'` write selected lexicalisation out as a token string to file 'f'

Parser control

`!parseCounts` print final parser statistics

select parser mode: for x in mgll gll gllClustered gllTWERecognise cnp lcnp mcnp lr glr rnglr brnglr earley earley2007 OSBRD RD `!xTermAPI` use algorithm x reading term based grammar and using standard API functions for support
`!xIndexedAPI` use algorithm x reading lookup table based grammar and using standard API functions for support
`!xIndexedPool` use algorithm x reading lookup table based grammar and Hash Pool memory management for support
`!xGeneratorPool` write out standalone parser which uses algorithm x reading lookup table based grammar and Hash Pool memory management for support

SPPF management

`!sppfShow` output visualisation of SPPF in file 'sppf.dot'
`!sppfChooseCounts` Report on node numbers before and after SPPF choosers run
`!sppfShortest` suppress SPPF packed node ¡l,j¿ if l¡¡m and there exists sibling ¡m,k¿ with k¡j (l,m slots; j pivot)
`!sppfLongest` suppress SPPF packed node ¡l,j¿ if l¿¿m and there exists sib-

ling ¡m,k¿ with k¿j (l,m slots; j pivot)
`!sppfPriority` suppress SPPF packed node ¡l,j¿ if l¿m and there exists sibling ¡m,k¿ (l,m slots; j,k pivot)
`!sppfDead` suppress SPPF packed node if it is unreachable from node ¡S,0,n¿
`!sppfOrderedLongest` suppress SPPF packed node ¡l,j¿ if there exists sibling ¡m,k¿ with k¿j OR if m appears before l in the specification ** Issue
`!sppfSelectOne` arbitrarily select a single derivation from the SPPF if there is one after chooser application
`!sppfCountArities` compute a histogram of the arities of all symbol/intermediate nodes reachable from the SPPF root
`!sppfCountDerivations` attempt to count all of the derivations in an SPPF (warning: time exponential in arities)
`!sppfCountSentences` attempt to enumerate all of the sentences in an SPPF by constructing each yield and adding it to a set of sentences (warning: time exponential in arities and potentially space exponential too)
`!sppfToTWE` (was!tweFromSPPF)construct a TWE set containing the yields for all of the unsuppressed derivations in the SPPF

Rewrite management

`!rewriteConfiguration <relation> <entities>` Create a rewrite configuration allowing elided eSOS rules to be used
`!rewriteDisable` disable the term rewriter
`!rewritePure` redex matching at root only (Default)
`!rewritePreorder` redex matching in preorder traversal
`!rewritePostorder` redex matching in postorder traversal
`!rewriteOneStep` perform only a single step - default: step until normalised
`!rewriteContractum` after a rewrite has been performed, resume original term traversal at the rewritten redex
`!rewriteResume` after a rewrite has been performed, resume original term traversal at the right-sibling of the rewritten redex

Attribute evaluator management

`!evalDisable` disable the attribute evaluator

Standalone tools

`!termTool` start the term rewriting tutorial tool
`!grammarWrite` write out parse, lexer, character level, token and pretty printed grammars
`!generateDepth` generate strings from grammar depth first
`!generateBreadth` generate strings from grammar breadth first
`!generateRandom` generate strings by randomly selecting the expansion instance and right hand side
`!extractJLS` extract Java Language Specification grammar from text snipped document
`!compressWhitespaceJava` compress Java file whitespace runs to single character

Rules ART uses three kinds of rule: context free grammar rules, choser rules and rewrite inference rules. CFG rules and rewrite rules may optionally contain attribute equations which are evaluated after the tree stabilises.

Context Free Grammar rules

A CFG rule has the form ¡nonterminal¿ ::= ¡cfgRHS¿

CFG rules are composed from nonterminals, terminals, the empty string symbol, EBNF operators and the ::= symbol

A nonterminal is denoted by an alphnumeric symbol or a string delimited by $ characters: An alphanumeric identifier may not begin with ART, art or any other mix of cases Terminals have four subclass: 'case sensitive' "case insensitive" &builtin 'c where c is a single character The empty string is denoted by # EBNF operators are: postfix * Kleene star, postifx + positive closure, postfix ? optional, ( ) do-first, infix — alternation, infix concatentaion, infix difference, prefix not,

adrian ::= 'xb'* C C ::= ('c' — 'C')+ — #

Choosers

The choice mechanism utilises named sets of three-tuples (higher, longer, shorter) where higher, longer and shorter are sets of ordered pairs of grammar elements which may include terminals, nonterminals, productions and slots.

A chooser declaration has this form:

!choose chooserSetName? (L OP R)*

that is, the directive !choose followed by an optional name followed by zero or more choosers, where: L and R are set expressions over grammar elements and the operators — and () representing union, difference and do-first respectively Grammar elements may be terminals, nonterminals, productions and slots, or one of these keywords: anyLiteralTerminal, anyBuiltinTerminal, anyParaterminal, anyTerminal OP is one of ¿¿ (longer) ¡¡ (shorter) ¿ (L higher than R) ¡ (R higher than L) chooserSetName is an optional named set. If chooserSetName is omitted, the set with the empty name "" is selected

In the grammarPrint phase, chooser relations are computed, and the individual written out to files names ARTChooseXyz.art where Xyz is a chooser set name. Since the default name is empty, the unnamed chooser set is written to ARTChoose.art

The batch files lexGLL and parseMGLL expect to find files ARTChooseLexTWE.art, ARTChooseParseSPPF.art and ARTChooseParseTWE.art, so specifications to be used with these batch files must include !choose directive for those three names. This is only a convention, and the names could be changed if the batch files were changed to match.

The relations are written into the generated lexer and parser, so any changes to them will require regeneration. However, ambiguity reduction is only actually enabled by the directives !tweLongest, !twePriority, !sppfLongest and !sppfPriority applied to the corresponding ARTV3TestGenerated instance.

Rewrite rules

A rewrite rules has the form conditions — transition

Notes on V3 and V4 ==========

The notes above are for V4.Version 3 does not have the pipeline and associated directive list. Instead, ART is activated once for each

Top level batch files ——————

1. grammarWrite ¡spec¿output ART specification files derived from ¡spec¿: ARTParserGrammar.art, ARTLexerGrammar.art, ARTChoose*.art, ARTCharacterGrammar.art, ARTPrettyGrammar.art, ARTTokenGrammar.art

2. lexGLL ¡inp¿use EBNF GLL to lexicalise the contents of ¡inp¿ and dump the TWE set to file ARTTWE.twe

3. parseMGLL ¡inp¿use BNF-only MGLL to parse ARTTWE.twe - argument ¡inp¿ identical to the lexGLL run that built ARTTWE.twe

4. validateTokens ¡spec¿ ¡inp¿run steps 1-3 and then perform a token level parse using a lexicalisation derived from the SPPF

5. cleanremoves intermediate files created by steps 1-4

Utility batch files used by the top level batch files ——————————————— artRun ART V4 artV3Run ARTV3 artV3CompileGeneratedRun the Java compiler on the most recently produced ARTGeneratedLexer.java and ARTGeneratedParser.java artV3TestGeneratedExercise the most recently compiled ARTGeneratedParser and ARTGeneratedLexer

B Laboratory materials

This book introduces a range of techniques and formalisms. In this section, we provide a series of tutorial guides and examples that use the ART tool to implement small languages. To use these lab scripts you need to fetch an up-to-date version of `SLELabs.zip`. When unpacked, you will have a new directory `SLELabs` which contains the ART system itself in `art.jar` along with this series of subdirectories.

`ART` User manuals for the ART system.

`Solid` Examples of `solid modelling`

`TermRewriting` Examples for the `termTool` application

`SOS` A set of graduated examples illustrating the use of eSOS rules.

`Syntax` Simple parsing and syntax directed translation.

`Attributes` A set of languages and their interpreters specified using ART's attribute evaluation notations, and some examples of the use of GIFT rewrites to produce inner syntax trees.

`ProjectMaster` A 'project in miniature' which illustrates the various stages required for the project specified in Appendix C

`ProjectWork` A clone of ProjectMaster that you can use as the basis of your own project language.

ART is written in Java, so you will need a suitable Java runtime or development kit. Use a search engine to find an up to date install page for your system. It is a good idea to check the Wikipedia page that outlines Java's version history to ensure that you are using a recent version: some obsolete offerings are still online

Helper scripts are provided to reduce the amount you have to type when running ART. On Windows systems these appear as batch files with filename suffix `.bat`. On Un*x based operating systems such as Linux and MacOS, the shell scripts have filename suffix `.sh`; *you may need to make these shell scripts executable before running them for the first time by typing* `chmod +x *.sh`

B.1 Domain Specific Languages for solid modelling

This laboratory session aims to help you understand the users' view of Domain Specific Languages. We shall look at the domain of solid modelling languages which may be used, amongst other things, to create objects suitable for 3D printing.

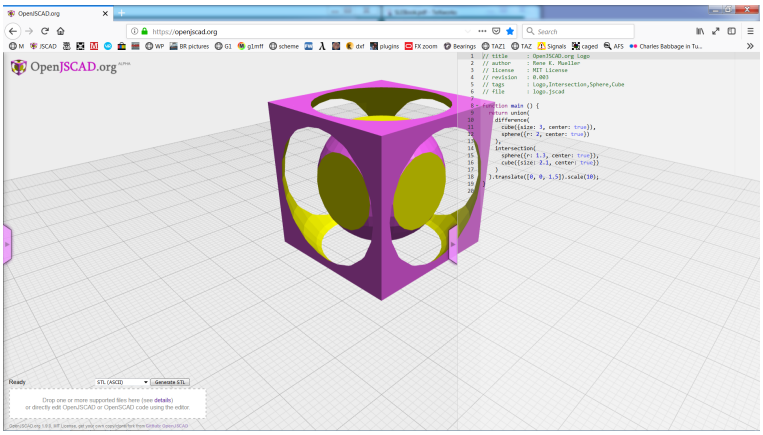
We shall use an online version of the OpenJSCAD language. Other interesting systems which address the same domain include OpenSCAD (which was the inspiration for OpenJSCAD), the ray tracer PoVRay and the animation tool Blender, which uses Python for scripting.

You will find most of the materials you need in subdirectory `Solid`. The OpenJSCAD tool itself runs in your browser, which you should point at

`https://openjscad.azurewebsites.net/`

. Note that there are other sites running versions of OpenJScad: to use this tutorial you open this specific site.

Your browser should show something like this:



The window contains a text editing area to the right which floats above a display showing the most recently rendered program.

B.1.1 Changing the view

Click on the displayed object outside of the text area and drag the mouse pointer around. You will find that you can rotate the object around the origin.

If you try pressing the shift key and then dragging, you will be able to pan the view across the screen. If you try pressing the ctrl key and then dragging, you will zoom in and out. (You can also use the middle mouse wheel to zoom if you have one.)

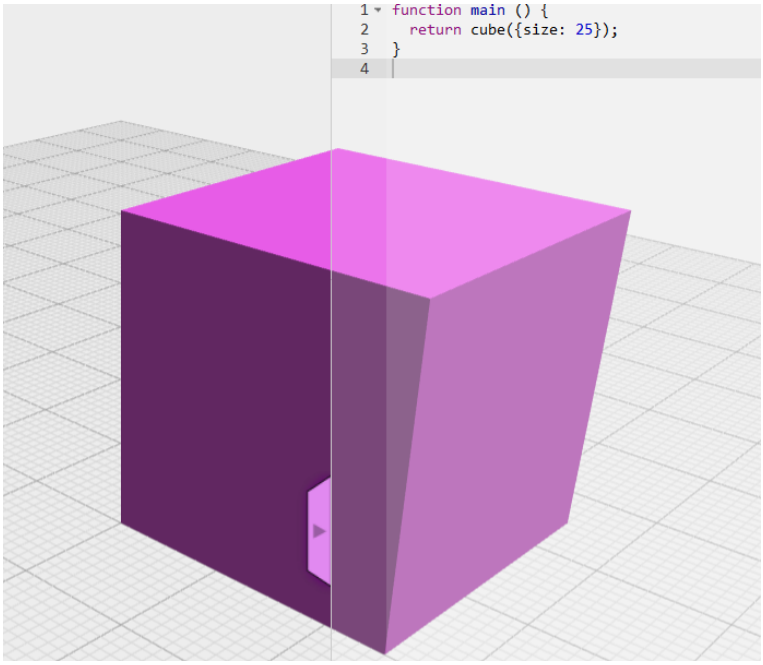
There is a pink tab on the left hand side of the window that opens some documentation.

B.1.2 A first object

Delete all of the text in the editor window, and replace it with the following code. (You will find the source code for these examples in the `SOLID` subdirectory.)

```
function main () {
  return cube({size: 25});
}
```

You *render* the code by pressing shift-return. This specification asks for a cube with 25mm edges. When rendered you should see this:

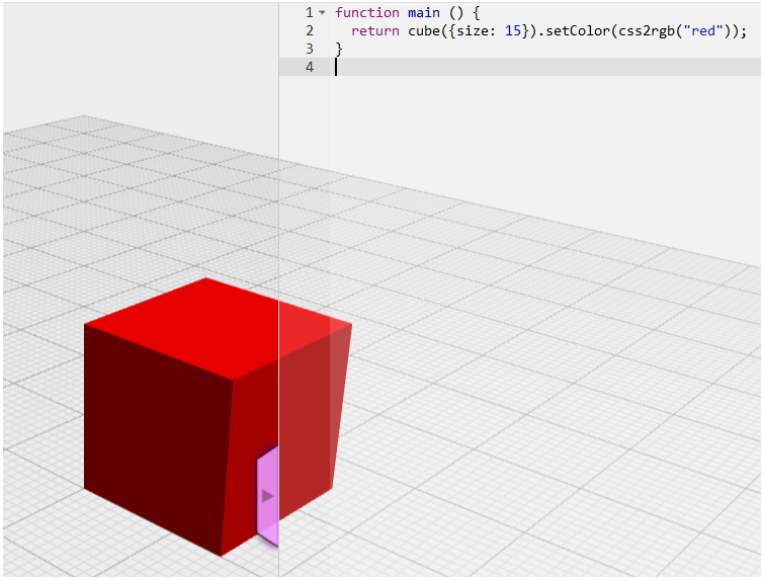


B.1.3 Changing size and color

The size argument sets the length of the cube's edge. Change it to 15 and rerender: the cube will become smaller. You can set the colour of objects using the .setColor() method which takes three arguments in the range 0–1.0 representing the proportion of red, green and blue (RGB) in the colour. Modify your code by adding this setColor call:

```
function main () {
  return cube({size: 15}).setColor(1,0,0);
}
```

The rendered output should look like:

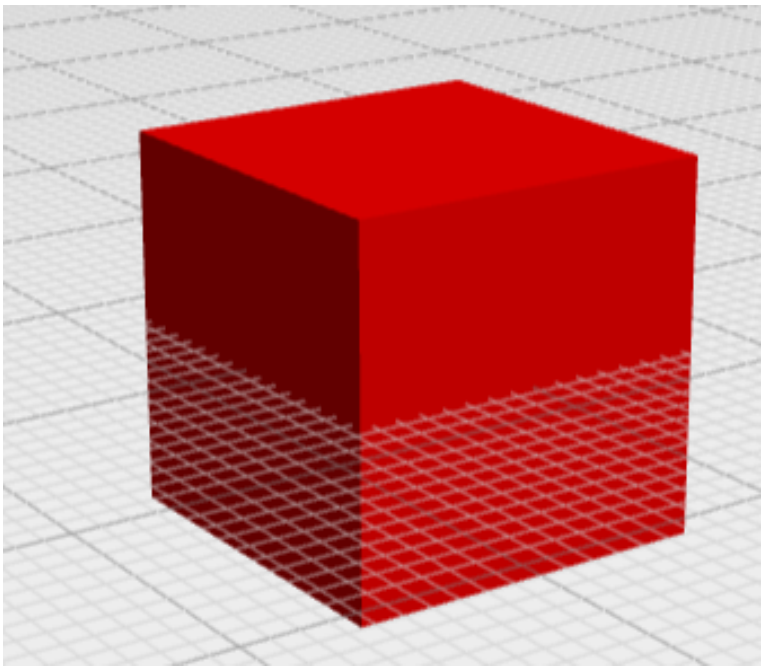


B.1.4 Where is the centre?

Rotations are specified around the origin, and as a result it is useful to ensure that all objects are created at the origin so that they can be re-oriented before being moved to their final position. Some objects like spheres are centered by default, but cubes are not. However, we can add an argument to force centring:

```
function main () {
  return cube({size: 15, center: true}).setColor(1,0,0);
}
```

Notice how the rendered object is centered in all three axes: the coordinate origin is at the centre of the cube.

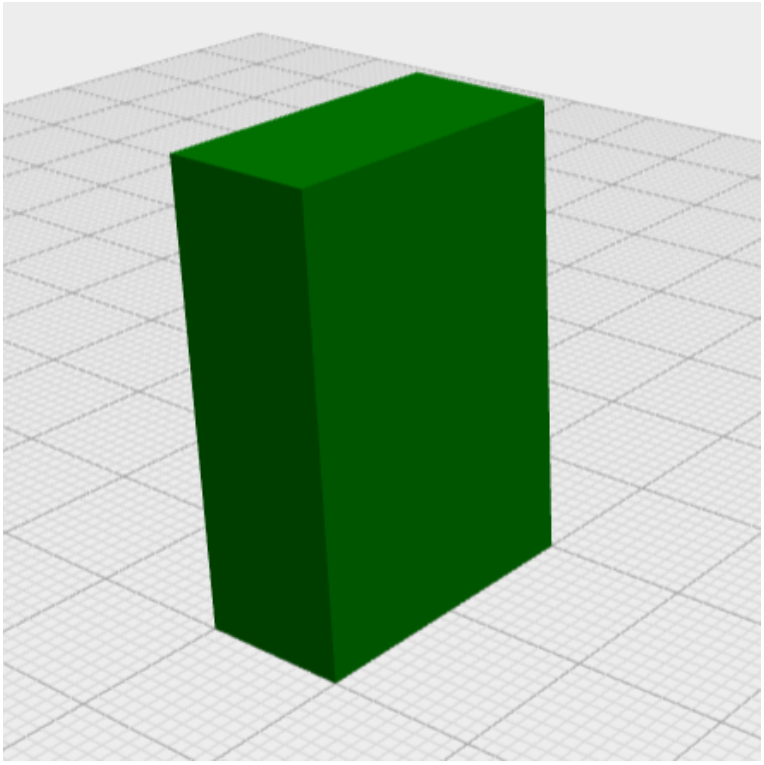


B.1.5 Cubes are really cuboids

The `cube` primitive can be used to specify cuboids, that is objects with varying x, y and z edge lengths.

```
function main () {
  return cube({size: [10,20,30]}).setColor(0,1,0);
}
```

The sequence of three numbers within square brackets is a *vector* and may be used to specify the three coordinates. Actually, just using a single integer, say `13`, is taken to be shorthand for `[ 13, 13, 13 ]`.

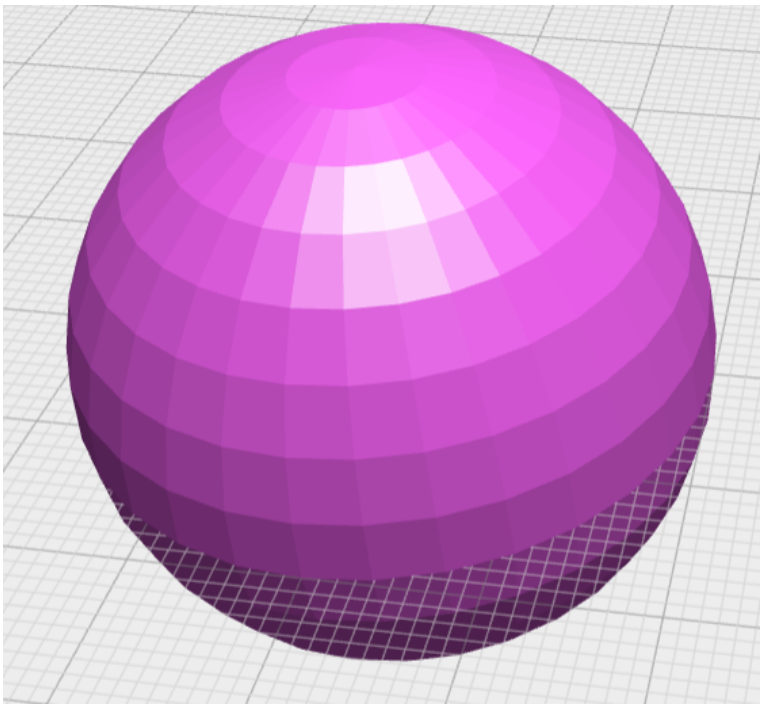


B.1.6 Spheres

JSCAD uses the *triangle mesh* method of representing objects: in reality all of the objects are rendered as flat triangles.

We can model spheres as polyhedra with sufficient faces to make the surface look smooth. The default for a sphere is only 32, which looks blocky if the sphere is large. The `sphere()` primitive constructs a spherical mesh, with radius specified by an argument called `r` and the number of facets around the equator and poles specified by argument called `fn`.

```
function main () {
  return sphere({r: 20, fn:32});
}
```

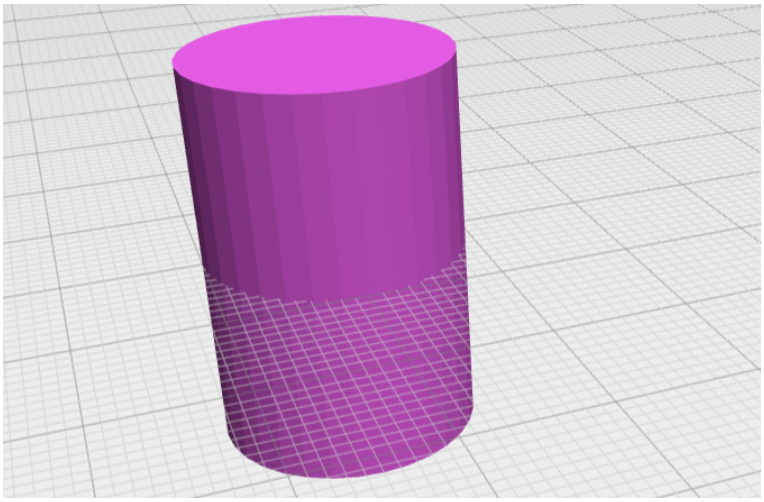


Try raising the value of `fn` to 120, and then to 360. You will see that the sphere gets much smoother, but you'll also notice that the rendering time increases.

B.1.7 Cylinders

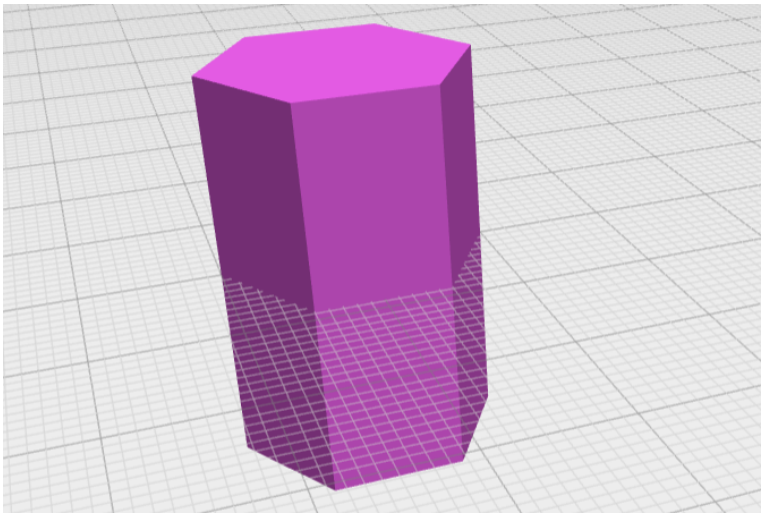
The `cylinder()` primitive takes a radius `r` and a height `h`.

```
function main () {
  return cylinder({r: 10, h: 30, center: true});
}
```



Now really, the cylinder primitive is just extruding a polygon. As before, we can use the `fn` argument to make a cylinder smoother, or we can make simpler objects. For instance, if we want to make an hexagonal bar, we can set it to six:

```
function main () {
  return cylinder({r: 10, h: 30, center: true, fn:6});
}
```



We could in fact make many kinds of box this way too. Is there a `cylinder()` equivalent for every `cube()`?

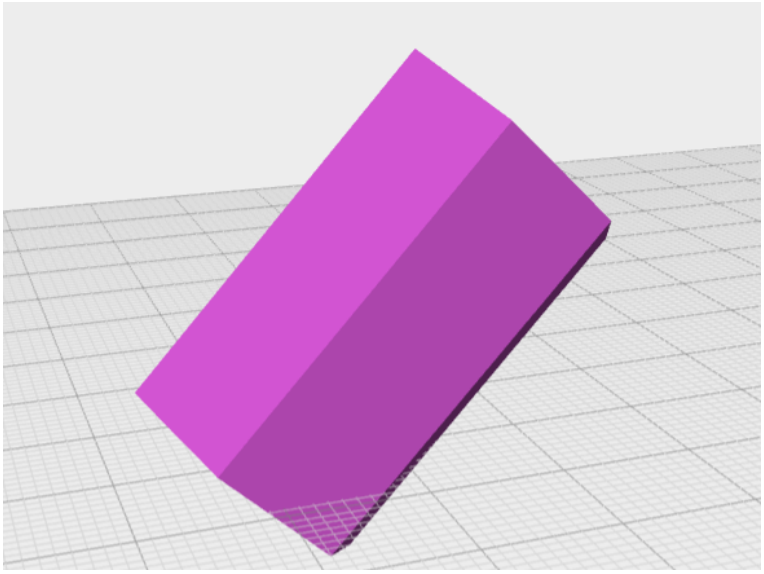
B.1.8 Translation and rotation

So far, we have made objects at the coordinate origin. We can move an object in space using the `translate()` method which takes as an argument a vector: three values within square brackets corresponding to the x, y and z displacements.

For instance, changing the previous example to

```
function main () {
  return cylinder({r: 10, h: 30, center: true, fn:6}).rotateY(45).translate([10,10,10]);
}
```

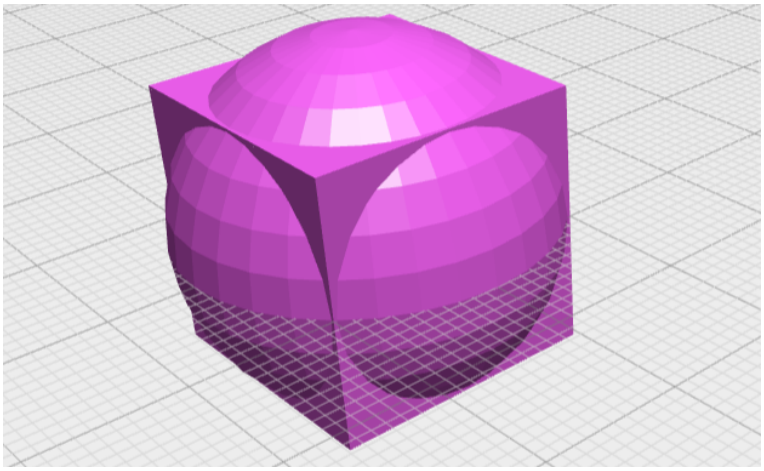
moves the hexagonal rod so that its centre is at $(x, y, z(=) = (10, 10, 10)$ and so that is tilted 45 degrees around the y axis.



B.1.9 Multiple objects

In JSCAD, functions return a single result. We need some way of constructing scenes with more than one object in, though. The answer is to make a new mesh which is the union of two other meshes.

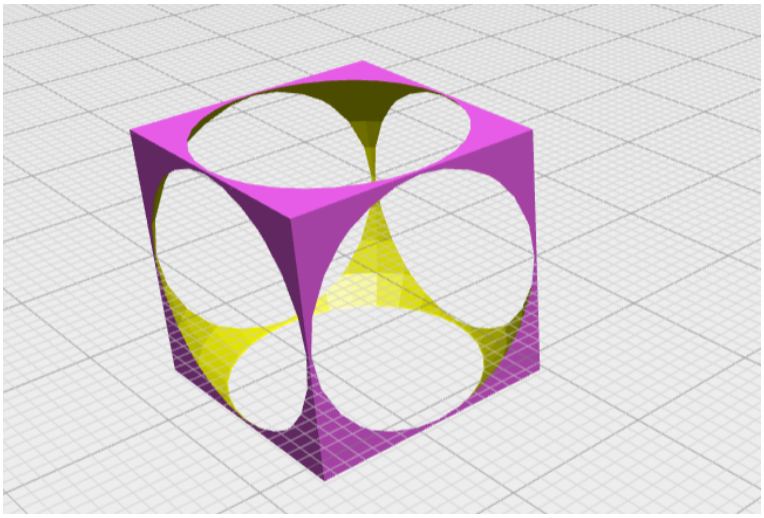
```
function main () {
  return union(
    cube({size: 20, center: true}),
    sphere({r: 14, center: true})
  )
}
```



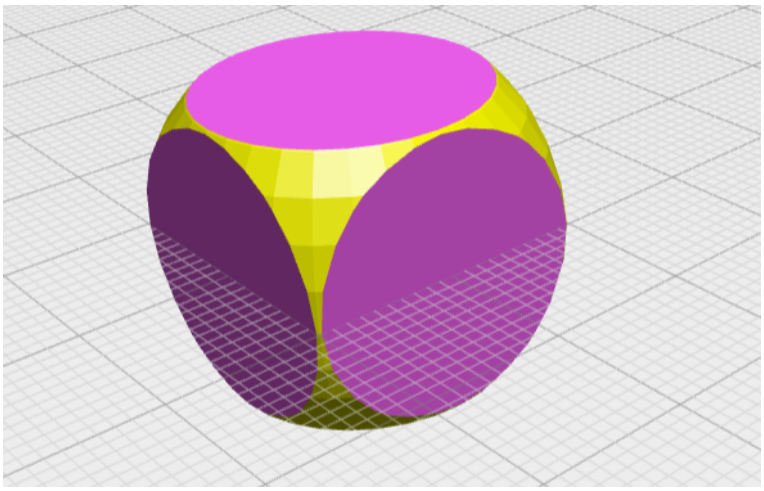
B.1.10 Computational solid geometry

Computational Solid Geometry (CSG) is a technique for making new objects from old via the operations of `union`, `difference` and `intersection`. We met `union` in the previous example. Here the equivalent examples for difference and intersection.

```
function main () {
  return difference(
    cube({size: 20, center: true}),
    sphere({r: 14, center: true})
  )
}
```



```
function main () {
  return intersection(
    cube({size: 20, center: true}),
    sphere({r: 14, center: true})
  )
}
```



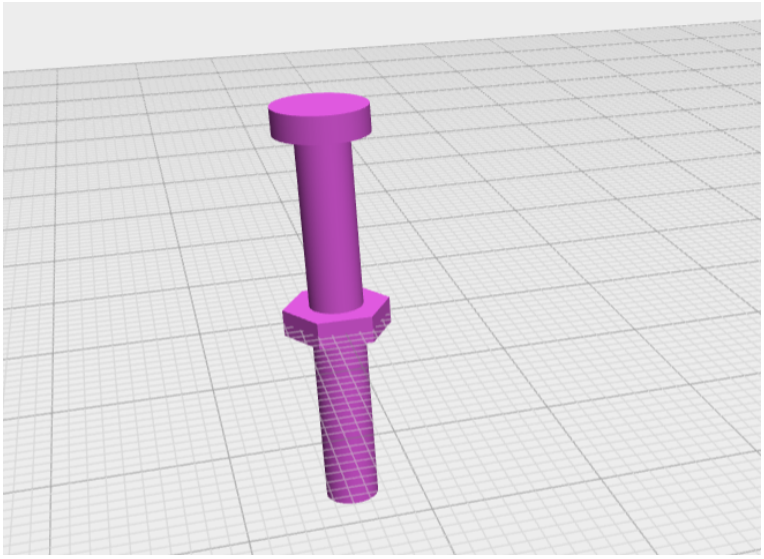
The `difference()` operation is widely used to make holes in objects by subtracting a cylinder from them.

B.1.11 Using functions to structure a design

We can use the full facilities of Javascript in our JSCAD specifications, because JSCAD is simply a Javascript library. (This style of Domain Specific Language is called an *internal* DSL.)

We write functions that return meshes, and then combine them together in the calling function. We can also use function arguments to parametrise our meshes. In this example, we make a rough model of an hexagonal nut and a cylindrical bolt which are combined together in the `main()` function. The `bolt()` function takes an argument `length` which specifies how long the bolt should be.

```
function nut () {
    return cylinder({r:4, h:2, fn:6, center:true});
}

function bolt (length) {
    return union (
        cylinder({r:2, h:length, fn:200, center:true}),
        cylinder({r:3.5, h:2, fn:200, center:true}).translate([0,0,length/2])
    );
}

function main () {
  return union(
    nut(),
    bolt(30)
  )
}
```



B.1.12 Internal and external Domain Specific Languages

A Domain Specific Language has some set of facilities (for instance special datatypes or special operations) over and above a general purpose language like Java or JavaScript. DSLs are also often limited compared to general purpose languages - we call these *little* DSLs.

In this lab, the main special data type is a *mesh*, that is an array of triangles arranged in space that we use to represent three dimensional objects. There are other special types hiding in here too: colors for instance. The special operations include things like *translation* and *rotation*, and CSG operations like *union* and *difference*.

Sometimes we just make a Domain Specific Library and access it from a general purpose language: that is an *internal Domain Specific Language*. OpenJSCAD is an example of an internal DSL that uses JavaScript as its host.

Sometimes we make up a completely new syntax and build complete, self contained language processors: we call that an *external* DSL.

OpenSCAD (`https://openscad.org/`) is an external DSL that addresses the same domain as OpenJSCAD: in fact OpenSCAD is the original tool, and OpenJSCAD is essentially a JavaScript addon to replicate OpenSCAD. You might like to install OpenSCAD and try it out. It turns out that OpenSCAD is a *little* DSL in that it has very limited facilities for input and output as well as a host of other restrictions.

B.1.13 Signatures and internal syntax

We need some way to concisely list the features of a language. In this course we make heavy use of *signatures* which you can think of as function definitions that name individual capabilities of a language. For instance, we describe integer addition as

```
add(_l: __int32, _r: __int32): __int32
```

This says that there is an operation called add which has two parameters called `_l` and `_r` (for left and right) both of which are constrained to be of type `__int32` (that is a 32-bit integer); after the close parenthesis there is another type constraint which tells us the type of the result computed by this add which is also a 32-bit integer.

The ART tool that we use on this course has a set of builtin type names which start with two underscores. ART also understand *metavariables* which have a single leading underscore, and are used as placeholders for arbitrary expressions. The type constraints say that those arbitrary expressions must reduce to particular types for this signature to become active.

Noe how the signature above *only* describes addition of 32-bit integers. You need a different signature for real-number addition and indeed for any other sized integers, so the list can grow rather quickly. Also, what about mixed mode arithmetic? Well the usual approach is to define a *cast* signature as a separate operation which reduces the overall number of signatures needed (why?)

Here are two other examples:

```
union(_l: __mesh, _r: __mesh): __mesh

translate(_m: __mesh, _x: __real64, _y: __real64, _z: __real64) : __mesh
```

The `union` operation takes two expressions that yield a *mesh* of triangles (type `__mesh`) and yields another mesh.

The `translate` operation takes a mesh and three expressions which each yield a 64-bit real number, and returns a mesh.

Note how these signatures are just describing the name, arguments and return type of these operations and not actually saying what they do - though the name of the operation is a clue.

B.1.14 How to design a programming language

There is a natural tendency for programming language designers to start with syntax, that is the way that programs look to the programmer. That can be useful for inspiration in the early stages, but in general is a Very Bad Idea.

A much better approach is to start of by thinking about the types of data that we want to manipulate and the operations that we want to perform. Signatures are our way of doing this. We call the complete set of signatures for a language the *internal syntax* of that language. Once we have the internal syntax, we can write rules or little programs that implement the actual actions associated with those operation, and that gives us a semantic interpreter which will take expressions composed from those signatures and execute them. When we are happy with that, we can design an *external syntax* for the language where, for instance addition is represented as $3 + 4$. We then need a *parser* for that external syntax that builds expressions in the internal syntax which can be handed to the semantic interpreter. This is the approach that we shall follow on this course.

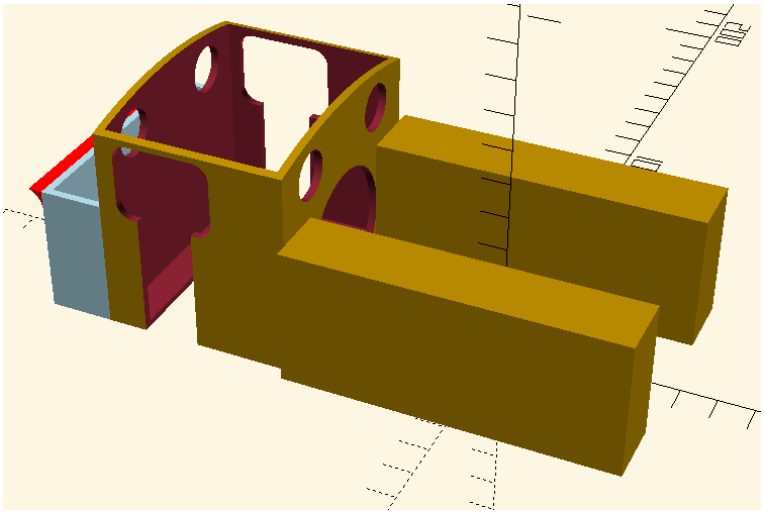
B.1.15 Your exercises

A: Write signatures for my examples

Go through the examples above and list the operations and types that you find. Then write a set of signatures that describe those operations. Some, such as function definition and call, are likely to be quite hard for you to think about and it is OK at this stage to just write list the operation without having to decide what its signature should be. Some, like `rotate` are very similar to the signature examples in section B.1.12 above.

B: Make a 3D model of a steam engine

This is the body section from a simple model steam engine, designed in this case in OpenSCAD rather than OpenJSCAD.

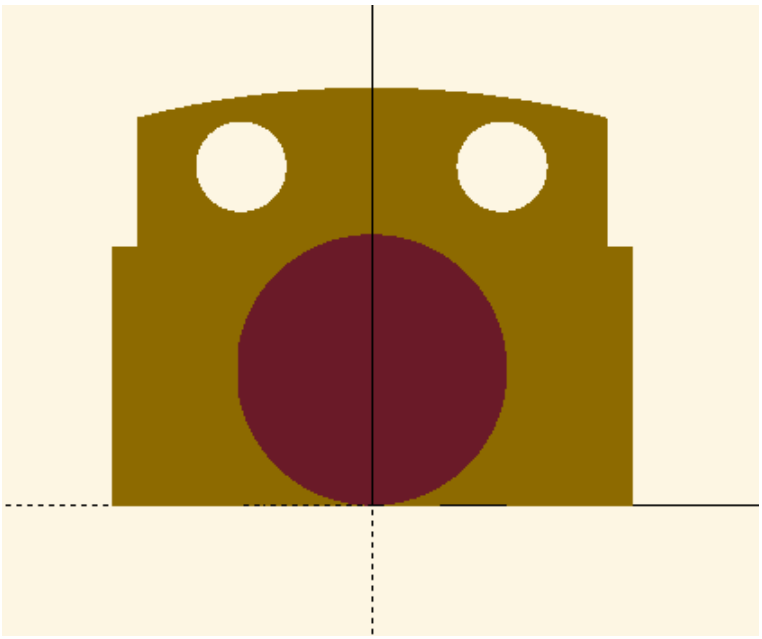


It is based on this prototype, real world engine. (The engine is a member of the J69 class, and you can read more about them at `https://www.lner.info/locos/J/j67j69.php`.)

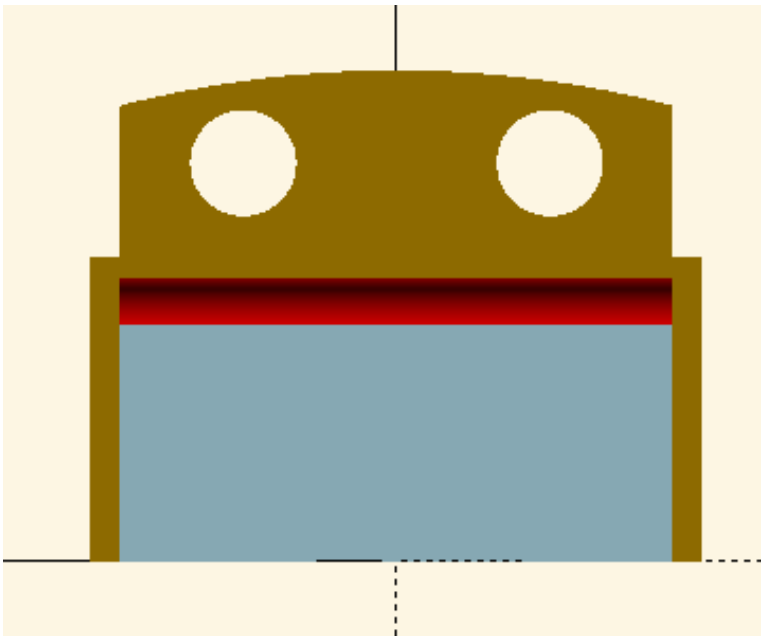
Your task now is to build your own version of the simple steam engine body, and if you are so inclined to embellish it so that it captures more of the prototype. Begin by defining the tanks, which are simple cuboids, and then make the cab as a cuboid which has another cuboid subtracted from it to make a hollow box. You can then subtract cylinders from it to make the round windows.

Here are reference views of the model from various angles.

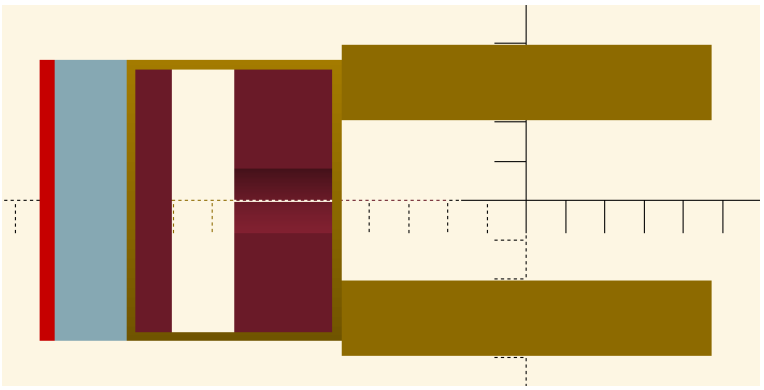
Front



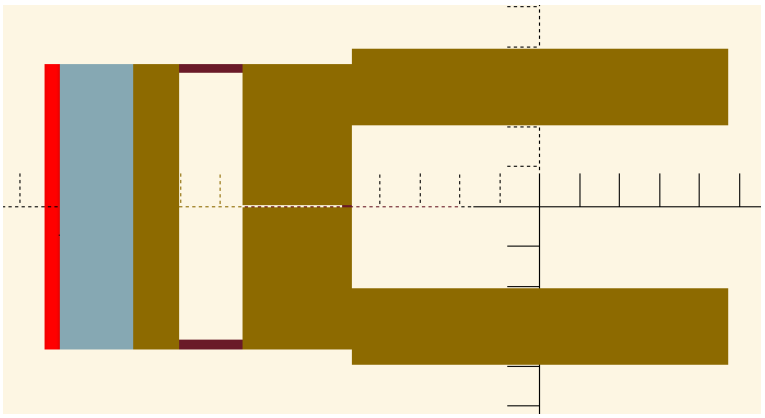
Rear



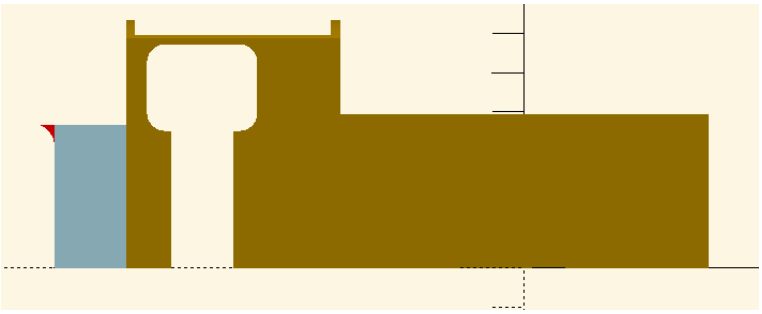
Above



Below



Side



When you have something you are pleased with, take a screenshot and then submit your work via Moodle.

C: Write signatures for your submission

Repeat the signature writing exercise, but this time go through your own model's code, adding in any new signatures that are needed.

B.2 Terms rewriting basics with TermTool

In this section we shall look at terms written in a prefix syntax, and at the use of pattern matching and substitution to make new terms. We shall also use ART's `Value` classes to perform some computations via terms. You should review the material in Chapter 2.4 before starting.

TermTool is a line-oriented interpreter — you type a line, and after pressing return TermTool will execute that line and return to the prompt. Start the TermTool interpreter by typing `java -jar ../art.jar !termTool` Note that the capitalisation is important.

TermTool will introduce itself and then leave the cursor at a `>` prompt, awaiting your input.

```
TermTool: type !?<return> for help

>
```

Input lines in TermTool are of two kinds: *commands* which begin with a `!` character and *expressions* which do not begin with a `!` character.

B.2.1 Getting help

The first command to try is `!?` which prints a summary of TermTool's features.

```
TermTool

Command lines have a ! character in column 1
!?              Help (this message)
!#              Show variables
!S              Show table statistics
!>              Show tables
!> filename     Output tables to a dump file
!< filename     Update tables from a dump file
!-              Delete contents of tables

!@ filename     Read commands from file
!.              Exit

Expression examples
A(B, c)
#X := A(B, c)
#Y := A(B, c) |> A(_1, _)
#Z := A(B, c) |> A(_1, _)  <| P(_1, _1)
#P := #Y <| P(_1, _1)
#Q := #Y
#Q += A(B,c) |> A(_2, _3)
#Z := A |> A
#Z <| __add(__int32(10), __int32(4))
```

B.2.2 Exiting TermTool

The `!.` command exits TermTool and returns you to the operating system prompt.

B.2.3 Expressions

Now try typing an expression. Expressions are made up of terms which may be combined with match and substitute operators, and whose result may be remembered in *TermTool variables*.

A term on its own is a valid expression, so at the TermTool prompt enter `A(B , c)` and press return, giving

```
> A(B , c)
A(B, c)
>
```

TermTool has read the line you typed, and echoed back the result of evaluating it. A term on its own (as here) just evaluates to itself, but note that the spacing has been standardised with no spaces before and one space after the comma. You can use as many space characters as pleases you, but TermTool will always echo terms in their canonical form, using standard spacing.

B.2.4 TermTool variables

It can be tiresome to have to repeatedly type in long terms, so TermTool maintains a set of *tool variables* which can be set to the result of any expression. These names of these tool variables always start with a `#` character, and can continue with mixture of alphabetic characters and digits. The `:=` operator assigns the result of an expression to a tool variable.

```
> #X := A(B, c)
A(B, c)
>
```

The echoed result is the expression that was assigned, but that value will now be remembered. You can get a list of the current tool variables and their values using the `!#` command.

```
> !#
#X = A(B, c)
>
```

A variable name alone evaluates to its assigned value, so if you just want to know the value of a single variable you can type it on its own

```
> #X:=A(B,c)
A(B, c)
> #X
A(B, c)
>
```

You can delete a tool variable by assigning nothing to it.

```
> #x := a(b,c)
a(b, c)
> !#
#x = a(b, c)
> #x:=
a(b, c)
> !#
No TermTool variables defined
>
```

The expression `#x:=` looks up the value assigned to x, but deletes the variable `x` and returns the value that had been assigned to it.

B.2.5 Matching with the ▷ operator

The *match* operator ▷ compares two terms, starting at the outermost symbol and then recursively descending into their subterms. The whole term is checked and an error message is printed if the match fails.

If the match succeeds, TermTool prints out the result which in this example is simply an empty set.

```
> A(b,c) |> A(b, c)
{ }
>
```

Equality of symbols

Names in terms are just sequences of characters; they have no associated meaning, and matching operates over those simple character sequences. Therefore you shouldn't make assumptions about common conventions. For instance, in arithmetic 023 would be the same value as 23, but here `023` is a string three characters long and `23` is a different string of two characters. Thus they are not equal and will not match!

Although names can have any character in them, you may not use space characters or the following characters directly since TermTool uses them for itself - we say they are *metasymbols* in the TermTool language.

```
! :   =   +   |   <   >   #   _   \
```

All is not lost though because TermTool uses an *escape convention* in which a backslash `\` followed by some character evaluates to that character, so in fact we have two syntactic forms for most characters. For instance, if we wanted to have a name that started with a `!` that would conflict with TermTool's use of `!` to introduce commands, but we can get round that with an escape sequence. There are three exceptions: as in Java, the sequences `\n` `\t` and `\r` stand for newline, tab and carriage return respectively, but we wouldn't recommend using the in term names.

B.2.6 Pattern matching and term variables

As you will have read in Chapter 2.4, terms can include *term variables*. These act as placeholders in the term against which an arbitrary subterm may be substituted. Alternatively we can view them as holes in a term which are filled in during a pattern match. We shall use a combination of pattern matching and substitution to transform terms.

In TermTool, term variable names begin with an underscore character _ and continue with an integer number that must be greater than zero. There is an limit on the number of term variables that are allowed in each term, typically 16, but a fresh set of variables is available for each term. (We should note that formally there is no limit to the number of variables in a term, but our proposed hardware implementation does require a small upper bound to avoid wasting hardware resources.)

Here is a first example. As before, the match operator ▷ checks that the terms are identical but when it arrives at a right-hand-side variable, it creates a *binding* from that variable to the corresponding subterm on the left hand side. The result of a match is then the set of bindings from variables to subterms that were found during the complete match.

```
> A(B, c) |> A(_1, c)
{ _1->B }
>
```

So, the position of variable `_1` on the right corresponds to the subterm `B` on the left, and so the binding `_1->B` is added to the result of evaluating the expression.

There are many possible formulations for pattern matching, some of which can be very expensive to compute. The TermTool match operation ▷ imposes several restrictions on the form of patterns so as to provide efficient, unambiguous pattern matching. Recall that a term that has no term variables within it is called a *closed* term. A term that has one or more term variables within it is called an open term.

1. The left operand must be a closed term.
2. The right operand may be a closed term or an open term with these restrictions.
   (a) In the right operand, term variables may only appear as leaf nodes in the term tree, that is term variables must have arity zero.
   (b) In the right operand, a term variable may only appear once in a term.

The don't care variable

Sometimes we just want to ensure that a term has the right number of subterms, and do not need to remember their actual values. A single underscore, with no following number matches in the usual way, but nothing is added to the binding set.

```
> A(B, c) |> A(_1, _2)
{ _1->B _2->c }
> A(B, c) |> A(_, _2)
{ _2->c }
>
```

B.2.7 Term variables and tool variables

It is important not to confuse term variables, which hold subterms from matching, and tool variables which hold the results of complete expressions. In TermTool we emphasise the difference syntactically — tool variables begin with a `#` character, and term variables begin with an underscore `_` character. Tool variables can have alphanumeric names, but the names of term variables are numeric, and there is an upper limit on how many there can be. The scope of a term variable is limited to the term it appears within whereas tool variables continue to exist until TermTool exits.

Tool variables can be used to hold the results of matches, and this is useful because we often want to extract some subtrees with a match and then use them in several other terms. In this example we

```
> #Y := A(B, c) |> A(_1, _)
{ _1->B }
> !#
#Y = { _1->B }
>
```

B.2.8 Extending bindings with the union-into += operator

We sometimes want to build up sets of bindings from several matches. We can extend the set of bindings held in a tool variable by using the union-into `+=` operator instead of the assignment operator `:=`. This operator adds bindings into an existing set.

In this example, we first create the set of bindings `{ _1->b _2->C }` and assign it to `#X`. We then extend `#X` with `{ _5->q }` to give `{ _1->b _2->C _5->q }`.

```
> #X := A(b, C) |> A(_1, _2)
{ _1->b _2->C }
> !#
#X = { _1->b _2->C }
> #X += P(q,r) |> P(_5,_)
{ _5->q }
> #X
{ _1->b _2->C _5->q }
>
```

Use of the `=+` operator is quite constrained. First, the left hand side may not be a variable that is bound to a term, because it makes no sense to take the union of two terms, or of a term and a set of bindings. Secondly, the bindings

in the left and right hand sides must not *collide*, that is no term variable may appear in both the left and right sets. If they did, we would have to decide which one to keep after the union operation and which one to throw away, and thus information could then be lost.

If the left hand side of a `+=` operator was previously undefined, then it is created as an empty set of bindings. Effectively the `+=` operator degenerates to `:=` for fresh tool variables.

B.2.9 Using tool variables in expressions

The purpose of tool variables is to save typing. If a tool variable contains a term, it may be used anywhere a term may be.

```
> #A := A(b,c)
A(b, c)
> #B := A(_1,_)
A(_1, _)
>
> #A |> #B
{ _1->b }
>
```

If a tool variable contains a set of bindings, it may be used on the right hand side of side of a substitution. In fact this is the standard way of specifying substitutions since you cannot write a binding set directly: it first must be constructed using a match operation.

B.2.10 Substitution and unconditional rewrites

The pattern matching operator $\triangleright$ extracts subterms from terms and returns a (possibly empty) set of bindings. The substitution operator $\triangleleft$ takes a term that may include term variables and a set of bindings, and builds a new term from them.

For instance, if we wanted to rewrite the term `A(b, c)` into `A(c, b)` (that it, swap the children over) we could write

```
> A(b,c) |> A(_1, _2) <| A(_2, _1)
A(c, b)
>
```

The closed term `A(b,c)` is matched against open term `A(_1, _2)`, yielding the bindings `{ _1->b _2->c }`. These are then substituted into `A(_2, _1)` to give `A(c, b)`.

Note that this is a direct implementation of the unconditional rewrite

$$A(X, Y) \rightsquigarrow A(Y, X)$$

In general, if you have an unconditional rewrite

$$L \rightsquigarrow R$$

that you want to apply to term T, then you write $T \triangleright L \triangleleft R$, or in TermTool script

```
> T |>  L  <| R
```

It is an error for the open term on the right of a $\triangleright$ substitution operator to contain a term variable that is not in the set of bindings provided as a left operand. It is *not* an error for there to be bindings to term variables that do not appear in the right hand operand. It is (of course) an error for the wildcard variable `_` to appear in the term on the right hand side of a substitute operator (`<|`) since we would not know what to substitute in at that point.

B.2.11 Evaluation of functions during substitution

Simply using pattern matching and substitution allows us to break up and rebuild terms, and in fact we can, with care, perform arbitrary computations by, for instance, implementing the untyped lambda calculus. However, something as simple as addition of integers can require very lengthy sequences of rewrites and terms whose size is proportional to the integers being manipulated. Neither of these is very comfortable for practical computation. Our terms may therefore include some built in functions that take terms and return terms, and which are automatically evaluated as part of the substitution mechanism.

Functions have names beginning with two underscores, such as `__add` and `__multiply`. The underscores remind us that they are a bit like special term variables during substitution, and the name is meant to indicate the function that is computed.

We said early that the names in terms were simply strings of characters, and that a name like `0123` was distinct from a name like `123`. That is true for pattern matching and substitution, but the built in functions *do* provide an interpretation of these names, and will fail if you call them with the wrong kinds of subterms.

The full gamut of operations is described in Chapter **??**, and as an aide memoir their names are preloaded into the string table and may be examined within a session using the `!>` command.

Since functions are only evaluated during substitution, we need to perform a substitution to see any effects. It is sometimes useful to create an empty set of bindings and assign it to, say, variable `#Z`.

```
> #Z := A |> A
{ }
> #Z <| __add(__int32(10), __int32(4))
Substitute __add(__int32(10), __int32(4)) into { } returned __int32(14)
__int32(14)
```

B.3 SOS – An introduction to eSOS

We now turn from programming to *meta*-programming, that is the design and implementation of programming languages. These exercises use an implementation of the SOS style of formal semantics called *eSOS* (which stands for elided-SOS: the sense in which eSOS specifications are elided will be examined later).

The materials for this section reside in subdirectory `SLELabs/SOS` and comprise a sequence of example specifications with names of the form `Ln.art`.

To interpret specification `L1.art` on Windows, you issue the following command

```
..\art L1.art
```

To interpret specification `L1.art` on Un*x style operating systems, you issue the following command

```
../art.sh L1.art
```

The interpreter will display trace output on the console, and in addition create the file `artSpecification.tex` which contains the fully instantiated rules from the source file. If you have a running LaTeX system, you can produce a pretty-printed version of the rules in file `artSpecification.pdf` by typing:

```
pdflatex artSpecification
```

B.3.1 A first example

We shall design a programming language which can do only one thing: take the constant 3 and increment it to 4. Make a file `L1.art` containing this text which is a full specification of the language that we shall call $L1$.

```
---
increment(3) -> 4

!trace 1
!try increment(3)
!try increment(4)
```

Rules in eSOS comprise zero or more conditions followed by a line made up of three `-` characters followed by a single transition called the conclusion. A transition will be of the form *term relation-symbol term* where *relation-symbol* is defined in a `relation` directive, and the terms are trees written in the usual prefix-function style.

The single rule here is written

```
---
increment(3) -> 4
```

from which we can see that it has zero conditions and just a conclusion. These *unconditional* rules are called *axioms*.

The rule tells us that a program `increment(3)` will be rewritten in one step to the program `4`. The fact that this is the *only* rule and that it contains no term variables tells us that this is actually the only thing that our programming language can do.

We have three *directives*:

```
!trace 1
!try increment(3)
!try increment(4)
```

ART *directives* are all preceded by a shriek (`!`) which indicates that they are to be executed *procedurally*, as opposed to rules which are *declarative* in nature. Essentially directives make up a simple *script* that tells us how to exercise the rules.

A phrase like `!try increment(3)` simply specifies a term which eSOS will attempt to reduce using the default relation, that is the relation in the conclusion of the first rule in the file. The directive `!trace 1` sets the trace level for subsequent directives (up to the next `!trace`). The available trace levels are as follows:

0	silent
1	final result only
2	top level rewrites
3	all rewrites
4	conditions
5	bindings

These levels are cumulative: each level includes all the information from the levels with a lower trace number.

Now run ART on `L1.art` to produce this output:

```
*** try increment(3) with relation  ->
Normal termination on 4 after 2 steps and 2 rewrites
*** try increment(4) with relation  ->
Stuck on increment(4) after 1 step and 1 rewrite
```

This tells is that the first try terminated normally after rewriting to 4, but that the second try became stuck, which means that it could not find a matching rule.

If we change to `!trace 5` then we get much more voluminous output which tells us exactly what is going on:

```
*** try increment(3) with relation  ->
Step 1
  Rewrite call 1 increment(3)  ->
  -R1  --- increment(3) -> 4
  -R1 bindings after Theta match {  }
  -R1 rewrites to 4
Step 2
```

```
  Rewrite call 2 4  ->
    Terminal 4
Normal termination on 4 after 2 steps and 2 rewrites
*** try increment(4) with relation  ->
Step 1
  Rewrite call 1 increment(4)  ->
  -R1  --- increment(3) -> 4
  -R1  Theta match failed: seek another rule
  Failed rewrite call 1 increment(4) ->
Stuck on increment(4) after 1 step and 1 rewrite
```

We shall look in more detail at trace outputs in a later section.

B.3.2 Normal termination and stuck configurations

The eSOS interpreter terminates when it cannot find any rule with which it can reduce the program term. Now, of course some configurations are intended to represent *successful* termination, so some program terms are what we call *terminals*. In eSOS, numeric literals, strings and boolean true values true and false are all terminals, and there is a way to declare extra terminal terms: we shall see examples of this later.

If the interpreter stops and the final program term is a terminal term, then the interpreter reports `Normal termination`; otherwise the interpreter will report that it is 'stuck'.

In this example, all integer literals are automatically terminals, and so when the interpreter terminates with a program term which is just an integer it reports `Normal termination`. However, `increment(4)` is not a terminal, so the interpreter reports that it is stuck.

B.3.3 Generalising with term variables and functions

We shall now make a new programming language which can increment any integer. We shall still only need one eSOS 'rule', though when we were being careful we would say that the specification now has a single *rule schema* which induces a rule for every number, and a particular number will be represented by the name X which will label a term containing that number.

Make a file `L2.art` with these contents:

```
---
increment(_X) -> __add(_X,1)

!try increment(3)
!try increment(4)
```

When we run the interpreter on L2 we get this output:

```
*** try increment(3) with relation  ->
Step 1
```

```
  Rewrite call 1 increment(3)  ->
  -R1  --- increment(_X) -> __add(_X, 1)
  -R1 rewrites to 4
Step 2
  Rewrite call 2 4  ->
    Terminal 4
Normal termination on 4 after 2 steps and 2 rewrites
*** try increment(4) with relation  ->
Step 1
  Rewrite call 1 increment(4)  ->
  -R1  --- increment(_X) -> __add(_X, 1)
  -R1 rewrites to 5
Step 2
  Rewrite call 2 5  ->
    Terminal 5
Normal termination on 5 after 2 steps and 2 rewrites
```

Our language apparently works for both 3 and 4. Let's examine in detail what is happening.

We begin with a program term such as `increment(3)`. We look through the set of rules (there is only one in this case!) and try to match the term with the left hand side of a conclusion.

Our one rule has `increment(X))` as its left hand side. Now X in not a terminal, nor does X appear as the first element in any term so eSOS assumes that it is a term *variable*. That means it can match any subterm, including the subterm `3`. Thus the binding $X \mapsto 3$ is created.

Having matched the left hand side of the conclusion, we now proceed to the conditions. The only condition is `__add(1,X) |> Y`. We begin by substituting X by 3 since that is what is in the set of bindings to give `addOp(1,3) |> Y`. The two leading underscores in the constructor `__add` operator tells us that the term `__add(1,3)` must be a *function call*, that is an attempt to access a lookup table (or equivalent mechanism) which will return some term, which will then be matched against the term `Y`. eSOS has a fixed repertoire of such tables: this particular one has been loaded with numbers that match our expectation of adding numbers together: the number at coordinated $(3, 4)$ for instance would be 7.

Now that we have checked the conditions, adding to the set of bindings as we go, we can process the right hand side of the conclusion below the line by substituting and then deleting that part of the program term that was originally matched by the left hand side by the results and inserting the substituted right hand side. For the program `increment(3)`, Y will have been bound to 4, and the conclusion matched the whole program term, so the whole program will be deleted and replaced with 4.

B.3.4 Runtime type errors

If we add the try `!try increment("five")` to our file then we get a type error from the `__add` function

```
*** try increment("five") with relation  ->
Step 1
  Rewrite call 1 increment("five")  ->
  -R1  --- increment(_X) -> __add(_X, 1)
!! Function error: __add(__string,__int32) - operands must be of same type; retur
  -R1 rewrites to __bottom
Step 2
  Rewrite call 2 __bottom  ->
    Terminal __bottom
Normal termination on __bottom after 2 steps and 2 rewrites
```

This unhappy situation arises because `"five"` is a string, not a number, and whilst the eSOS interpreter is perfectly happy to match X to `"five"` and pass it to the `__add` function (lookup table), addition is not defined over strings, so we get a runtime type error.

The set of types and operations provided by eSOS is exactly the set of types and their operations described in Section **??** and summarised in the spreadsheet `sleLabs\ART\ARTTypeAndOperationTable.xlsx`

B.3.5 Filtering out type errors using conditions

All is not lost! We can introduce some type checking into our rules by adding extra conditions. Make a file `L3.art` containing:

```
_X |> __int32(_)
---
increment(_X) -> __add(_X, 1)

!try increment(3)
!try increment(4)
!try increment(5.5)
!try increment("five")
```

We can check that a variable is bound to a particular type with a pattern line `__int32(_)`. In this case, failure must lead to the interpreter getting stuck, because there are no more rules that can be tried.

When we interpret L3 we see

```
*** try increment(3) with relation  ->
Step 1
  Rewrite call 1 increment(3)  ->
  -R1 _X |> _ --- increment(_X) -> __add(_X, 1)
  -R1 rewrites to 4
Step 2
```

```
  Rewrite call 2 4  ->
    Terminal 4
Normal termination on 4 after 2 steps and 2 rewrites
*** try increment(4) with relation  ->
Step 1
  Rewrite call 1 increment(4)  ->
  -R1 _X |> _ --- increment(_X) -> __add(_X, 1)
  -R1 rewrites to 5
Step 2
  Rewrite call 2 5  ->
    Terminal 5
Normal termination on 5 after 2 steps and 2 rewrites
*** try increment(5.5) with relation  ->
Step 1
  Rewrite call 1 increment(5.5)  ->
  -R1 _X |> _ --- increment(_X) -> __add(_X, 1)
  Failed rewrite call 1 increment(5.5) ->
Stuck on increment(5.5) after 1 step and 1 rewrite
*** try increment("five") with relation  ->
Step 1
  Rewrite call 1 increment("five")  ->
  -R1 _X |> _ --- increment(_X) -> __add(_X, 1)
  Failed rewrite call 1 increment("five") ->
Stuck on increment("five") after 1 step and 1 rewrite
C:\adrian\teaching\softwareLanguageEngineering\2022\SLELabs\SOS>
```

Now, instead of the string argument causing a run time failure from the ARTValue system, the eSOS interpreter runs normally, terminates and then notices that the final term is not a terminal term, and so reports that it got stuck (in this case, because it found no reductions at all).

B.3.6 Generalising by adding rules

Unfortunately we've gone a little too far: L3 cannot increment real numbers. We can restore that capability by adding another rule which handles reals. Make a file `L4.art` containing

```
-integerInc
_X |> __int32(_)
---
increment(_X) -> __add(_X, 1)

-realInc
_X |> __real64(_)
---
increment(_X) -> __add(_X, 1.0)

!try increment(4)
```

```
!try increment(5.5)
!trace 5
!try increment(5.5)
```

We now have two rules, and we have given them names: you can attach a label to a rule in eSOS by prefacing it with a - character. The rule name must be a single integer, a single word or a single string. If you do not specify a label in this way, the eSOS interpreter will attach a numeric label to them in the order it finds the rules in the file. The labels are purely to help the reader and have no significance to the interpreter.

Recall that eSOS will search through its rule set seeking matches. Now, both test programs have a rule that will handle them. The interpreter gives:

```
*** try increment(4) with relation  ->
Step 1
  Rewrite call 1 increment(4)  ->
  -integerInc _X |> _ --- increment(_X) -> __add(_X, 1)
  -integerInc rewrites to 5
Step 2
  Rewrite call 2 5  ->
    Terminal 5
Normal termination on 5 after 2 steps and 2 rewrites
*** try increment(5.5) with relation  ->
Step 1
  Rewrite call 1 increment(5.5)  ->
  -integerInc _X |> _ --- increment(_X) -> __add(_X, 1)
  -realInc _X |> _ --- increment(_X) -> __add(_X, 1.0)
  -realInc rewrites to 6.5
Step 2
  Rewrite call 2 6.5  ->
    Terminal 6.5
Normal termination on 6.5 after 2 steps and 2 rewrites
```

B.3.7 Examining the behaviour of the interpreter in detail

When debugging specifications, it is sometimes helpful to examine the behaviour of the interpreter in detail as it backtracks through the rules. Recall that we can set the trace message level for the interpreter using the `!trace` ***n*** directive where ***n*** can be an integer from zero to five with these (cumulative) meanings:

0	silent
1	final result only
2	top level rewrites
3	all rewrites
4	conditions
5	bindings

The default level is 3, and that was the trace level used for the previous example.

Change the `!trace` level in `L4.esos` to be `!trace 5` which will show us all of the available detail.

Now when we run the interpreter we see

```
*** try increment(4) with relation  ->
Step 1
  Rewrite call 1 increment(4)  ->
  -integerInc _X |> _ --- increment(_X) -> __add(_X, 1)
  -integerInc bindings after Theta match { _X=4 }
  -integerInc premise 1 _X |> _
  -integerInc bindings after premise 1 { _X=4 }
  -integerInc rewrites to 5
Step 2
  Rewrite call 2 5  ->
    Terminal 5
Normal termination on 5 after 2 steps and 2 rewrites
*** try increment(5.5) with relation  ->
Step 1
  Rewrite call 1 increment(5.5)  ->
  -integerInc _X |> _ --- increment(_X) -> __add(_X, 1)
  -integerInc bindings after Theta match { _X=5.5 }
  -integerInc premise 1 _X |> _
  -integerInc premise 1 failed: seek another rule
  -realInc _X |> _ --- increment(_X) -> __add(_X, 1.0)
  -realInc bindings after Theta match { _X=5.5 }
  -realInc premise 1 _X |> _
  -realInc bindings after premise 1 { _X=5.5 }
  -realInc rewrites to 6.5
Step 2
  Rewrite call 2 6.5  ->
    Terminal 6.5
Normal termination on 6.5 after 2 steps and 2 rewrites
```

The line `Rewrite call 1 increment(4)  ->` tells us that whilst checking the first step, the interpreter has been called on relation `->` with a configuration comprising just the program term `increment(4)`. We then see that we are working with rule `integerInc` and that after matching to the program term, `_X` is bound to 4.

The second `!try` on `increment(5.5)` proceeds differently in that the interpreter matches the conclusion in rule 1, but then fails after testing the first premise. The interpeter then searches for another rule that matches the program term, finds `realInc` and proceeds to the rewrite.

From this we can see that the order of the rules significantly affects performance: integer increments will be faster than real increments because the interpreter find the integer rule first. Do bear in mind though that our goal is to produce compact specifications of programming languages that we can reason about, and that performance is secondary to that goal.

B.3.8 Addition of two values

It is easy to modify our incrementing language L4 to perform addition of two values. Make a file `L5.art` containing:

```
_X |> __int32(_)  _Y |> __int32(_)
---
add(_X,_Y) -> __add(_X, _Y)

!try add(3,4)
```

We are now using the constructor `add` instead of `increment`, and the pattern on the left hand side of the conclusion has two term variables in it, X and Y. The subterms bound to these term variables are checked to ensure that they are integers, and then they are used to access the `__add` lookup table (function). The interpreter output is:

```
*** try add(3, 4) with relation  ->
Step 1
  Rewrite call 1 add(3, 4)  ->
  -R1 _X |> _  _Y |> _ --- add(_X, _Y) -> __add(_X, _Y)
  -R1 rewrites to 7
Step 2
  Rewrite call 2 7  ->
    Terminal 7
Normal termination on 7 after 2 steps and 2 rewrites
```

B.3.9 Nested additions

In our outer syntax, we might want to be able to evaluate expressions containing more than one operator instance, such as $3+4+5$. These would naturally give rise to internal syntax terms like `add(add(3, 4), 5)`

Make a copy of `L5.art` called `L6.art` and then modify the `!try` directive to read

```
!try add(add(3,4),5)
```

The interpreter then gives:

```
*** try add(add(3, 4), 5) with relation  ->
Step 1
  Rewrite call 1 add(add(3, 4), 5)  ->
  -R1 _X |> _  _Y |> _ --- add(_X, _Y) -> __add(_X, _Y)
  -R1 bindings after Theta match { _X=add(3, 4), _Y=5 }
  -R1 premise 1 _X |> _
  -R1 premise 1 failed: seek another rule
  Failed rewrite call 1 add(add(3, 4), 5) ->
Stuck on add(add(3, 4), 5) after 1 step and 1 rewrite
```

We have got stuck because although the add rule matches the initial program term, term variable X is bound to `add(3,4)` which is not a number, so the first condition fails.

The solution is perhaps the most subtle part of an SOS-style specification. We add a rule which allow the arguments of an `add` constructor to be recursively shrunk down to operations that can be directly executed, and then as the recursion unwinds program is rewritten as the values are propagated.

Create a specification called `L7.art` containing these rules:

```
-add _X |> __int32(_)  _Y |> __int32(_)  __add(_X, _Y) |> _Z
---
add(_X,_Y) -> _Z

-addExpr
_E1 -> _V1 _E2 -> _V2
---
add(_E1, _E2) -> add(_V1, _V2)

!trace 4
!try add(add(3,4),5)
```

The original rule `add` from `L6.art` can, as we have seen only add arguments which can be directly accepted by the `__add(,)` function. The new rule `addExpr` will match any term which has a root node labelled `add`

```
*** try add(add(3, 4), 5) with relation  ->
Step 1
  Rewrite call 1 add(add(3, 4), 5)  ->
  -add _X |> _  _Y |> _  __add(_X, _Y) |> _Z --- add(_X, _Y) -> _Z
  -add premise 1 _X |> _
  -add premise 1 failed: seek another rule
  -addExpr _E1 -> _V1  _E2 -> _V2 --- add(_E1, _E2) -> add(_V1, _V2)
  -addExpr premise 1 _E1 -> _V1
    Rewrite call 2 add(3, 4)  ->
    -add _X |> _  _Y |> _  __add(_X, _Y) |> _Z --- add(_X, _Y) -> _Z
    -add premise 1 _X |> _
    -add premise 2 _Y |> _
    -add premise 3 __add(_X, _Y) |> _Z
    -add rewrites to 7
  -addExpr premise 2 _E2 -> _V2
    Rewrite call 3 5  ->
      Terminal 5
  -addExpr rewrites to add(7, 5)
Step 2
  Rewrite call 4 add(7, 5)  ->
  -add _X |> _  _Y |> _  __add(_X, _Y) |> _Z --- add(_X, _Y) -> _Z
  -add premise 1 _X |> _
  -add premise 2 _Y |> _
```

```
  -add premise 3 __add(_X, _Y) |> _Z
  -add rewrites to 12
Step 3
  Rewrite call 5 12  ->
    Terminal 12
Normal termination on 12 after 3 steps and 5 rewrites
```

B.3.10 Forcing deterministic execution

We have discussed elsewhere about the difference between an SOS specification as a declarative, mathematical object and the particular concretisation or implementation that we are using here. Now from a declarative perspective, the specification in `l7.art` models several different executions since any order in which we evaluate the conditions is valid. That means that we might check the left condition first, or the right condition, and in fact where the checks trigger further behaviour, any interleaving of those actions is allowed.

So, in fact our `addexpr` rule admits many possible rewrite sequences, and if eSOS were faithfully implementing the specification so as to fulfill everything that the declarative meaning allows then we would need to explore all interleavings of rewrites.

This is undesirable for practical reasons since that suggests that very large amounts of computation might be required.

Much more importantly though, if we were to write a similar specification for subtraction then we would end up with a non-confluent rewrite system. We know that in general $(a - b) - c$ is not the same as $a - (b - c)$, so we can see that if we were to explore the left-first and then the right-first SOS traces for subtraction in the style of `l8.esos` we would in general get different final states.

Now, all is not lost because we can write the rules for arithmetic operators in such a way as to *force* left-associative or right-associative evaluation. This has the twin benefits of ensuring the the specification is confluent, and that only one rewrite trace has to be explored (that is, the rewrites are *deterministic*).

Create a specification called `L8.RT` containing these rules:

```
-add
_X |> __int32(_)  _Y |> __int32(_)  __add(_X, _Y) |> _Z
---
add(_X,_Y) -> _Z

-addRight
 _n |> __int32(_) _E2 -> _I2
---
add(_n, _E2) -> add(_n, _I2)

-addLeft
_E1 -> _I1
---
add(_E1, _E2) -> add(_I1, _E2)
```

```
!trace 4
!try add(add(3,4), 5)
```

In this specification, rule `-addright` can cause a step if the left hand operand of the `add(,)` term is already an integer, as opposed to a subexpression. That means that it cannot run until some other rules has reduced the left operand to a value, and that is the purpose of the `addleft rule`. We call these rules – the ones that force a particular ordering – *congruence* rules.

The interpreter trace shows the effect of this change:

```
*** try add(add(3, 4), 5) with relation  ->
Step 1
  Rewrite call 1 add(add(3, 4), 5)  ->
  -add _X |> _  _Y |> _  __add(_X, _Y) |> _Z --- add(_X, _Y) -> _Z
  -add premise 1 _X |> _
  -add premise 1 failed: seek another rule
  -addRight _n |> _  _E2 -> _I2 --- add(_n, _E2) -> add(_n, _I2)
  -addRight premise 1 _n |> _
  -addRight premise 1 failed: seek another rule
  -addLeft _E1 -> _I1 --- add(_E1, _E2) -> add(_I1, _E2)
  -addLeft premise 1 _E1 -> _I1
    Rewrite call 2 add(3, 4)  ->
    -add _X |> _  _Y |> _  __add(_X, _Y) |> _Z --- add(_X, _Y) -> _Z
    -add premise 1 _X |> _
    -add premise 2 _Y |> _
    -add premise 3 __add(_X, _Y) |> _Z
    -add rewrites to 7
  -addLeft rewrites to add(7, 5)
Step 2
  Rewrite call 3 add(7, 5)  ->
  -add _X |> _  _Y |> _  __add(_X, _Y) |> _Z --- add(_X, _Y) -> _Z
  -add premise 1 _X |> _
  -add premise 2 _Y |> _
  -add premise 3 __add(_X, _Y) |> _Z
  -add rewrites to 12
Step 3
  Rewrite call 4 12  ->
    Terminal 12
Normal termination on 12 after 3 steps and 4 rewrites
```

In the following examples, you are expected to analyse the rules so as to understand their function, and then run the interpreter on the example to see how the rules are used.

B.3.11 Assignment

Example `l9.art` shows *assignment*, the creation of a binding within a store `sig`.

```
-assign
_n |> __int32(_)  __put(_sig, _X, _n) |> _sig1
---
assign(_X, _n), _sig -> __done, _sig1

!try assign(tmp, 32), __map()
```

B.3.12 Sequencing

Sequencing, illustrated in example `l10.art` requires two rules: a base rule that reduces the pair `__done, command` to just `command`, and a resolution rule that reduces the pair `command1, command2` to `__done, commandf2`

```
-sequenceDone
---
seq(__done, _C) -> _C

-sequence
_C1 -> _C1P
---
seq(_C1, _C2) -> seq(_C1P, _C2)

!try seq(__done, 7)
```

B.3.13 Assigning the result of an expression

Example `l11.art` shows the resolution of an expression which is then assigned to a program variable.

```
-add
_X |> __int32(_)  _Y |> __int32(_)  __add(_X, _Y) |> _Z
---
add(_X,_Y),_sig -> _Z,_sig

-addRight
 _n |> __int32(_) _E2,_sig -> _I2,_sig
---
add(_n, _E2),_sig -> add(_n, _I2),_sig

-addLeft
_E1,_sig -> _I1,_sig
---
add(_E1, _E2),_sig -> add(_I1, _E2),_sig
```

```
-assign
_n |> __int32(_)  __put(_sig, _X, _n) |> _sig1
---
assign(_X, _n), _sig -> __done, _sig1

-assignResolve
_E, _sig -> _I, _sigP
---
assign(_X,_E), _sig -> assign(_X, _I), _sigP

!trace 2
!try assign(a,add(3,4)), __map
```

B.3.14 Sequenced assignments

Example `l12.art` shows the effect on the store of sequences of assignments.

```
-sequenceDone
---
seq(__done, _C), _sig -> _C, _sig

-sequence
_C1, _sig1 -> _C1P, _sig2
---
seq(_C1, _C2), _sig1 -> seq(_C1P, _C2), _sig2


-assign
_n |> __int32(_)  __put(_sig, _X, _n) |> _sig1
---
assign(_X, _n), _sig -> __done, _sig1

!try assign(tmp, 32), __map()
!try seq(assign(tmp1, 32),assign(tmp2, 64)), __map()
!try seq(seq(assign(tmp1, 32),assign(tmp2, 64)),assign(tmp3,128)), __map()
```

B.3.15 Dereferencing and assignment

In example `l13.art` we assign 3 to variable `a` and then assign to `b` the result of dereferencing `a` from the store.

```
-sequenceDone
---
seq(__done, _C), _sig -> _C, _sig

-sequence
```

```
_C1, _sig1 -> _C1P, _sig2
---
seq(_C1, _C2), _sig1 -> seq(_C1P, _C2), _sig2

-assign
_n |> __int32(_)  __put(_sig, _X, _n) |> _sig1
---
assign(_X, _n), _sig -> __done, _sig1

-assignResolve
_E, _sig -> _I, _sigP
---
assign(_X,_E), _sig -> assign(_X, _I), _sigP

-deref
__get(_sig, _R) |> _Z
---
deref(_R),_sig -> _Z, _sig

!try seq(assign(a,3),assign(b,deref(a))), __map
```

B.3.16 Output

We *output* data by modelling the a 'printer' as a `__list` semantic entity called ϕ, Example `l14.art` illustrates a sequence of two output operations.

```
-sequenceDone
---
seq(__done, _C), _phi -> _C, _phi

-sequence
_C1, _phi1 -> _C1P, _phi2
---
seq(_C1, _C2), _phi1 -> seq(_C1P, _C2), _phi2

-output _x,_phi1 ->_y,_phi2
---
output(_x),_phi1 -> __done, __put(_phi2,_y)

!try seq(output(4),output(5)),__list
```

B.3.17 Selection with `if`

In example `l15.art`, rules for a not-equals operator with constructor `ne` are provided in the usual three-rule style, and used with rules for `if` to conditionally evaluate.

```
-ifTrue
---
if(True, _C1, _C2) -> _C1

-ifFalse
---
if(False, _C1, _C2) -> _C2

-ifResolve
_E ->_EP
---
if(_E,_C1,_C2) -> if(_EP, _C1, _C2)

-ne
_n1 |> __int32(_) _n2 |> __int32(_)
---
ne(_n1, _n2) ->  __ne(_n1, _n2)

-neRight
_n |> __int32(_)  _E2 -> _I2
---
ne(_n, _E2) -> ne(_n, _I2)

-neLeft
_E1 -> _I1
---
ne(_E1, _E2) -> ne(_I1, _E2)

!try if(True, 7, 9)
!try if(False, 7, 9)

!try if(ne(3,3),7,9)
!try if(ne(3,4),7,9)
```

B.3.18 Iteration with `while`

Example `l16.art` illustrates the handling of `while` loops by expanding to an `if` statement. The second `!try` is commented out since it is an *infinite* loop. Uncomment it and see what happens.

```
-sequenceDone
---
seq(__done, _C) -> _C

-sequence
_C1 -> _C1P
---
```

```
seq(_C1, _C2) -> seq(_C1P, _C2)

-ifTrue
---
if(True, _C1, _C2) -> _C1

-ifFalse
---
if(False, _C1, _C2) -> _C2

-ifResolve
_E ->_EP
---
if(_E,_C1,_C2) -> if(_EP, _C1, _C2)

-while
---
while(_E, _C) -> if(_E, seq(_C, while(_E,_C)), __done)

!try while(False, __done)

//!try while(True, __done)
```

B.3.19 The GCD language

We arrive, at last, at the full set of rules for our GCD language which are in the file `SLELabs/SOS/gcdSmallStep.art`. This language supports only subtraction, the greater-than and not-equal operations, along with variables, `if` statements and `while` loops. However, it is sufficient to implement Euclid's GCD algorithm, and the extension to other arithmetic operators and relations simply follows the pattern of the rules here for subtraction and $<$.

After the rules, you will see a large number of `!try` directives which exercise all of the language features. The final `!try` computes the GCD of 6 and 9. Work your way through the individual language features, ensuring that you understand how they work.

```
-sequenceDone
---
seq(__done, _C), _sig -> _C, _sig

-sequence
_C1, _sig -> _C1P, _sigP
---
seq(_C1, _C2), _sig -> seq(_C1P, _C2), _sigP

-ifTrue
---
```

```
if(True, _C1, _C2),_sig -> _C1, _sig

-ifFalse
---
if(False, _C1, _C2),_sig -> _C2,_sig

-ifResolve
_E, _sig ->_EP, _sigP
---
if(_E,_C1,_C2),_sig -> if(_EP, _C1, _C2), _sigP

-while
---
while(_E, _C),_sig -> if(_E, seq(_C, while(_E,_C)), __done), _sig

-assign
_n |> __int32(_)
---
assign(_X, _n), _sig -> __done, __put(_sig, _X, _n)

-assignResolve
_E, _sig -> _I, _sigP
---
assign(_X,_E), _sig -> assign(_X, _I), _sigP

-gt
_n1 |> __int32(_) _n2 |> __int32(_)
---
gt(_n1, _n2),_sig ->  __gt(_n1, _n2),_sig

-gtRight
_n |> __int32(_) _E2, _sig -> _I2,_sigP
---
gt(_n, _E2),_sig -> gt(_n, _I2), _sigP

-gtLeft
_E1, _sig -> _I1, _sigP
---
gt(_E1, _E2),_sig -> gt(_I1, _E2), _sigP

-ne
_n1 |> __int32(_) _n2 |> __int32(_)
---
ne(_n1, _n2),_sig ->  __ne(_n1, _n2),_sig

-neRight
_n |> __int32(_)  _E2, _sig -> _I2,_sigP
```

```
---
ne(_n, _E2),_sig -> ne(_n, _I2), _sigP

-neLeft
_E1, _sig -> _I1, _sigP
---
ne(_E1, _E2),_sig -> ne(_I1, _E2), _sigP

-sub
_n1 |> __int32(_) _n2 |> __int32(_)
---
sub(_n1, _n2),_sig ->  __sub(_n1, _n2),_sig

-subRight
_n |> __int32(_) _E2,_sig -> _I2,_sigP
---
sub(_n, _E2),_sig -> sub(_n, _I2), _sigP

-subLeft
_E1,_sig -> _I1,_sigP
---
sub(_E1, _E2),_sig -> sub(_I1, _E2), _sigP

-deref
__get(_sig, _R) |> _Z
---
deref(_R),_sig -> _Z, _sig

!try seq(__done, __empty), __map
!try seq (666,667), __map


!try if(True, 7, 9),__map
!try if(False, 7, 9),__map

!try while(False, S), __map

!try gt(4,3),__map
!try gt(3,4),__map
!try ne(3,3),__map
!try ne(3,4),__map

!try if(ne(3,3),7,9),__map
!try if(ne(3,4),7,9),__map
!try if(gt(4,3),7,9),__map
!try if(gt(3,4),7,9),__map
```

```
!try sub(3,4),__map

!try  seq(assign(x,3), assign(y,28)),__map
!try  seq(assign(x,3), assign(y,deref(x))),__map

!try seq(seq(assign(x, 1), assign(y,2)), deref(x)),__map

!try seq(seq(assign(x, 1), assign(y,2)), sub(deref(x), deref(y))),__map

!try seq(assign(z, 1), while(gt(deref(z), 0), assign(z,0))),__map
!try seq(assign(z,3), while(gt(deref(z), sub(0, 2)),
             assign(z, sub(deref(z),1)))),__map

!try seq(seq(seq(
      assign(a, 6), assign(b, 9)),
      while(ne(deref(a), deref(b)),
          if(gt(deref(a), deref(b)), assign(a, sub(deref(a), deref(b))),
                          assign(b, sub(deref(b), deref(a))))),
       assign(gcd, deref(a))),__map
```

B.4 Syntax – an introduction to parsing

This laboratory introduces parsing, parser generation and simple attribute grammars. In later labs you will use general parsing techniques called CNP parsing and MGLL parsing, but in this lab we shall use Ordered Singleton Backtrack recursive Descent (OSBRD) parsers which are so simple they can be written by hand.

B.5 Attributes – using ART with attribute grammars and GIFT rewrites

ART is a *general* parser generator which takes a specification written in the ART input language and outputs a general parser written in Java which you can compile and use from your own code. You can specify the semantics of your language using attributes and actions, and we shall also see how to do some simple rewriting of the derivation trees so as to output trees that are usable as input terms to the eSOS interpreter.

B.5.1 Getting started

Start in directory `SLELabs/Attributes/miniActionJava` by performing `cd` *`xxx`*`/SLELabs/Attributes/miniActionJava` where *`xxx`* is the directory into which you unpacked `slelabs.zip`.

If you are running on Unix or MacOS, you may need to run the command `chmod +x parse.sh` to make it executable.

You may need to set the `arthome` environment variable to the location of `art.jar`. On Windows systems, type `set arthome ..\...` On Unix systems, the shell script is already set up.

```
parse miniCall
```

or, on Un*x-style operating systems

```
./parse.sh miniCall
```

You should see the following output

```
x is 3
x is 2
x is 1
Hello from a procedure
```

B.5.2 Understanding the `parse` script – Windows version

Examine the contents of file `parse.bat` It includes

```
call clean
java –jar ../art.jar %1.art
javac –classpath .;../art.jar ARTGLLParser.java ARTGLLLexer.java
call run %1 %2 %3 %4 %5 %6 %7 %8 %9
```

On the line we call the script `clean.bat` to delete any old generated parsers, their class files and any tree visualisations that you have in the directory.

The next line runs the ART parser generator on a file of type `.art` using the name you supply as a parameter to `parse.bat`. If all goes well, ART will write out the file `ARTGLLParser.java`.

We then compile the supplied test harness `ARTTest.java` and the generated file `ARTGLLParser.java`, making sure that the jar file `art.jar` is in the class path so that the Java compiler can find the parse-time classes.

Finally, we run the compiled classes on an input which must be in a file with the same name as the grammar and file type `.str` by calling the script `run.bat`.

B.5.3 Understanding the `parse` script – Unix version

Examine the contents of file `parse.sh` It includes

```
1 #!/bin/bash
2  ./clean.sh
3 java -jar ../art.jar $1.art
4 javac -classpath ".:../art.jar" ARTGLLParser.java ARTGLLLexer.java
5 ./run.sh $1 $2 $3 $4 $5 $6 $7 $8 $9
```

On the line we call the script `clean.sh` to delete any old generated parsers, their class files and any tree visualisations that you have in the directory.

The next line runs the ART parser generator on a file of type `.art` using the name you supply as a parameter to `parse.sh`. If all goes well, ART will write out the file `ARTGLLParser.java`.

We then compile the supplied test harness `ARTTest.java` and the generated file `ARTGLLParser.java`, making sure that the jar file `art.jar` is in the class path so that the Java compiler can find the parse-time classes.

Finally, we run the compiled classes on an input which must be in a file with the same name as the grammar and file type `.str` by calling the script `run.sh`.

B.5.4 Visualising derivation trees

After an ART parser has found all of derivations of an input string it will call the *attribute evaluator* to execute the semantics and at the same time construct a (potentially rewritten) derivation tree.

If we add a `+showAll` command line option to the `parse` or to the `run` scripts as a second parameter then the evaluated tree will be printed on the console in an indented style, and in addition a file called `rdt.dot` will be output which can be used to generate a graphical version of the tree.

Try this out by (for Windows) typing

```
1 parse miniCall +showAll
```

or, on Un*x-style operating systems

```
./parse.sh miniCall +showAll
```

You will get the following console output.

```
x is 3
x is 2
x is 1
Hello from a procedure
1: statement
   2: {
   3: statements
      4: statement
         5: procedure
         6: ID
            7: &ID sub
         8: statement
      9: statements
         10: statement
            11: ID
               12: &ID x
            13: =
            14: e0
               15: e1
                  16: e2
                     17: e3
                        18: e4
                           19: e5
                              20: INTEGER
                                 21: &INTEGER 3
            22: ;
         23: statements
            24: statement
               25: while
               26: e0
               27: do
               28: statement
            230: statements
               231: statement
                  232: call
                  233: ID
                     234: &ID sub
                  235: ;
   247: }
```

The numbered lines of output each correspond to a single node in the tree shown above: the node number appear before the colon and the label after it,

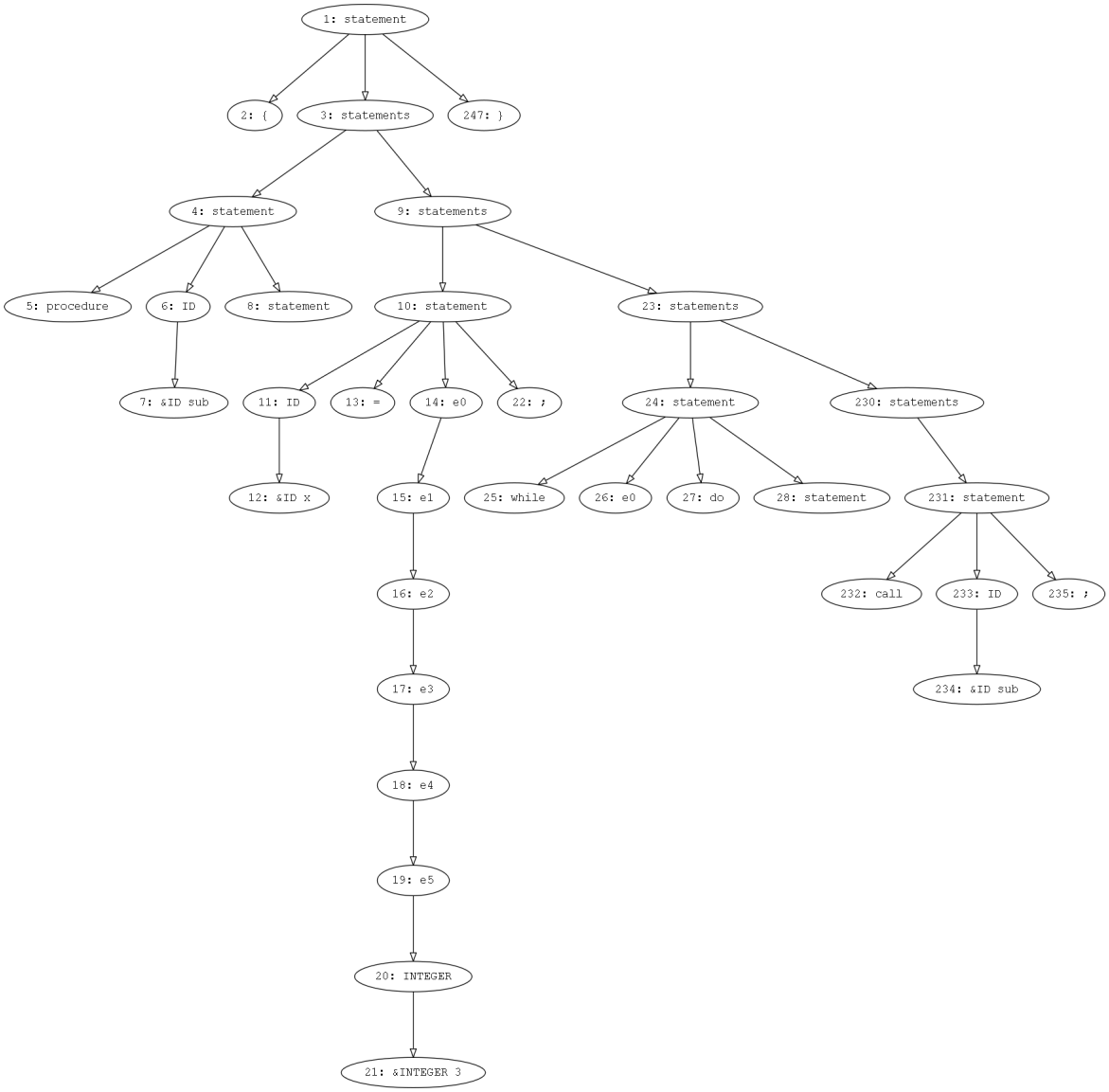
and the depth of the node in the tree is represented by the indentation level.

If you have the `GraphViz` graph drawing utilities installed on your system, you can type

```
dot -Tpdf rdt.dot > rdt.dot.pdf
```

to generate a graphical representation of the tree that can then be displayed using your PDF reader. Other output formats are available including `.png` files.

Warning! These trees can get very big very quickly, and the dot tool may give up if the tree is very large. This tree is produced after running `parse miniCall`:



When you use `+showAll` the tree is also printed as a term, which will be useful when we start building trees that are intended as inputs to the eSOS interpreter. These terms can quickly swamp the output, so they are written to the file `term.txt` and you need to look in that file for the rendered output.

The term for the `miniCall` example is:

```
statements(statement('{', statements(statement('procedure', ID < rightExtent=16
leftExtent=12 lexeme=sub v=sub >(sub), statement),
statements(statement(ID < rightExtent=62 leftExtent=55 lexeme=x v=x
>(x), '=', e0 < v=3 >(e1 < v=3 >(e2 < v=3 >(e3 < v=3 >(e4
< v=3 >(e5 < v=3 >(INTEGER < v=3 rightExtent=66 leftExtent=64
lexeme=3 >(3)))))))), ';'), statements(statement('while', e0 < v=0
>, 'do', statement), statements(statement('call', ID < rightExtent=134 leftExtent=130
lexeme=sub v=sub >(sub), ';')))))), '}'))
```

B.5.5 Simple grammars

Create a new file `first.art` with this content:

```
S ::= 'b' | 'a' X '@'
X ::= 'x' X | #
```

This is the ART version of the first grammar that we wrote in the section on OSBRD parsing. Each rule starts with a nonterminal followed by a `::=` symbol. Terminals are delimited by single quotes and ϵ, the empty string, is denoted by `#`.

Now make a file `first.str` containing:

```
axx@
```

Generate and run the parser by typing `parse first` Do not add a file type.

If you have the textual tree output routine enabled, you should see the following output:

```
1: S
  2: a
  3: X
    4: x
    5: X
      6: x
      7: X
        8: #
  9: @
```

Try changing the input by adding more `x` characters and observe what happens.

Try removing the final `@` character and see what happens.

B.5.6 Using builtins

ART provides a family of useful builtin lexer functions which can be used to process things like strings, integers and identifiers.

Create a file `assign.art` containing

```
S ::= &ID '=' &INTEGER ';'
```

and an input file `assign.str` containing

```
x = 23;
```

Process the files using `parse assign` and see that the `&ID` and `&INTEGER` builtins have matched the alphanumeric identifier and the integer. Experiment with changing the input file to have a longer identifier or a different number. What happens if you try a negative integer?

B.5.7 Exercises

Now complete these exercises.

1. Write and test a grammar that specifies the language of well nested subtraction expressions over integers. Check the tree to ensure that you have the correct associativity.

2. Write and test a grammar that specifies the language of well formed boolean expressions using the Java Boolean operators `& | !` and the constants `true` and `false`. Ensure that Java's operator priorities and associativities are correctly implemented.

3. Write and test a grammar that specifies the language of BNF expressions using ART syntax.

B.5.8 Attribute evaluation in ART

In this lab we are going to learn how to extend ART grammars with semantic actions and attributes so that we can build complete translators.

B.5.9 Simple grammars and actions

Create a file `abaction.art` with these contents:

```
S ::= A | B

A ::= 'a'

B ::='b'
```

Now create a file containing a single `a` character and run `parse abaction +showAll`.

You should see the following output

```
1: S
  2: A
    3: a
```

This shows that the parser found the derivation

$$S \Rightarrow A \Rightarrow a$$

We can arrange for the grammar to announce what it has found by adding semantic actions. In ART, an action is enclosed in braces `{ }`. Within the braces we can add any syntactically valid Java fragment.

Modify your `abaction.art` file so that it announces when it has matched the letter `a`.

```
S ::= A | B

A ::= 'a' {System.out.println("Found an a");}

B ::='b'
```

Now we get this output:

```
Found an a
1: S
  2: A
    3: a
```

Change the input to `b` and satisfy yourself that that the message no longer appears.

B.5.10 The execution order of actions

ART first finds all the derivations of the input in the grammar, then selects one of the (potentially infinite set of) derivations. Only then does the evaluator run: it visits the derivation tree top-down, left-to-right and executes actions as it passes between nodes, in the order that they are written in the grammar.

We now extend the grammar to match a sequence of `a` characters.

```
S ::= A | B

A ::= 'a' {System.out.println("Found an a");} A | #

B ::='b'
```

Run this using the input `aaa` to get this output:

```
Found an a
Found an a
Found an a
1: S
  2: A
    3: a
    4: A
      5: a
      6: A
        7: a
        8: A
          9: #
```

The tree has three terminal `a` nodes, and the message is printed out three times.

The deepest node in the tree is an epsilon node labeled `#`, and it will be visited last. If we add an action to the `A ::= #` production, we can announce that we have reached the end of the list.

```
S ::= A | B

A ::= 'a' {System.out.println("Found an a");} A |
        {artText.println("End of list of a"); } #

B ::='b'
```

which yields

```
Found an a
Found an a
Found an a
End of list of a
1: S
  2: A
    3: a
    4: A
      5: a
      6: A
        7: a
        8: A
          9: #
```

Notice, by the way, that ART provides a special object called `artText` which supports some useful text manipulation methods, including a print method.

B.5.11 Attributes

Simply adding print statements to a grammar does not provide much useful capability because all they can do is report where the evaluator has got to. To make a useful translator, we need to be able to transfer information across the tree, possibly transforming it as we go.

In an attribute grammar we define a (possibly empty) set of attributes for each nonterminal. The actions execute in the context of a small sub-tree: each action can 'see' a single parent node and its children, but no more. The name of the parent node will be the name of a nonterminal, and the names of the child nodes will be the name of a nonterminal suffixed by an integer instance number.

We can add an attribute `listLength` to nonterminal `A`, and an action which propagates the length of the list up the tree. We then add an action to the start symbol to print out the length:

Change your `abaction.art` grammar to

```
S ::= A { artText.println("List length is " + A1.listLength); } | B

A<listLength:int> ::=
  'a' A { A.listLength = A1.listLength + 1; } |
  # { A.listLength = 0; }

B ::='b'
```

When run on `aaa` we get

```
List length is 3
1: S
  2: A
    3: a
    4: A
      5: a
      6: A
        7: a
        8: A
          9: #
```

We now have enough machinery to implement arbitrary transformations.

B.5.12 `miniCalc` – a simple calculator

Review the grammar `minisyntax.art` and ensure that you understand the way in which operator associativities and priorities are encoded into the grammar.

Now compare with the grammar `minicalc.art`.

```
(*************************************************************************
*
* miniCalc.art – Adrian Johnstone 9 January 2016
*
*************************************************************************)
statement ::= 'print' '(' printElements ')' ';'

printElements ::= STRING_DQ { artText.printf("%s", STRING_DQ1.v); } |
          STRING_DQ { artText.printf("%s", STRING_DQ1.v); } ',' printElements |
          e0 { artText.printf("%d", e01.v); } |
          e0 { artText.printf("%d", e01.v); } ',' printElements

e0 <v:int> ::= e1 { e0.v = e11.v; } |
                  e1 '>' e1 { e0.v = e11.v > e12.v ? 1 : 0; } |
                  e1 '<' e1 { e0.v = e11.v < e12.v ? 1 : 0; } |
                  e1 '>=' e1 { e0.v = e11.v >= e12.v ? 1 : 0; } |
                  e1 '<=' e1 { e0.v = e11.v <= e12.v ? 1 : 0; } |
                  e1 '==' e1 { e0.v = e11.v == e12.v ? 1 : 0; } |
                  e1 '!=' e1 { e0.v = e11.v != e12.v ? 1 : 0; }

e1 <v:int> ::= e2 { e1.v = e21.v; } |
                   e1 '+' e2 { e1.v = e11.v + e21.v; } |
                   e1 '-' e2 { e1.v = e11.v - e21.v; }

e2 <v:int> ::= e3 { e2.v= e31.v; } |
                   e2 '*' e3 { e2.v = e21.v * e31.v; } |
                   e2 '/' e3 { e2.v = e21.v / e31.v; } |
                   e2 '%' e3 { e2.v = e21.v % e31.v; }

e3 <v:int> ::= e4 {e3.v = e41.v; } |
                   '+' e3 {e3.v = e41.v; } |
                   '-' e3 {e3.v = -e41.v; }

e4 <v:int> ::= e5 { e4.v = e51.v; } |
                   e5 '**' e4 {e4.v = (int) Math.pow(e51.v, e41.v); }

e5 <v:int> ::= INTEGER {e5.v = INTEGER1.v; } |
                   '(' e1 { e5.v = e11.v; } ')'
```

As we visit each node of the tree, we perform the computation required by that part of the syntax. The values propogate up via the `v` attributes.

B.5.13 `miniAssign` – adding variables

The previous grammar performs computations over expressions involving literal integers. We want to be able to add variables and an assignment statement.

Now, at the time we write the grammar, we do not know the names of the variables that a user might write into a program, so we cannot simply create attributes to hold the variables. Instead, we create a map which holds bindings of values to identifiers. Assignment statements update the map with new values. Variable uses on the right hand side of an expression access the map to retrieve values.

The grammar `miniAssign.art` extends Mini with a symbol table implemented as a map from identifiers to integer values.

```
(*****************************************************************************
*
* miniAssign.art – Adrian Johnstone 9 January 2016
*
*****************************************************************************)
prelude {import java.util.HashMap;}

support { HashMap<String, Integer> symbols = new HashMap<String, Integer>(); }

statements ::= statement | statement statements

statement ::= ID '=' e0 ';' { symbols.put(ID1.v, e01.v); } |
              'print' '(' printElements ')' ';'

printElements ::= STRING_DQ { artText.printf("%s", STRING_DQ1.v); } |
       STRING_DQ { artText.printf("%s", STRING_DQ1.v); } ',' printElements |
       e0 { artText.printf("%d", e01.v); } |
       e0 { artText.printf("%d", e01.v); } ',' printElements

e0 <v:int> ::= e1 { e0.v = e11.v; } |
               e1 '>' e1 { e0.v = e11.v > e12.v ? 1 : 0; } |
               e1 '<' e1 { e0.v = e11.v < e12.v ? 1 : 0; } |
               e1 '>=' e1 { e0.v = e11.v >= e12.v ? 1 : 0; } |
               e1 '<=' e1 { e0.v = e11.v <= e12.v ? 1 : 0; } |
               e1 '==' e1 { e0.v = e11.v == e12.v ? 1 : 0; } |
               e1 '!=' e1 { e0.v = e11.v != e12.v ? 1 : 0; }

e1 <v:int> ::= e2 { e1.v = e21.v; } |
               e1 '+' e2 { e1.v = e11.v + e21.v; } |
               e1 '-' e2 { e1.v = e11.v - e21.v; }

e2 <v:int> ::= e3 { e2.v= e31.v; } |
               e2 '*' e3 { e2.v = e21.v * e31.v; } |
               e2 '/' e3 { e2.v = e21.v / e31.v; } |
               e2 '%' e3 { e2.v = e21.v % e31.v; }

e3 <v:int> ::= e4 {e3.v = e41.v; } |
```

```
38                      '+' e3 {e3.v = e41.v; } |
39                      '−' e3 {e3.v = −e41.v; }
40
41 e4 <v:int> ::= e5 { e4.v = e51.v; } |
42                      e5 '**' e4 {e4.v = (int) Math.pow(e51.v, e41.v); }
43
44 e5 <v:int> ::= INTEGER {e5.v = INTEGER1.v; } |
45                      ID { e5.v = symbols.get(ID1.v); } |
46                      '(' e1 { e5.v = e11.v; } ')'
```

Now, we need to handle two new technical difficulties before proceeding. ART generated parsers are written into a Java class. If we want to access parts of the Java API, we need to import them, and if we want to create instances of Java API classes we need to declare them as members of the parser class. To allow this we have two ART declarations: `prelude {...}` which inserts arbitrary Java code at the top of the class file, and `support{...}` which inserts arbitrary Java code at the top of the class itself.

In this case, we import the `HashMap` class, and declare a member which maps `String` instances to `int`. We shall use this as a symbol table. We are not attempting to have any kind of scope regime —there is a single global map which is accumulated as we work through the input.

In fact, the approach is very fragile. If we try to access an undefined variable, then the semantic action in the rule `e5 ::= ID` will yield a `null` value. Try changing the input to generate this error.

We now have much of a real programming language, but as yet no control flow. That is the topic of next week's lab.

B.5.14 Exercises

1. Add left shift and right shift operators (`<<` and `>>`) to the `miniAssign` grammar. Look up Java's priority and associativity rules to ensure that you implement them in your Mini grammar, and add appropriate attributes and actions to allow such expressions to be correctly evaluated.

2. Add a check action to the rule for `e5` which catches the use of an undefined variable.

3. Write a grammar which matches decimal literals, and add semantic actions so that your grammar will match the input 12300, load an attribute with the appropriate decimal value and then print it out.

B.5.15 Delayed attributes in ART

We can postpone evaluation of a subtree by giving it a so-called delayed attribute, and these subtrees can then be evaluated under the control of semantic actions. This allows us, for instance, to repetetively evaluate the body of a `while` loop.

B.5.16 A first example of delayed attributes

Create a new file `delay.art` containing:

```
S ::= 'if' P 'then' A
P ::= 'true' | 'false'
A ::= 'print'
```

The language of this grammar is

{ `if true then print`, `if false then print` }

. Use the `parse.bat` batch file to parse both inputs using `delay.art` and verify that, for instance, the derivation tree for the first element is:

```
1: S
  2: if
  3: P
    4: true
  5: then
  6: A
    7: print
```

Now expand the grammar with attributes and actions as follows:

```
S ::= 'if' P 'then' A
P<v:boolean> ::= 'true' {P.v = true;} | 'false' {P.v = false;}
A ::= 'print' {artText.println("Printed");}
```

When run with the input `if true then print`, we get this output

```
Printed
1: S
  2: if
  3: P
    4: true
  5: then
  6: A
    7: print
```

Unfortunately, we get almost the same output with the other input...

```
Printed
1: S
  2: if
  3: P
    4: false
```

```
  5: then
  6: A
    7: print
```

We shall now add a delayed attribute to the instance of A, and use the synthesized result of P to decide whether to evaluate A, thus building an interpreter for `if` statements.

```
S<dummy:int> ::= 'if' P 'then' A< { if (P1.v) artEvaluate(S.A1, A1);}
P<v:boolean> ::= 'true' {P.v = true;} | 'false' {P.v = false;}
A ::= 'print' {artText.println("Printed");}
```

The `<` annotation on the instance of `A` delays the evaluation. In the semantic action, we look at the synthesized value from P, and only evaluate `A` if it is true.

Look closely at the tree too. The tree is built by the 'automatic' outer instance of the evaluator function. Since it does not descend into `A`, the subtree for `A` is truncated and as a result node 7 does not appear.

Here is the output for `if true then print`

```
Printed
1: S
  2: if
  3: P
    4: true
  5: then
  6: A
```

and here is the output for `if false then print`

```
1: S
  2: if
  3: P
    4: false
  5: then
  6: A
```

B.5.17 `miniIf` – adding `if-then-else` to Mini

In Mini, the only available type is `int`. We shall use an integer value of zero to represent false and any other integer value to represent true. (This is how booleans are represented in ANSI-C, by the way. Later versions of C add a boolean type.)

We need to take some care with the syntax of the `if then else` statement — we only have BNF available so we make a rule called `elseOpt` which matches either ϵ or `else ...`. We then delay evaluation of both `statement`

and `elseOpt`, placing the evaluation under the control of the value computed by `e0`.

```
(*****************************************************************************
*
* miniIf.art – Adrian Johnstone 9 January 2016
*
*****************************************************************************)
prelude {import java.util.HashMap;}

support { HashMap<String, Integer> symbols = new HashMap<String, Integer>(); }

statement ::= ID '=' e0 ';' { symbols.put(ID1.v, e01.v); } | (* assignment *)

                'if' e0 'then' statement< elseOpt< (* if statement *)
                { if (e01.v != 0)
                      artEvaluate(statement.statement1, statement1);
                  else
                      artEvaluate(statement.elseOpt1, elseOpt1);
                } |

                'print' '(' printElements ')' ';' |

                '{' statements '}'

elseOpt ::= 'else' statement | #

statements ::= statement | statement statements

printElements ::= STRING_DQ { artText.printf("%s", STRING_DQ1.v); } |
   STRING_DQ { artText.printf("%s", STRING_DQ1.v); } ',' printElements |
   e0 { artText.printf("%d", e01.v); } | e0 { artText.printf("%d", e01.v); }
       ',' printElements

e0 <v:int> ::= e1 { e0.v = e11.v; } |
    e1 '>' e1 { e0.v = e11.v > e12.v ? 1 : 0; } | (* Greater than *)
    e1 '<' e1 { e0.v = e11.v < e12.v ? 1 : 0; } | (* Less than *)
    e1 '>=' e1 { e0.v = e11.v >= e12.v ? 1 : 0; } | (* Greater than or equals*)
    e1 '<=' e1 { e0.v = e11.v <= e12.v ? 1 : 0; } | (* Less than or equals *)
    e1 '==' e1 { e0.v = e11.v == e12.v ? 1 : 0; } | (* Equal to *)
    e1 '!=' e1 { e0.v = e11.v != e12.v ? 1 : 0; } (* Not equal to *)

e1 <v:int> ::= e2 { e1.v = e21.v; } |
    e1 '+' e2 { e1.v = e11.v + e21.v; } | (* Add *)
    e1 '-' e2 { e1.v = e11.v - e21.v; } (* Subtract *)

```

```
44 e2 <v:int> ::= e3 { e2.v= e31.v; } |
45     e2 '*' e3 { e2.v = e21.v * e31.v; } | (* Multiply *)
46     e2 '/' e3 { e2.v = e21.v / e31.v; } | (* Divide *)
47     e2 '%' e3 { e2.v = e21.v % e31.v; } (* Mod *)
48
49 e3 <v:int> ::= e4 {e3.v = e41.v; } |
50     '+' e3 {e3.v = e41.v; } | (* Posite *)
51     '-' e3 {e3.v = -e41.v; } (* Negate *)
52
53 e4 <v:int> ::= e5 { e4.v = e51.v; } |
54     e5 '**' e4 {e4.v = (int) Math.pow(e51.v, e41.v); } (* exponentiate *)
55
56 e5 <v:int> ::= INTEGER {e5.v = INTEGER1.v; } | (* Integer literal *)
57     ID { e5.v = symbols.get(ID1.v); } | (* Variable access *)
58     '(' e1 { e5.v = e11.v; } ')' (* do-first *)
59
60 ID <leftExtent:int rightExtent:int lexeme:String v:String> ::=
61   &ID {ID.lexeme = artLexeme(ID.leftExtent, ID.rightExtent);
62         ID.v = artLexemeAsID(ID.leftExtent, ID.rightExtent); }
63
64 INTEGER <leftExtent:int rightExtent:int lexeme:String v:int> ::=
65   &INTEGER {INTEGER.lexeme = artLexeme(INTEGER.leftExtent, INTEGER.rightExtent);
66       INTEGER.v = artLexemeAsInteger(INTEGER.leftExtent, INTEGER.rightExtent); }
67
68 STRING_DQ <leftExtent:int rightExtent:int lexeme:String v:String> ::=
69   &STRING_DQ {STRING_DQ.lexeme =
70                     artLexeme(STRING_DQ.leftExtent, STRING_DQ.rightExtent);
71   STRING_DQ.v = artLexemeAsString(STRING_DQ.leftExtent, STRING_DQ.rightExtent); }
```

Exercise: write programs to test the nested-if ambiguity. Do ART's default disambiguation rules generate the expected results?

B.5.18 `miniWhile` – adding loops

The specification `miniWhile.art` further extends Mini with a `while` loop. Here is the key addition:

```
1                 'while' e0< 'do' statement< (* while statement *)
2                 { artEvaluate(statement.e01, e01);
3                   while (e01.v != 0) {
4                     artEvaluate(statement.statement1, statement1);
5                     artEvaluate(statement.e01, e01);
6                   }
7                 } |
```

The syntactic structure is very similar to an `if` statement, but we need to implement the actions with care. We make an initial evaluation of `e0`, and then

loop over the body and a re-evaluation of `e0` as long as the returned value is non-zero.

This is the first time we have seen a sub-tree evaluated more than once. The tree is a purely syntactic structure, and our attribute schemes (even these higher-order delayed attributes) do not allow us to change the tree. Therefore, the only way that we can see any variation in the evaluation of a sub-tree results from side effects. In this case, the relevant side effects are the updating of values in the symbol table as a result of assignments.

When run on this input

```
{
x = 3;
while x > 0 do { print("x is ", x, "\n"); x = x -1; }
}
```

We get this output

```
x is 3
x is 2
x is 1
```

Exercise: add a `do - while` statement to Mini.

B.5.19 `miniCall` – adding procedures

Attributes are only locally visible, and that is true for delayed attributes too. Procedure call is non-local in the sense that we define procedures (functions, subroutines, methods, call them what you will) in one part of a program, and we call the code from potentially many places in the program.

To connect calls to their procedure definitions, therefore, we need to be able to propagate information across the tree, in much the same way that we need to connect assignment statements to their corresponding variable usages. As we saw in the previous lab, we can do this by creating a map between identifiers and values. We can use the same idea to connect the names of procedures to their code bodies.

In a real compiler we often use a single hierarchical name space to handle variables and procedure names. To keep things simple in minicall.art, we have two independent maps, one for the variable names and one for the procedures. This allows us have a map from identifiers to integers to support assignments and variable usages, and another map from identifiers to tree nodes to support procedure definition an call. As a further simplification, we use explicit syntax to flag procedure calls with a `call` keyword.

You have the file `miniCall.art` which extends `miniWhile.art` with these productions:

```
support {HashMap<String, ART_TT> procedures = new HashMap<String, ART_TT>();}
```

```
2
3 statement ::= 'procedure' ID statement<
4                                 { procedures.put(ID1.v, statement.statement1); } |
5
6               'call' ID ';' { artEvaluate(procedures.get(ID1.v), null); } |
```

If we run this extended grammar with the input

```
1 {
2 procedure sub { print("Hello from a procedure\n"); }
3 x = 3;
4 while x > 0 do { print("x is ", x, "\n"); x = x -1; }
5 call sub;
6 }
```

we get this output

```
1 x is 3
2 x is 2
3 x is 1
4 Hello from a procedure
```

Now, this implementation is quite limited. Apart from the syntactic clumsiness, our procedures have no parameters or return values. Your task now is to consider how to make a better implementation. Here is one idea for you to pursue.

It is easy to add syntax to support one (or indeed more) formal parameters to the procedure declaration, and to add one or more expressions to the call. To simplify things, let us just add a single parameter. Now, the fundamental problem we have is that our maps only provide a single scope region, and we would not much enjoy having to think of unique parameters names for every procedure. A proper solution requires multiple scopes, but we can 'uniquify' parameter names by internally concatenating them with the name of the procedure. We can use an illegal character such as `$` as a separator, so for instance a parameter `x` to procedure `myproc` would be put into the variable map under the name `myproc$x`. This cannot clash with a user variable name, because Mini user variables names cannot include a `$` character.

Your task now is to add such a feature to Mini, and to add semantics to the `call` statement so that the argument expression is evaluated and assigned to the relevant variable.

This scheme has deep flaws. What happens if such a procedure calls itself?

B.5.20 GIFT operators in ART

In this lab we shall use ART's GIFT operators to generate Rewritten Derivation Trees (RDT's). A well-designed RDT contains all of the essential information

from the derivation tree but is more compact, and possibly rearranged to better suit the semantics. We shall restrict ourselves to the two fold operators: fold-under ^ and fold over ^^

B.5.21 `miniSyntax` – folding derivation trees

Following the examples in the main text, exercise these three grammars:

```
X ::= `a `b `c `d
```

and

```
X ::= `a `b `c^ `d
```

and

```
X ::= `a `b `c^^ `d
```

Ensure that you visualise and fully understand the effect of the fold operators on terminals.

B.5.22 Folding nonterminals

When the in-edge to a nonterminal is folded, the children of the nonterminal are dragged upwards and become siblings of the folded nonterminal's siblings.

Exercise these three grammars and ensure you fully understand the action of the fold operators.

```
X ::= a b Y d
Y ::= y z
```

and

```
X ::= a b Y^ d
Y ::= y z
```

and

```
X ::= a b Y^^ d
Y ::= y z
```

B.5.23 Suppressing punctuation

Programming language syntaxes are defined over one-dimensional strings of characters, but derivation trees are two-dimensional. In addition, the tree nodes naturally break up the string into 'lumps'. As a result, we can separators (such as commas and semicolons) and various kinds of brackets as simply one-dimensional cues that can be discarded in the tree.

Write a simple grammar that describes procedure calls which must have exactly zero or one variable. Example strings include `Y()` and `Z(a)`.

Now add fold operators so that the resulting trees have root node labelled with the name of the function,

B.5.24 Flattening lists

Write a recursive grammar that matches comma delimited lists of identifiers such `a,b,c,d a` and `a,b` Your grammar should also match the empty string.

Now add fold operators so that the commas are removed, and the identifiers are siblings under a root node.

B.5.25 Function calls

Now combine your two previous solutions to make a grammar which describes function calls with arbitrary numbers of arguments and which produces RDT's which have a root node labelled with the name of the function and which have all of the arguments as siblings under that root node.

B.5.26 Expression trees

Write a grammar that describes expressions over the four standard arithmetic operators and parentheses which correctly captures the priorities of those operators, and the effect of parentheses.

Now add fold operators so that the resulting RDT is composed entirely of terminals, and still reflects the priorities of the operators. the applications, advantages and disadvantages of Domain Specific Languages. Start with the Wiki page and stack overflow.

C Project work

The goal

The project is an opportunity for you to display your creativity. The goal is to produce a Domain Specific Language that could be used by somebody who is not a computer scientist to access advanced features.

Choice of domain

Many DSLs are developed inside companies, and their functions are naturally very company specific, so they wouldn't make good examples for our purposes. A lot of other DSLs are essentially data description languages, and they are less interesting because they lack control flow (and are thus not Turing Complete).

Java comes with powerful libraries that include a full MIDI synthesizer, an advanced 3D graphics capability and image handling. This project will be based around those subsystems.

You must choose from one of three domains in which to work, and then you will write a small programming language suitable for use by a non-specialist. The domains are:

1. 3D modelling of objects, targeting 3D printing;
2. 2D image processing; and
3. music.

This appendix includes introductions to the Java APIs which will give you some experience of each domain so that you can make an informed decision.

Weekly activity

To give structure to your work, here is a week-by-week list of activities that you should be working to. *It is entirely your responsibility to manage your own time.* This is a framework to keep you on track, not a series of hoops to jump through!

Week	Submission	Activity
1		Experiments with JavaFX 3D
2		Experiments with simple image processing and Java MIDI
3	A	Choice of domain, language features and internal syntax
4	A	Core eSOS interpetation
5	A	DSL feature and plugin development
6	B	Design of external syntax
7	B	External to internal syntax parsing
–	–	**Part A submission at end of week 7**
8	B	Attribute evaluator for control flow
9	B	Attribute evaluator for scope and types
10	B	Example programs and testing
11	B	Finish write up
–	–	**Part B submission at end of week 11**

Assessment

The project is a *substantial* piece of work, worth 50% of the mark on this final year module with *seven* deliverables.

The marking scheme is:

1. 5% Informal language specification
2. 5% Internal syntax signatures
3. 25% eSOS rules for an interpreter
4. 10% External syntax parser generating internal syntax trees
5. 25% Attribute evaluation based interpreter
6. 10% Example domain specific programs
7. 20% A concise write up as a single PDF file which includes descriptions of the other deliverables and which emphasises your personal achievements.

C.1 Getting started

In section C.4 you will find a the deliverables for complete miniature project language called PiM which extends the GCD language with a single backend command. It is perfectly allowable for you to use the source files for PiM as the starting point for your work. Of course, **you must add significant value**,

and your project report will contain a final section called *Achievements* where you can list the extensions to PiM that you have made. It is also perfectly allowable for you to write a new language from scratch without using PiM as a base.

C.2 Submission

There are two submission deadlines, one at the end of week seven and the other at the end of term. For each, submit via Moodle a single zip file containing the required deliverables. For the first submission, you must include deliverables 1–3 (informal language specification, internal syntax signatures and eSOS rules). For the second submission, you must include all seven deliverables.

C.2.1 The writeup

Your writeup should be provided as a single PDF file. There should be an introductory section explaining your choice of domain, then a section for each of deliverables 1–6 highlighting any interesting aspects of your work. Finally, there should be an *Achievements* section which lists the novelty: that is the extensions you have made to PiM. You do not need to write extensively: the goal is to show (a) that this is your own work and not copied from elsewhere and (b) highlight to the markers anything that you are particularly proud of. You can reasonably assume that the first thing the markers will do is turn to your Achievements section to get a sense of what you have done, so take particular care to give a good summary of your language's capabilities.

C.3 Ideas

Ideas for features to implement come in two categories: general purpose programming language features and Domain Specific features. The PiM example is deliberately minimalist which leaves great scope for extension. Here is an **incomplete** list of programming language features that are *not* present in PiM and could be candidates for your own extensions. You may have other great ideas of your own.

for loops, do-while loops, repeat-until loops, break, labelled break, continue, switch statements, procedure call and return, procedure parameters, named and default parameters, scope, reference variables that allow two names to refer to the same memory location allowing linked data structures to be constructed, any kind of type system, most of the arithmetic operators, logic operators, pattern matching, lambdas, exceptions, objects, ...

C.4 PiM – the project in miniature

In this section we look at a *tiny* version of the project. The scope of the language is extremely limited, but we shall illustrate the stages you need to go through, and give examples of all of the techniques you need to exercise.

> Source code for all of the material discussed in this Appendix is available in the directory `SLELabs/Project`.
>
> You may use SLELabs/ProjectWork as the base for your work: ProjectWork is a clone of ProjectMaster.
>
> I recommend that you do not change the files in ProjectMaster so that you have a clean copy of the files for reference.
>
> The file 00README.txt in SLELabs/ProjectMaster contains instructions for running all of the examples.

Our exemplar PiM language is the GCD language from Chapter 1 extended with a statement `backend(__int32, __int32, __int32)` which connects to backend code via class `ValueUserPlugin`.

The marking scheme lists the six deliverables that you should include in your write up; the write up itself constitutes the seventh deliverable, of course.

1. An informal language specification in the style shown below, listing your languages features.
2. A list of internal syntax signatures, each with an informal comment as to its function.
3. A set of eSOS rules that can interpret terms over your internal syntax and an associated `ValueUserPlugin` for the backend.
4. A context free parser decorated with promotion operators ^ and ^^ that translates from external syntax to internal syntax.
5. A context free grammar decorated with attributes and actions that directly interprets your language an associated `ValueUserPlugin` for the backend.
6. Example programs and test outputs.

We shall now go through these six sections for the PiM language. Advice on the seventh deliverable (the write up) may be found in section C.2.1.

C.4.1 Informal language specification

The language PiM can perform simple arithmetic and call a back end function.

S.1 Programs
A PiM program is one or more statements
Statements are separated by the ; and-also operator. There is no statement terminator.
A sequence of whitespace characters can be used wherever one whitespace is valid.

S.2 Arithmetic and expressions
PiM has only 32-bit integer arithmetic
Non-keyword alphanumeric identifiers denote variables that may have integers bound to them.
The only operations provided are subtraction over constant integers and variables such as $x - y$, $x - 3$ or $3 - 4$

S.3 Predicates
PiM allows comparison of 32-bit integers using relational operators that return a boolean result
$x > y$ x greater than y
x`!=`y x not equal to y

S.4 Selection statements
PiM provides two selection statements
`if pred then statement else statement`
`if pred then statement`
where pred is a predicate as defined in section S.3, and `statement` is any statement

S.5 Iteration statements
PiM provides one iteration statement
`while pred do statement`
where pred is a predicate as defined in section S.4, and `statement` is any statement

S.6 Backend statement
PiM provides one DSL-type statement
`backend(v1, v2, v3)`
where v1, v2 and v3 are integer expressions.
This statement then activates the corresponding method in class ValueUserPlugin via the `__user()` function in the ART value library.
The behaviour depends on the code implemented in that class.
Implementation note: The only *connection between the eSOS interpreter and the Java backend is the ART value function* `__user()` *which can have any arity and which returns a single* `Value`*.*
A protocol *must be defined for passing information between the eSOS interpreter and the Java backend.*
Typically the first argument will be an operation code, and subsequent arguments will be operation-specific data. In the example plugin `ValueUserPlugin_TEXT.java`*, the only action is to print to the console the values passed, and to return a string.*

C.4.2 Internal syntax constructors and arities

- ◇ `seq(_C1, _C2)` execute command C_1 followed by command C_2
- ◇ `sub(_E1:__int32, _E2:__int32)` integer subtraction: E_1 − E_2
- ◇ `gt(_E1:__int32, _E2:__int32)` integer greater-than E_1 > E_2
- ◇ `ne(_E1:__int32, _E2:__int32)` integer not-equals E_1 ≠ E_2
- ◇ `assign(_N:__int32, _E:__int32)` bind _E to name _N in variables map
- ◇ `deref(_N))` retrieve binding for name _N in variables map
- ◇ `if(_P:__bool, _C1, _C2)` select if _P then execute C_2 else execute C_3
- ◇ `while(_P:__bool, _C1, _C2)` iterate while _P then execute C_2 else execute C_3
- ◇ `backend(_V1, _V2, _V3)` call __user(_V1, _V2, _V3)

C.4.3 eSOS rules

Source form

```
-sequenceDone
---
seq(__done, _C), _sig -> _C, _sig

-sequence
_C1, _sig -> _C1P, _sigP
---
seq(_C1, _C2), _sig -> seq(_C1P, _C2), _sigP
```

```
-ifTrue
---
if(True, _C1, _C2),_sig -> _C1, _sig

-ifFalse
---
if(False, _C1, _C2),_sig -> _C2,_sig

-ifResolve
_E, _sig ->_EP, _sigP
---
if(_E,_C1,_C2),_sig -> if(_EP, _C1, _C2), _sigP

-while
---
while(_E, _C),_sig -> if(_E, seq(_C, while(_E,_C)), __done), _sig

-assign
_n |> __int32(_)
---
assign(_X, _n), _sig -> __done, __put(_sig, _X, _n)

-assignResolve
_E, _sig -> _I, _sigP
---
assign(_X,_E), _sig -> assign(_X, _I), _sigP

-gt
_n1 |> __int32(_) _n2 |> __int32(_)
---
gt(_n1, _n2),_sig ->  __gt(_n1, _n2),_sig

-gtRight
_n |> __int32(_) _E2, _sig -> _I2,_sigP
---
gt(_n, _E2),_sig -> gt(_n, _I2), _sigP

-gtLeft
_E1, _sig -> _I1, _sigP
---
gt(_E1, _E2),_sig -> gt(_I1, _E2), _sigP

-ne
_n1 |> __int32(_) _n2 |> __int32(_)
---
ne(_n1, _n2),_sig ->  __ne(_n1, _n2),_sig
```

```
-neRight
_n |> __int32(_)  _E2, _sig -> _I2,_sigP
---
ne(_n, _E2),_sig -> ne(_n, _I2), _sigP

-neLeft
_E1, _sig -> _I1, _sigP
---
ne(_E1, _E2),_sig -> ne(_I1, _E2), _sigP

-sub
_n1 |> __int32(_) _n2 |> __int32(_)
---
sub(_n1, _n2),_sig ->  __sub(_n1, _n2),_sig

-subRight
_n |> __int32(_) _E2,_sig -> _I2,_sigP
---
sub(_n, _E2),_sig -> sub(_n, _I2), _sigP

-subLeft
_E1,_sig -> _I1,_sigP
---
sub(_E1, _E2),_sig -> sub(_I1, _E2), _sigP

-variable
__get(_sig, _R) |> _Z
---
deref(_R),_sig -> _Z, _sig

-backend
---
backend(_P1, _P2, _P3),_sig -> __user(_P1, _P2, _P3)

//!try 2 seq(assign(a, 15), seq(seq(seq(assign(b, 9), while(ne(deref(a), deref(b)), if

!try 2 @"term.txt", __map ->
```

Typeset form

[sub]
$$\frac{n_1 \triangleright \_\_\mathit{int32}(\_) \quad n_2 \triangleright \_\_\mathit{int32}(\_)}{\langle \mathsf{sub}(n_1, n_2), \sigma\rangle \to \langle \_\_\mathit{sub}(n_1, n_2), \sigma\rangle}$$

[subRight]
$$\frac{n \triangleright \_\_\mathit{int32}(\_) \quad \langle E_2, \sigma\rangle \to \langle I_2, \sigma'\rangle}{\langle \mathsf{sub}(n, E_2), \sigma\rangle \to \langle \mathsf{sub}(n, I_2), \sigma'\rangle}$$

[subLeft] $$\frac{\langle E_1, \sigma\rangle \to \langle I_1, \sigma'\rangle}{\langle \mathsf{sub}(E_1, E_2), \sigma\rangle \to \langle \mathsf{sub}(I_1, E_2), \sigma'\rangle}$$

[gt] $$\frac{n_1 \triangleright \_\_\mathit{int32}(\_) \quad n_2 \triangleright \_\_\mathit{int32}(\_)}{\langle \mathsf{gt}(n_1, n_2), \sigma\rangle \to \langle \_\_\mathit{gt}(n_1, n_2), \sigma\rangle}$$

[gtRight] $$\frac{n \triangleright \_\_\mathit{int32}(\_) \quad \langle E_2, \sigma\rangle \to \langle I_2, \sigma'\rangle}{\langle \mathsf{gt}(n, E_2), \sigma\rangle \to \langle \mathsf{gt}(n, I_2), \sigma'\rangle}$$

[gtLeft] $$\frac{\langle E_1, \sigma\rangle \to \langle I_1, \sigma'\rangle}{\langle \mathsf{gt}(E_1, E_2), \sigma\rangle \to \langle \mathsf{gt}(I_1, E_2), \sigma'\rangle}$$

[variable] $$\frac{\_\_\mathit{get}(\sigma, R) \triangleright Z}{\langle \mathsf{deref}(R), \sigma\rangle \to \langle Z, \sigma\rangle}$$

[backend] $$\frac{}{\langle \mathsf{backend}(P_1, P_2, P_3), \sigma\rangle \to \langle \_\_\mathit{user}(P_1, P_2, P_3)\rangle}$$

[ifTrue] $$\frac{}{\langle \mathsf{if}(\_\_\mathit{boolean}(\mathsf{True}), C_1, C_2), \sigma\rangle \to \langle C_1, \sigma\rangle}$$

[ifFalse] $$\frac{}{\langle \mathsf{if}(\_\_\mathit{boolean}(\mathsf{False}), C_1, C_2), \sigma\rangle \to \langle C_2, \sigma\rangle}$$

[ifResolve] $$\frac{\langle E, \sigma\rangle \to \langle E', \sigma'\rangle}{\langle \mathsf{if}(E, C_1, C_2), \sigma\rangle \to \langle \mathsf{if}(E', C_1, C_2), \sigma'\rangle}$$

[ne] $$\frac{n_1 \triangleright \_\_\mathit{int32}(\_) \quad n_2 \triangleright \_\_\mathit{int32}(\_)}{\langle \mathsf{ne}(n_1, n_2), \sigma\rangle \to \langle \_\_\mathit{ne}(n_1, n_2), \sigma\rangle}$$

[neRight] $$\frac{n \triangleright \_\_\mathit{int32}(\_) \quad \langle E_2, \sigma\rangle \to \langle I_2, \sigma'\rangle}{\langle \mathsf{ne}(n, E_2), \sigma\rangle \to \langle \mathsf{ne}(n, I_2), \sigma'\rangle}$$

[neLeft] $$\frac{\langle E_1, \sigma\rangle \to \langle I_1, \sigma'\rangle}{\langle \mathsf{ne}(E_1, E_2), \sigma\rangle \to \langle \mathsf{ne}(I_1, E_2), \sigma'\rangle}$$

[sequenceDone] $$\frac{}{\langle \mathsf{seq}(\_\_\mathit{done}, C), \sigma\rangle \to \langle C, \sigma\rangle}$$

[sequence] $$\frac{\langle C_1, \sigma\rangle \to \langle C_1', \sigma'\rangle}{\langle \mathsf{seq}(C_1, C_2), \sigma\rangle \to \langle \mathsf{seq}(C_1', C_2), \sigma'\rangle}$$

[while] $$\frac{}{\langle \mathsf{while}(E, C), \sigma\rangle \to \langle \mathsf{if}(E, \mathsf{seq}(C, \mathsf{while}(E, C)), \mathit{\_\_done}), \sigma\rangle}$$

[assign] $$\frac{n \triangleright \mathit{\_\_int32}(\_)}{\langle \mathsf{assign}(X, n), \sigma\rangle \to \langle \mathit{\_\_done}, \mathit{\_\_put}(\sigma, X, n)\rangle}$$

[assignResolve] $$\frac{\langle E, \sigma\rangle \to \langle I, \sigma'\rangle}{\langle \mathsf{assign}(X, E), \sigma\rangle \to \langle \mathsf{assign}(X, I), \sigma'\rangle}$$

C.4.4 Internal to external syntax translator

```
statement ::= seq   | assign   | if   | while   | backend

seq ::= statement statement
assign ::= ID ':='  subExpr ';'
if ::= 'if'  relExpr statement 'else'  statement
while ::= 'while'  relExpr statement
backend ::= 'backend'  '('  subExpr ','  subExpr ','  subExpr ')'

relExpr ::= subExpr   | gt   | ne
gt ::= relExpr '>'  subExpr
ne ::= relExpr '!='  subExpr

subExpr ::= operand   | sub
sub ::= subExpr '-'  operand

operand ::= deref   | INTEGER   | '('  subExpr   ')'
deref ::= ID
```

C.4.5 Attribute grammar interpreter

```
(* ART parser with attributes for the GCD language *)
prelude {import java.util.HashMap; }

support { HashMap<String, Integer> variables = new HashMap<String, Integer>();
  ValueUserPlugin valueUserPlugin = new ValueUserPlugin();
}
```

```
statements ::=
  statement { System.out.println("Variables at end of program: " + variables); }
| statement statements

statement ::=
  ID ':=' subExpr ';' { variables.put(ID1.v, subExpr1.v); }
| 'if' relExpr statement< 'else' statement<
  { if (relExpr1.v != 0)
      artEvaluate(statement.statement1, statement1);
    else
      artEvaluate(statement.statement2, statement2);
   }

| 'while' relExpr< statement<
  { artEvaluate(statement.relExpr1, relExpr1);
    while (relExpr1.v != 0) {
      artEvaluate(statement.statement1, statement1);
      artEvaluate(statement.relExpr1, relExpr1);
    }
  }

| 'backend' '(' subExpr ',' subExpr ',' subExpr ')'
  { valueUserPlugin.user(subExpr1.v,subExpr2.v,subExpr3.v); }

relExpr<v:int> ::=
  subExpr { relExpr.v = subExpr1.v; }
| relExpr '>' subExpr { relExpr.v = relExpr1.v > subExpr1.v ? 1 : 0; }
| relExpr '!=' subExpr { relExpr.v = relExpr1.v != subExpr1.v ? 1 : 0; }

subExpr<v:int> ::=
  operand { subExpr.v = operand1.v; }
| subExpr '-' operand { subExpr.v = subExpr1.v - operand1.v; }

operand<v:int> ::=
  ID {operand.v = variables.get(ID1.v); }
| INTEGER {operand.v = INTEGER1.v; }
| '(' subExpr ')' {operand.v = subExpr1.v; }

(* lexical items below this line *)
ID <leftExtent:int rightExtent:int lexeme:String v:String> ::=
  &ID {ID.lexeme = artLexeme(ID.leftExtent, ID.rightExtent);
          ID.v = artLexemeAsID(ID.leftExtent, ID.rightExtent); }

INTEGER <leftExtent:int rightExtent:int lexeme:String v:int> ::=
  &INTEGER {INTEGER.lexeme = artLexeme(INTEGER.leftExtent, INTEGER.rightExtent);
   INTEGER.v = artLexemeAsInteger(INTEGER.leftExtent, INTEGER.rightExtent); }
```

C.4.6 Examples and tests

```
a := 15; b := 9;

while a != b
  if a > b
    a := a - b;
  else
  b := b - a;

gcd := a;

backend(1,2,3)
```

```
** Accept
Variables at end of program: {a=3, b=3, gcd=3}
```

C.5 Back end libraries

In this section we look at back end code that you could connect to using your own version of `ValueUserPlugin`. There are three suggested options: JavaFX for 3D CAD and graphics, some hand rolled code for 2D image processing, and the Java MIDI system.

C.5.1 An introduction to JavaFX

There have been several major graphics libraries developed for Java. For a long time *Swing* was the 'official' graphics library, but around 2012 Oracle announced that it would be succeeded by JavaFX.

For our purposes, one of the most exciting aspects of JavaFX is that it offers 3D graphics based on meshes.

JavaFX is based in the notion of a *scene graph* (actually a tree) of objects which are rendered by underlying graphics hardware. So one might declare a box which contains different subsections, and each pane might have a picture in it or user interface buttons.

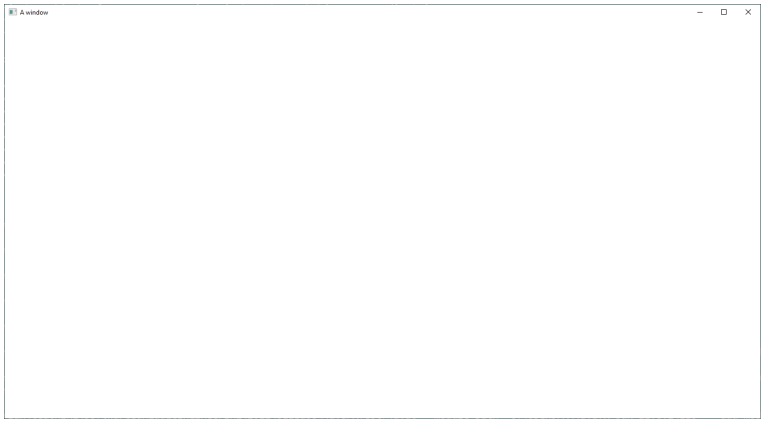
JavaFX also uses a theater metaphor to manage the display of windows. A *stage* is the platform on which a display may be built, and stages may appear as windows on the desktop.

Once we have a stage, we can attach different *scene*s to it, and each scene has an associated tree of objects to display.

WindowTest.java

Our first example simply opens a window. When you admired it, close it in the usual way by clicking on the window's close button.

```
1  import javafx.application.Application;
2  import javafx.stage.Stage;
3
4  public class WindowTest extends Application {
5    @Override
6    public void start(Stage primaryStage) throws Exception {
7      primaryStage.setTitle("A window");
8      primaryStage.show();
9    }
10 }
```



As you can see from the code above, JavaFX applications look a little different to conventional Java programs in that they appear to have no `main()` method.

The reasons for this is that JavaFX has to (a) perform a large amount of initialisation and (b) needs to be in *control*. In a point-and-click style user interface, programs are event driven. Java FX maintains various threads, one of which is called the application thread. Once everything is nicely set up, Java calls the allocation thread, which hooks into your code (class `WindowTest` in this case) as long as it is an extension of the JavaFX class `Application`.

So JavaFX user programs always extend `Applications` and instead of a `main()` method they have one called `start()`.

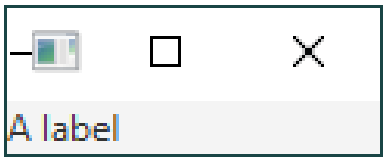
You can put initialisation code into your extension into `start()` and in a dynamic user interface that would include attaching listeners to various kinds of events. If all you want to do is just display something, then you can put all of the code into `start()` and your program will then exit, presumably opening at least one graphical window, and then wait until the user closes that window. This is the style that we shall use in these examples.

If you want a a dynamic user interface that triggers large scale processing, then you should make a new thread so that the avaFX application thread can remain responsive to user inputs such as mouse clicks. You'll find a useful tutorial referenced in subsection C.5.1 below.

SceneTest.java

We now extend the first example so that the window includes a small text label. The window will by default resize to fit the label, so it will be very small!

```
1  import javafx.application.Application;
2  import javafx.scene.Scene;
3  import javafx.scene.control.Label;
4  import javafx.stage.Stage;
5
6  public class SceneTest extends Application {
7    @Override
8    public void start(Stage primaryStage) throws Exception {
9      primaryStage.setTitle("A window");
10     Label label = new Label("A label");
11     Scene scene = new Scene(label);
12     primaryStage.setScene(scene);
13     primaryStage.show();
14   }
15 }
```



So in this example the scene graph (which was empty in the first example) is still rather degenerate: it is a only single node that is supplied as an argument to the `Scene()` constructor. The text on the label is specified on the `Label` constructor.

LabelSizeTest.java

By default, the window snaps to the size of its contents. We can override that by calling the `setWidth()` method on the `Stage` object that is associated with the window.

```
1  import javafx.application.Application;
2  import javafx.scene.Scene;
3  import javafx.scene.control.Label;
4  import javafx.stage.Stage;
5
6  public class LabelSizeTest extends Application {
7    @Override
8    public void start(Stage primaryStage) throws Exception {
9      primaryStage.setTitle("A window");
10     Label label = new Label("A label");
11     Scene scene = new Scene(label);
```

```
12
13      primaryStage.setWidth(400);
14      // primaryStage.setHeight(200);
15      primaryStage.setScene(scene);
16      primaryStage.show();
17    }
18  }
```

ImageTest.java

JavaFX knows how to render common image formats, and will deduce the format from the filename. The zip file containing these example files also includes a picture of one of my steam engines called `GERTanl.png`.

Displaying images is a two stage process. First you have to load the image into memory by creating an `Image` object. In the constructor you can specify the filename, along with sizes to scale to - the online documentation is viewable at `https://openjfx.io/javadoc/12/javafx.graphics/javafx/scene/image/package-summary.html`

After loading the image, you wrap it in an `ImageView` object. This is of the right type to be built into the JavaFX scene graph.

There is one more subtlety. We are now going to have a non-degenerate tree, so we need a parent node off which we can hang other scene nodes. The type `Group` represents a 'bland' tree node that simply acts as a scaffolding node for other tree elements. So we create a `Scene` with a `Group` as its root node, and then we can start hanging other nodes off of the group, which we do by adding them to the `Group`'s children.

```
1  import javafx.application.Application;
2  import javafx.scene.Group;
3  import javafx.scene.Scene;
4  import javafx.scene.image.Image;
5  import javafx.scene.image.ImageView;
6  import javafx.stage.Stage;
7
8  public class ImageTest extends Application {
9    @Override
10   public void start(Stage primaryStage) throws Exception {
11     primaryStage.setTitle("A window");
12     Group group = new Group();
13     Scene scene = new Scene(group);
14
15     Image image = new Image("GERTank.png", 400, 0, true, true);
16     ImageView imageView = new ImageView(image);
17
18     group.getChildren().add(imageView);
19
```

```
20        primaryStage.setWidth(1000);
21        primaryStage.setHeight(800);
22        primaryStage.setScene(scene);
23        primaryStage.show();
24    }
25 }
```

SolidTest.java

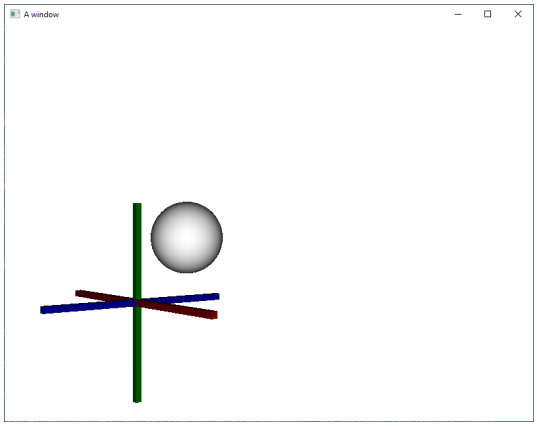
Up until now, all of our examples have used only two dimensional graphics. JavaFX has a powerful 3D imaging capability. Using 3D is a step up in complexity, but the principles remain the same: there is a scene graph (tree) and you add elements to it. You also need to add a camera to a scene or you won't see anything!

Often difficulties with 3D graphics turn out to involve having the camera looking the right way: in a later session we'll look at how to controlthe camera with the mouse.

```
1  import javafx.application.Application;
2  import javafx.application.ConditionalFeature;
3  import javafx.application.Platform;
4  import javafx.scene.Group;
5  import javafx.scene.PerspectiveCamera;
6  import javafx.scene.Scene;
7  import javafx.scene.paint.Color;
8  import javafx.scene.paint.PhongMaterial;
9  import javafx.scene.shape.Box;
10 import javafx.scene.shape.Sphere;
11 import javafx.scene.transform.Rotate;
12 import javafx.stage.Stage;
13
14 public class SolidTest extends Application {
15    public static void main(String[] args) {
16       launch(args);
```

```
17    }
18    @Override
19    public void start(Stage primaryStage) {
20      // Check to see if our graphics system will play nicely
21      if (!Platform.isSupported(ConditionalFeature.SCENE3D)) {
22        System.err.println("Your display system does not support JavaFX 3D - exiting");
23        System.exit(1);
24      }
25
26      final int windowX = 800;
27      final int windowY = 600;
28      primaryStage.setTitle("A window");
29
30      // Create a scene with a rotated group at its root
31      Group root = new Group();
32      root.setRotationAxis(Rotate.Y_AXIS);
33      root.setRotate(50);
34      Scene scene = new Scene(root, windowX, windowY, true);
35
36      // Make some coloured materials
37      final PhongMaterial redMaterial = new PhongMaterial();
38      redMaterial.setDiffuseColor(Color.DARKRED);
39      redMaterial.setSpecularColor(Color.RED);
40
41      final PhongMaterial greenMaterial = new PhongMaterial();
42      greenMaterial.setDiffuseColor(Color.DARKGREEN);
43      greenMaterial.setSpecularColor(Color.GREEN);
44
45      final PhongMaterial blueMaterial = new PhongMaterial();
46      blueMaterial.setDiffuseColor(Color.DARKBLUE);
47      blueMaterial.setSpecularColor(Color.BLUE);
48
49      // Make three coordinate boxes in different colours
50      final Box xAxis = new Box(windowX / 2, 10, 10);
51      xAxis.setMaterial(redMaterial);
52      final Box yAxis = new Box(10, windowY / 2, 10);
53      yAxis.setMaterial(greenMaterial);
54      final Box zAxis = new Box(10, 10, windowY / 2);
55      zAxis.setMaterial(blueMaterial);
56
57      // Make a sphere in the default colour (grey)
58      Sphere ball = new Sphere(50);
59      ball.setTranslateX(120);
60      ball.setTranslateY(-100);
61      ball.setTranslateZ(10);
62
63      // Attach the axes and the ball as children of the root of the scene graph
```

```
64        root.getChildren().addAll(xAxis, yAxis, zAxis, ball);
65
66        // Create camera and move it away from the origin
67        PerspectiveCamera camera = new PerspectiveCamera(false);
68        camera.setTranslateX(-0.25 * windowX);
69        camera.setTranslateY(-0.7 * windowY);
70        scene.setCamera(camera); // Putthe camera into the scene
71
72        // Now attach the scene to our stage, and 'open the curtains'
73        primaryStage.setScene(scene);
74        primaryStage.show();
75    }
76 }
```



Other JavaFX materials

There are some very good online tutorials that you can explore as part of your wider reading.

Oracle Java 8 tutorials

These are the original Oracle tutorials for Java FX. If you are working with the 3D domain you *must* read and try out the examples in the javafx-3d-graphics tutorial.

`https://docs.oracle.com/javase/8/javafx/get-started-tutorial`
`https://docs.oracle.com/javase/8/javafx/graphics-tutorial/preface.htm`
`https://docs.oracle.com/javase/8/javafx/graphics-tutorial/javafx-3d-graphics.htm`

Open JavaFX tutorials

JavaFX has now been unbundled by Oracle and is a community project. Ongoing development is under the umbrella of the openJavaFX group, and they have provided their own tutorials.

`https://openjfx.io/openjfx-docs/`

Jencov JavaFX tutorials

Jacob Jencov has provided many tutorials on Java and the Java APIs. I like his concise and fairly comprehensive approach, and I think youmight too.

`http://tutorials.jenkov.com/javafx/index.html`

Jencov JavaFX concurrency tutorial

IMPORTANT!

All modifications to the JavaFX scene graph must be performed from the main JavaFX thread. That thread also handles all UI inputs, such as mouse events. Now, if you also do a lot of back end processing on that thread then the user interface can become jittery and slow. So if you have a lot of non-UI processing to do you should multi-thread. This tutorial is a helpful introduction.

`http://tutorials.jenkov.com/javafx/concurrency.html`

C.5.2 An introduction to image processing

In this lab we shall work with the supplied Java class `ImageProcessingDemo` which you will find in the `imageProcessing` subdirectory of the `SLELabs` package.

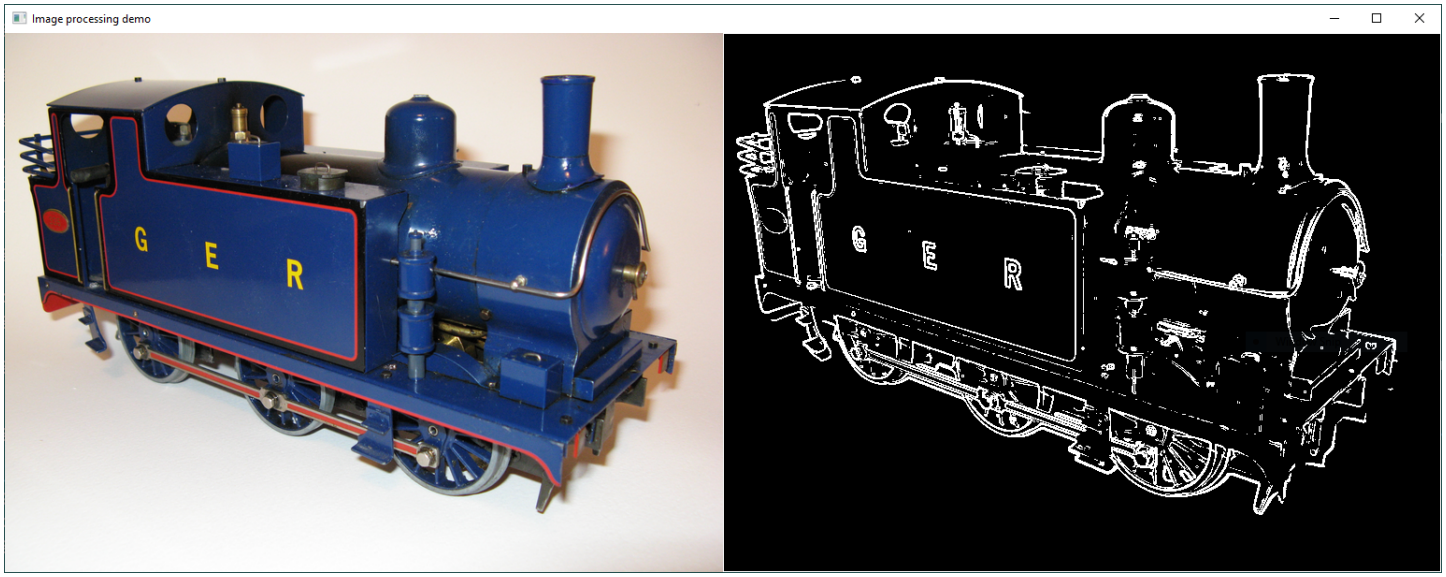
You compile and run the program by issuing the command

```
tst ImageProcessingDemo
```

The program outputs the message

```
Image Height: 576.0
Image Width: 768.0
```

and displays this window

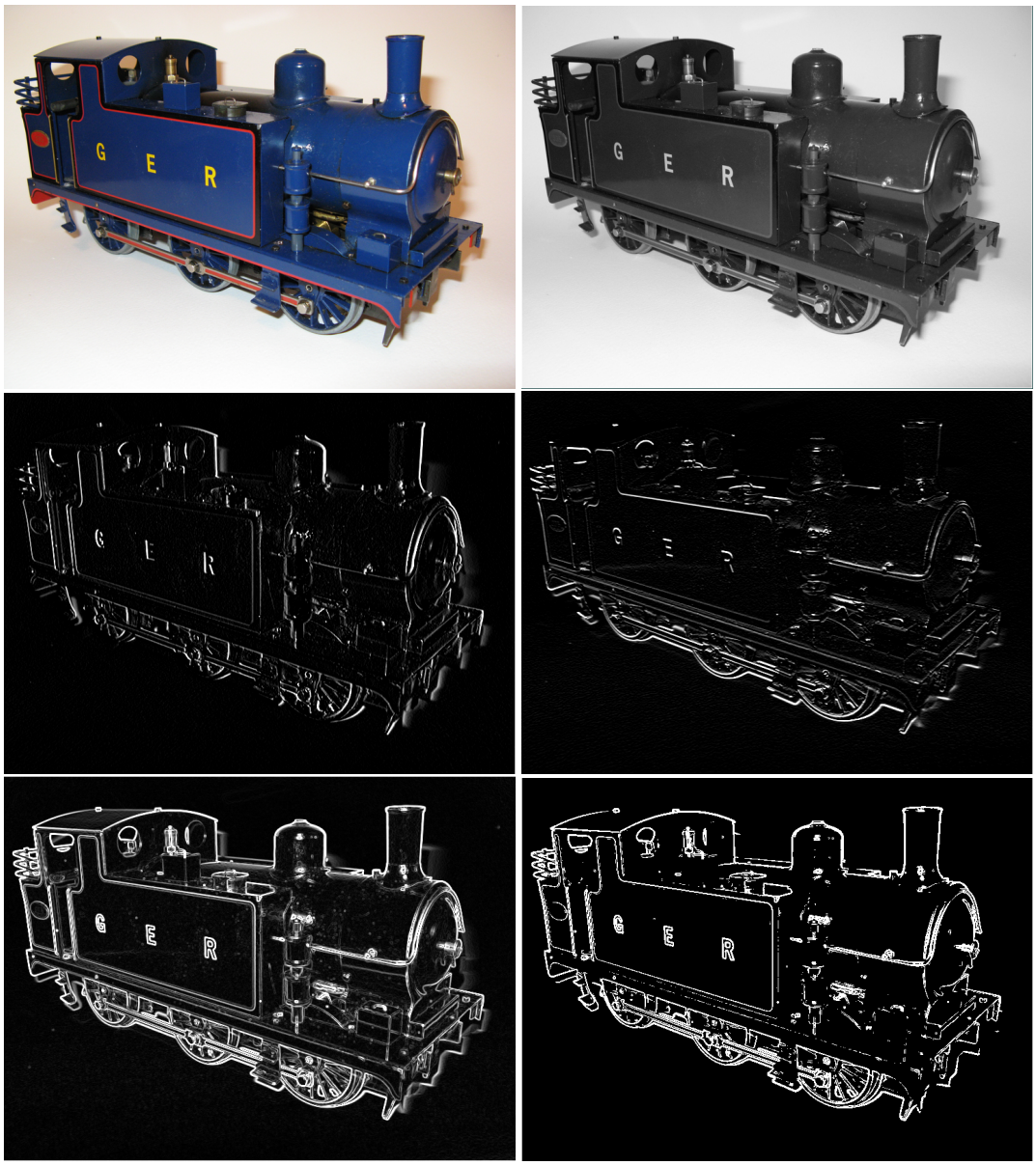


The monochrome image on the right is an edge map of the locomotive. It is bright where there are sharp edges in the original colour image, and dark where the original image has a uniform tone.

The pipeline of operations that produces this edge map is:

1. convert the colour image to monochrome
2. compute d_x, the horizontal grey-scale gradient component
3. compute d_y, the vertical grey-scale gradient component
4. compute the magnitude of the grey-scale gradient as $\sqrt{d_x^2 + d_y^2}$
5. *threshold* the gradient so that values above mid-grey are mapped to white and other values to black

The six stages look like this:




```
1  import javafx.application.Application;
2  import javafx.scene.Scene;
3  import javafx.scene.image.Image;
4  import javafx.scene.image.ImageView;
5  import javafx.scene.image.PixelReader;
6  import javafx.scene.image.PixelWriter;
7  import javafx.scene.image.WritableImage;
8  import javafx.scene.layout.HBox;
9  import javafx.scene.paint.Color;
10 import javafx.stage.Stage;
```

```
11
12 public class ImageProcessingDemo extends Application {
13   @Override
14   public void start(Stage primaryStage) throws Exception {
15     primaryStage.setTitle("Image processing demo");
16     HBox root = new HBox();
17     Scene scene = new Scene(root);
18
19     Image inputImage = new Image("GERTank.png");
20     System.out.println("Image Height: " + inputImage.getHeight());
21     System.out.println("Image Width: " + inputImage.getWidth());
22
23     WritableImage outputImage = new WritableImage((int) inputImage.getWidth(), (int) inputImage.getH
24
25     PixelReader pixelReader = inputImage.getPixelReader();
26     PixelWriter pixelWriter = outputImage.getPixelWriter();
27
28     for (int readY = 1; readY < inputImage.getHeight() - 1; readY++) {
29       for (int readX = 1; readX < inputImage.getWidth() - 1; readX++) {
30         /*
31          * p4 p3 p2
32          * p5 p0 p1
33          * p6 p7 p8
34          */
35         double p0 = pixelReader.getColor(readX, readY).grayscale().getRed();
36         double p1 = pixelReader.getColor(readX + 1, readY).grayscale().getRed();
37         double p2 = pixelReader.getColor(readX + 1, readY + 1).grayscale().getRed();
38         double p3 = pixelReader.getColor(readX, readY + 1).grayscale().getRed();
39         double p4 = pixelReader.getColor(readX - 1, readY + 1).grayscale().getRed();
40         double p5 = pixelReader.getColor(readX - 1, readY).grayscale().getRed();
41         double p6 = pixelReader.getColor(readX - 1, readY - 1).grayscale().getRed();
42         double p7 = pixelReader.getColor(readX, readY - 1).grayscale().getRed();
43         double p8 = pixelReader.getColor(readX + 1, readY - 1).grayscale().getRed();
44
45         double sobeldx = (p2 + 2 * p1 + p8) - (p4 + 2 * p5 + p6);
46         double sobeldy = (p4 + 2 * p3 + p2) - (p6 + 2 * p7 + p8);
47         double sobelFilter = Math.sqrt(sobeldx * sobeldx + sobeldy * sobeldy);
48
49         double edge = threshold(0.5, sobelFilter);
50
51         double meanFilter = (p0 + p1 + p2 + p3 + p4 + p5 + p6 + p7 + p8) / 9;
52
53         double q0 = range(0, 1, edge);
54
55         pixelWriter.setColor(readX, readY, new Color(q0, q0, q0, 1));
56       }
57     }
```

```
58
59      ImageView inputImageView = new ImageView(inputImage);
60      ImageView outputImageView = new ImageView(outputImage);
61      root.getChildren().addAll(inputImageView, outputImageView);
62      primaryStage.setScene(scene);
63      primaryStage.show();
64    }
65
66    private double range(double offset, double scale, double value) {
67      double ret = offset + (scale * value);
68      if (ret > 1.0) ret = 1.0;
69      if (ret < 0) ret = 0;
70      return ret;
71    }
72
73    private double threshold(double threshold, double value) {
74      return value >= threshold ? 1.0 : 0.0;
75    }
76  }
```

C.5.3 An introduction to the Java MIDI subsystem

```
public enum Scale {
   CHROMATIC, MAJOR, MINOR_NATURAL, MINOR_HARMONIC,
   MINOR_MELODIC_ASCENDING, MINOR_MELODIC_DESCENDING
}
```

```
public enum Scale {
public enum Chord {
      NONE, MAJOR, MINOR, MAJOR7, MINOR7
}
```

```
import javax.sound.midi.MidiChannel;
import javax.sound.midi.MidiSystem;
import javax.sound.midi.Synthesizer;

public class MiniMusicPlayer {
   private Synthesizer synthesizer;
   private MidiChannel[] channels;
   private int defaultOctave = 5;
   private int defaultVelocity = 50;
   private int bpm;
   private double bps;
   private double beatPeriod;
   private double beatRatio = 0.9;
   private int beatSoundDelay = (int) (1000.0 * beatRatio / bps);
   private int beatSilenceDelay = (int) (1000.0 * (1.0 - beatRatio) / bps);

   MiniMusicPlayer() {
      try {
         System.out.print(MidiSystem.getMidiDeviceInfo());
         synthesizer = MidiSystem.getSynthesizer();
         synthesizer.open();
         channels = synthesizer.getChannels();
      } catch (Exception e) {
         System.err.println("miniMusicPlayer exception: " + e.getMessage());
         System.exit(1);
      }

      setBeatRatio(0.9);
      setBpm(100);
      setDefaultVelocity(50);
      rest(2);
   }
```

```
33
34    public int getDefaultOctave() {
35        return defaultOctave;
36    }
37
38    public void setDefaultOctave(int defaultOctave) {
39        this.defaultOctave = defaultOctave;
40    }
41
42    public int getDefaultVelocity() {
43        return defaultVelocity;
44    }
45
46    public void setDefaultVelocity(int defaultVelocity) {
47        this.defaultVelocity = defaultVelocity;
48    }
49
50    public int getBpm() {
51        return bpm;
52    }
53
54    public void setBpm(int bpm) {
55        this.bpm = bpm;
56        bps = bpm / 60.0;
57        beatPeriod = 1000.0 / bps;
58        beatSoundDelay = (int) (beatRatio * beatPeriod);
59        beatSilenceDelay = (int) ((1.0 - beatRatio) * beatPeriod);
60    }
61
62    public void setBeatRatio(double beatRatio) {
63        this.beatRatio = beatRatio;
64        beatSoundDelay = (int) (beatRatio * beatPeriod);
65        beatSilenceDelay = (int) ((1.0 - beatRatio) * beatPeriod);
66    }
67
68    int noteNameToMidiKey(String n, int octave) {
69   // @formatter:off
70   int key = octave * 12 +
71             ( n.equals("C") ? 0
72             : n.equals("C#") ? 1
73             : n.equals("Db") ? 1
74             : n.equals("D") ? 2
75             : n.equals("D#") ? 3
76             : n.equals("Eb") ? 3
77             : n.equals("E") ? 4
78             : n.equals("F") ? 5
79             : n.equals("F#") ? 6
```

```
80            : n.equals("Gb") ? 6
81            : n.equals("G") ? 7
82            : n.equals("G#") ? 8
83            : n.equals("Ab") ? 8
84            : n.equals("A") ? 9
85            : n.equals("A#") ? 10
86            : n.equals("Bb") ? 10
87            : n.equals("B") ? 11
88            : -1);
89    // @formatter:on
90
91      if (key < 0 || key > 127) {
92        System.err.println("miniMusicPlayer exception: attempt to access out of range MIDI key "
93            + n + octave);
94        System.exit(1);
95      }
96      return key;
97    }
98
99    // Silence
100   public void rest(int beats) {
101     try {
102       Thread.sleep((long) (beats * beatPeriod));
103     } catch (InterruptedException e) {
104       /* ignore interruptedException */ }
105   }
106
107   // Single notes
108   void play(int k) {
109     try {
110       channels[0].noteOn(k, defaultVelocity);
111       Thread.sleep(beatSoundDelay);
112       channels[0].noteOn(k, 0);
113       Thread.sleep(beatSilenceDelay);
114     } catch (InterruptedException e) {
115       /* ignore interruptedException */ }
116   }
117
118   void play(String n) {
119     play(noteNameToMidiKey(n, defaultOctave));
120   }
121
122   void play(String n, int octave) {
123     play(noteNameToMidiKey(n, octave));
124   }
125
126   // Arrays of notes
```

```
127   void play(int[] k) {
128     try {
129       for (int i = 0; i < k.length; i++)
130         channels[1].noteOn(k[i], defaultVelocity);
131       Thread.sleep(beatSoundDelay);
132       for (int i = 0; i < k.length; i++)
133         channels[1].noteOn(k[i], 0);
134       Thread.sleep(beatSilenceDelay);
135     } catch (InterruptedException e) {
136       /* ignore interruptedException */ }
137   }
138
139   private void playSequentially(int[] k) {
140     try {
141       for (int i = 0; i < k.length; i++) {
142         channels[i].noteOn(k[i], defaultVelocity);
143         Thread.sleep(beatSoundDelay);
144         channels[i].noteOn(k[i], 0);
145         Thread.sleep(beatSilenceDelay);
146       }
147     } catch (InterruptedException e) {
148       /* ignore interruptedException */ }
149   }
150
151   // Scales
152   void playScale(String n, Scale s) {
153     playScale(noteNameToMidiKey(n, defaultOctave), s);
154   }
155
156   void playScale(String n, int octave, Scale s) {
157     playScale(noteNameToMidiKey(n, octave), s);
158   }
159
160   void playScale(int k, Scale s) {
161     int[] keys;
162     switch (s) {
163     case CHROMATIC:
164       keys = new int[] { k, k + 1, k + 2, k + 3, k + 4, k + 5, k + 6, k + 7, k + 8, k + 9,
165                                                   k + 10, k + 11, k + 12 };
166       break;
167
168     case MAJOR: // TTSTTTS
169       keys = new int[] { k, k + 2, k + 4, k + 5, k + 7, k + 9, k + 11, k + 12 };
170       break;
171
172     case MINOR_NATURAL: // TSTTSTT
173       keys = new int[] { k, k + 2, k + 3, k + 5, k + 7, k + 8, k + 10, k + 12 };
```

```
174         break;
175       case MINOR_HARMONIC: // TSTTS3S
176         keys = new int[] { k, k + 2, k + 3, k + 5, k + 7, k + 8, k + 11, k + 12 };
177         break;
178       case MINOR_MELODIC_ASCENDING: // TSTTS3S – harmonic with with sixth sharpened
179         keys = new int[] { k, k + 2, k + 3, k + 5, k + 7, k + 9, k + 11, k + 12 };
180         break;
181       case MINOR_MELODIC_DESCENDING: // TSTTS3S – harmonic with seventh
182                                      // flattened making it the same as the natural minor
183         keys = new int[] { k + 12, k + 10, k + 8, k + 7, k + 5, k + 3, k + 2, k };
184         break;
185
186       default:
187         keys = new int[] { k };
188         break;
189       }
190       playSequentially(keys);
191   }
192
193   // Programmed chords
194   void playChord(String n, Chord type) {
195     playChord(noteNameToMidiKey(n, defaultOctave), type);
196   }
197
198   void playChord(String n, int octave, Chord type) {
199     playChord(noteNameToMidiKey(n, octave), type);
200   }
201
202   private void playChord(int k, Chord type) {
203     int[] keys;
204     switch (type) {
205     case MAJOR:
206       keys = new int[] { k, k + 4, k + 7 };
207       break;
208     case MAJOR7:
209       keys = new int[] { k, k + 4, k + 7, k + 11 };
210       break;
211     case MINOR:
212       keys = new int[] { k, k + 3, k + 7 };
213       break;
214     case MINOR7:
215       keys = new int[] { k, k + 4, k + 7 };
216       break;
217     default:
218       keys = new int[] { k };
219       break;
220     }
```

```
221       play(keys);
222     }
223
224     public void tune() {
225       int base = 47;
226       play(base + 14);
227       play(base + 12);
228       play(base + 11);
229       play(base + 7);
230       play(base + 5);
231       play(base + 7);
232       play(base + 2);
233       rest(2);
234     }
235
236     public void tuneChordMajor() {
237       int base = noteNameToMidiKey("C", 5);
238       playChord(base + 14, Chord.MAJOR);
239       playChord(base + 12, Chord.MAJOR);
240       playChord(base + 11, Chord.MAJOR);
241       playChord(base + 7, Chord.MAJOR);
242       playChord(base + 5, Chord.MAJOR);
243       playChord(base + 7, Chord.MAJOR);
244       playChord(base + 2, Chord.MAJOR);
245     }
246
247     public void tuneChordMinor() {
248       int base = noteNameToMidiKey("C", 5);
249       playChord(base + 14, Chord.MINOR);
250       playChord(base + 12, Chord.MINOR);
251       playChord(base + 11, Chord.MINOR);
252       playChord(base + 7, Chord.MINOR);
253       playChord(base + 5, Chord.MINOR);
254       playChord(base + 7, Chord.MINOR);
255       playChord(base + 2, Chord.MINOR);
256     }
257
258     void close() {
259       rest(3);
260       synthesizer.close();
261     }
262
263     public static void main(String[] args) {
264       System.err.println("miniMusicPlayer test routine");
265       MiniMusicPlayer mp = new MiniMusicPlayer();
266
267       mp.playScale("C", Scale.CHROMATIC);
```

```
268         mp.rest(2);
269         String note = "C";
270         int octave = 6;
271         mp.play(note, octave);
272         mp.rest(2);
273         mp.playScale("C", Scale.MAJOR);
274         mp.rest(2);
275         mp.playScale("C", Scale.MINOR_NATURAL);
276         mp.rest(2);
277         mp.playScale("C", Scale.MINOR_HARMONIC);
278         mp.rest(2);
279         mp.playScale("C", Scale.MINOR_MELODIC_ASCENDING);
280         mp.playScale("C", Scale.MINOR_MELODIC_DESCENDING);
281         mp.rest(2);
282         mp.playChord("C", Chord.MAJOR);
283         mp.rest(2);
284         mp.playChord("C", Chord.MINOR);
285         mp.rest(2);
286         mp.tune();
287         mp.rest(2);
288         mp.tuneChordMajor();
289         mp.rest(2);
290         mp.tuneChordMinor();
291         mp.rest(2);
292         mp.close();
293     }
294 }
```

D A mathematics primer