Software Language Engineering

Adrian Johnstone

0 Course structure

The course material is divided into six parts. Each part has its own chapter in the course book *Software Language Engineering*; we shall study each chapter for one or two weeks in this order:

Weeks	Chapter	Topic
1	1	Formalising language
2–3	2	Rewriting systems
4–5	3	Structural Operational Semantics
6–7	4	Syntax specification and analysis
8–9	5	Attribute evaluation
10	6	Pragmatics and case studies
11		Consolidation and revision

Course sessions

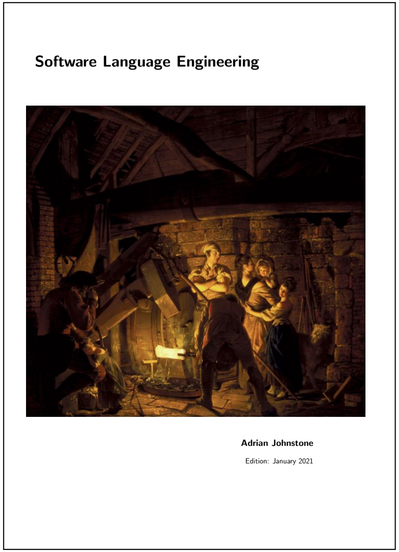
This is a 15 credit course, taught over eleven weeks, with 44 hours of programmed sessions

Monday	13.00-13.50	Lecture slot A
Tuesday	9.00–9.50	Lecture slot B
Thursday	9.00–9.50	Laboratory
Thursday	10.00–10.50	Project development

Full engagement and submissions for all assessments including the exam and the project is *required*

Assessment comprises 50% exam and 50% individual project

Learning resources



Software Language Engineering

Adrian Johnstone

Edition: January 2021

- Text book
- ART software
- Laboratory scripts and follow on questions
- Slides
- Pre-recorded videos
- Machine generated transcripts
- Exercises
- Dummy exam and past papers

- Recordings of lecture sessions

  Lab and project sessions are not recorded

Week by week – lectures and labs

Week	Book chap	Lecture topic	Lab topic
1	1	Formalising language	Languages for 3D design
2	2	Rewriting systems	Music and imaging languages
3	2	Rewriting systems	termTool
4	3	Structural Operational Semantics	eSOS 1
5	3	Structural Operational Semantics	eSOS 2
6	4	Syntax specification and analysis	OSBRD
7	4	Syntax specification and analysis	MGLL and lexicalisation
8	5	Attribute evaluation	Mini 1
9	5	Attribute evaluation	Mini 2
10	6	Case studies and the industrial context	Double project slot
11		Contingency, consolidation and revision	Double project slot

Week by week – the project

Week	Activity
1	Experiments with JavaFX 3D
2	Experiments with Java MIDI and simple image processing
3	Choice of domain and identification of domain features
4	Design of internal syntax
5	eSOS interpretation
	End of first section: submission due at end of week 7
6	Design of external syntax
7	External to internal syntax parsing
8	Attribute evaluator for control flow
9	Attribute evaluator for scope and types
10	Example programs and testing
	End of second section: submission due at end of week 11
11	Finish write up

Project marking criteria

The project is a *substantial* piece of work, worth 50% of the marks on a final year module. Within that 50%, the marking scheme is:

- Part one worth 35%
  - 5% Planned list of domain specific features with Java examples
  - 5% Internal syntax specification
  - 25% eSOS interpreter
- Part two worth 65%
  - 10% external syntax parser generating internal syntax trees
  - 25% attribute based interpreter
  - 10% example domain specific programs
  - 20% write up

Some students merely submit a version of one of my examples with some minor improvements; that is not enough to earn a pass.

The examination

The exam format changed in 2021 to be directly aligned with the style of the project deliverables.

Before 2021, there was typically a standalone question for each topic. In the new format, an informal specification of a small language is given, and you are required to construct specifications in various styles for that small language.

An example paper in this style is available on the Moodle page. Model answers will **not** be made available for that example paper, or for past papers.

Past papers from before 2021 are still worth working through since the broad intellectual content of the course has not changed.

How much work?

Lots, or lots and lots, depending on how clever you are

This is from the QAA document *Academic credit in higher education in England — an introduction*:

The amount of learning indicated by a credit value is based on an estimate using the idea of notional hours of learning. The number of notional hours of learning provides a rough guide to how long it will take a **typical** student, on average, to achieve the learning outcomes (what you will know, understand and be able to do having successfully completed the learning) specified for the module or programme.

The estimate of notional hours of learning doesn't just include formal classes, but estimates the amount of time spent in preparation for these classes, along with private or independent reading and study, plus revision and the completion of course-work required on the module.

Within the UK, one credit represents 10 notional hours of learning. Institutions use this guide as a basis for setting the credit value of a module or programme before it is offered to students. For example, a module that is estimated to involve 150 notional hours of learning will be assigned 15 credits.

Time management

Exams begin in 15 weeks time

A typical student hoping for a mid-range result will need to budget 150 hours, as per the previous slide. If you want to improve your current grade average then you should budget more hours

So that is an average of ten hours per week, including the four weeks of the vacation

(Which makes sense if we assume a 40 hour working week, and that you are doing four courses simultaneously)

We have four contact hours per week, thus a total of 44 contact hours leaving 106 for your private study. The project is designed to give structure to your private study, with a weekly programme of activities that will require you to engage with that week's lecture and laboratory topics.

Your feedback on this module and its materials is always welcome: email to `a.johnstone@rhul.ac.uk`.

Now let's begin.

Gottlob Frege 1848–1925

https://en.wikipedia.org/wiki/Gottlob_Frege

1 Formalisation

A formal system is a set of consistent rules that are susceptible to mechanisation.

Formal systems typically have *configurations* which we move between according to the rules. Board games such as Chess, Go and Draughts are all formal systems, and in fact we might think of formalisation as making a *game* which models a problem domain. Mechanisation does not necessarily mean that judgement becomes superfluous: to win we must choose the best move.

Section	Topic
1A	It's languages all the way down
1B	Utility and power in languages and their processors
1C	What do we mean by *formal*?
1D	Conway's Game of Life
1E	The reduction model: Euclid's Greatest Common Divisor

1A It's languages all the way down

In *A Brief History of Time*, Stephen Hawking wrote:

A well-known scientist (some say it was Bertrand Russell) once gave a public lecture on astronomy. He described how the earth orbits around the sun and how the sun, in turn, orbits around the centre of a vast collection of stars called our galaxy.
At the end of the lecture, a little old lady at the back of the room got up and said: "What you have told us is rubbish. The world is really a flat plate supported on the back of a giant tortoise."
The scientist gave a superior smile before replying, "What is the tortoise standing on?" "You're very clever, young man, very clever," said the old lady. "But it's turtles all the way down!"

This *infinite regression conundrum* appears in many forms throughout history - see https://en.wikipedia.org/wiki/Turtles_all_the_way_down for further discussion.

Our topic is programming languages (not metaphysics) but we have own version of the conundrum.

The Java translation stack

To run a Java program we:

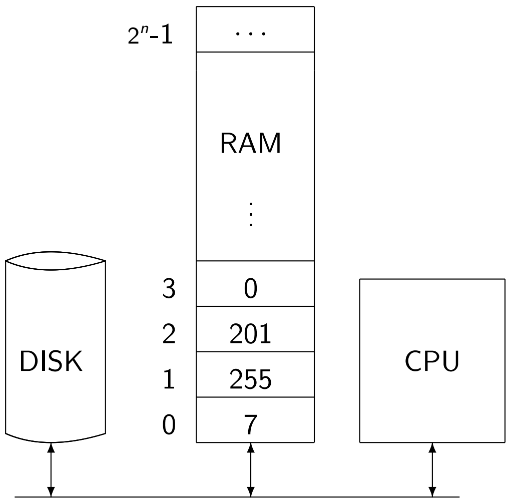
1. Translate the human-friendly Java source code into instructions for a pseudo-computer called the *Java Virtual Machine* using the `javac` compiler
2. Interpret the JVM instructions using the `java` JVM interpreter.

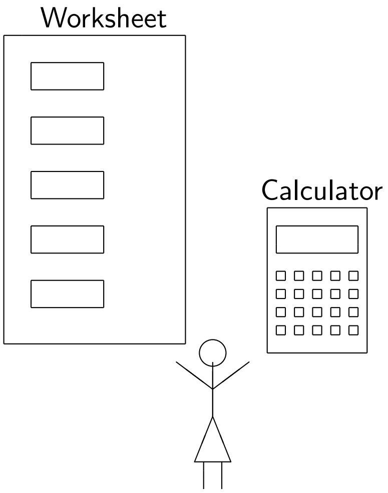
The JVM interpreter is itself a program, not an actual computer. The JVM has its own source code, and might even be written in Java. . .

Let's assume that it is written in C. Then the JVM source code will have been translated into the instructions for some real computer by a C compiler.

How do the instructions for the physical computer get executed? (Or, where do the turtles stop?)

Real computers, automatic and manual





The essence of computing

The human computer uses a four function calculator and a worksheet with formulae, jumps and boxes into which the computed results of formulae may be written.
The automatic computer has a memory into which both the computed results of formulae *and a representation of jumps and of the formulae themselves*, along with a central processor which can execute the calculator's functions and keep track of where we are in the program using a special variable called the *Program Counter* (PC).
Let's define these six instructions which capture the notions of typing a number into the calculator, jumps, and specifying one of the four functions the calculator provides.

Action and code		Syntax				Effect on configuration
Set value	1	set	dst,	k		$mc(dst) \leftarrow k,\ PC \leftarrow PC + 1$
Branch if equal	2	beq	k	src1	src2	$\textbf{if } src_1 = src_2 \textbf{ then } PC \leftarrow k \textbf{ else } PC \leftarrow PC + 1$
Addition	3	add	dst,	src1,	src2	$mc(dst) \leftarrow mc(src_1) + mc(src_2),\ PC \leftarrow PC + 1$
Subtraction	4	sub	dst,	src1,	src2	$mc(dst) \leftarrow mc(src_1) - mc(src_2),\ PC \leftarrow PC + 1$
Multiplication	5	mul	dst,	src1,	src2	$mc(dst) \leftarrow mc(src_1) \times mc(src_2),\ PC \leftarrow PC + 1$
Division	6	div	dst,	src1,	src2	$mc(dst) \leftarrow mc(src_1)\ /\ mc(src_2),\ PC \leftarrow PC + 1$

Programs

The automatic computer's memory is a sequence of pigeonholes, each of which can hold a number which may be changed during the program's execution. The sequence number for each pigeonhole is called it's *address* and never changes.
We can represent programs as sequences of numbers held within the pigeonholes. For instance, here is Java (or C) fragment and its six-instruction-code program

```
a = 3; c = b = 2; while (a != 0) { c = c * b; a-- };
```

```
200: 1, 100, 0 // set location 100 to constant 0
203: 1, 101, 1 // set location 101 to constant 1
206: 1, 102, 3 // set variable a to 3
209: 1, 103, 2 // set variable b to 2
212: 1, 104, 2 // set variable c to 2
215: 2, 231, 102, 100 // branch if a=0 to location 231
219: 5, 104, 104, 103 // multiply b by c and store in c
223: 4, 102, 102, 101 // subtract 1 from 1 and store in a
227: 2, 215, 100, 100 // branch always to location 215 (since 0 = 0)
231: // The rest of the program...
```

Is the six-instruction computer sufficient?

I claim informally that this six instruction computer can execute any Java program.

Logic operations such as `a && b` may be implemented by using constant zero to represent false, and any other value to represent true. We then use subtraction and branching instructions to compute the elements of the logic operation's truth table.

Shifts may be achieved by multiplying or dividing by two.

Array accesses need nasty trickery. Since all of our instructions have constant addresses built into them, when we compute an array access such as `x[y*2] = 7;` we need to calculate the address of the requested array element and then *write that address* into the set instruction `1, ?, 7`. This is *self modifying code* which was once common, but is now thoroughly deprecated because it makes programs very hard to read, and also disrupts the smooth execution of instructions on modern pipelined architectures.

Are all six instructions necessary?

It is not hard to implement multiplication and division using only addition and subtraction, and in fact most computers sold before the mid-1980s did not have hardware multipliers or dividers.

We do not need the **set** instruction since zero is the identity under addition and subtraction, so instead of `set x, k` we could write `sub x, k, zero` where `zero` is the address of a location containing a constant zero (such as location 100 in the earlier example)

More interestingly, we do not need the add instruction either since

$$a + b = a - (-b) = a - (0 - b)$$

So, assuming that we have a spare location `tmp` available, we can replace every occurrence of `add x, a, b` with the sequence `sub tmp, zero, b; sub x, a, tmp`. Now we only have subtraction and branching left.

Cheap computing

If resources were very scarce (perhaps we are a 1940s computing pioneer) or if we enjoy intellectual parlour games, we can go one step further and make the program counter appear as one of the pigeonholes in our memory.

This would allow calculations to directly write the address of the next instruction to be executed into the program counter and so we do not even need a branch instruction. Now we have a single-instruction computer: everything is a subtraction.

Many early computers were *bit serial*. That means that, say, a 16-bit subtraction was computed one bit at a time, working across the operands from least significant to most significant bit.

I know how to make such a CPU using only 30 switching elements and some one bit memories, but that is a story for another module.

Expensive computing

These minimal computers are fun to design, but not much fun to program. Broadly speaking, the first forty years of computer architecture research focused on closing the so-called *semantic gap* between human-friendly high-level programming languages, and the low level *machine code* by adding new, expressive hardware instructions.

An important part of those developments was the use of *addressing modes* that allowed, for instance, an entire array index calculation to be performed during the fetching of an instruction's operands – the constant address of our instruction sets was expanded to allow quite baroque expressions.

In the mid-1980s a revolution occurred. Called *Reduced Instruction Set* (RISC) architecture (though I think *Reduced Addressing Mode* (RAMP) would have been a more accurate acronym), this new style eschewed any feature which made it hard to *pipeline* operations. Examples include MIPS and ARM: the only pre-RISC architecture being commercially developed is the Intel ISA x86 line.

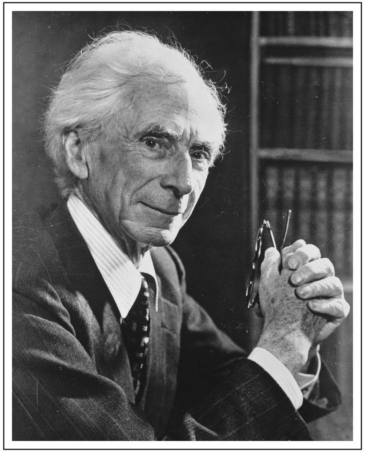
Languages all the way down... with the hardware as a base case

As computer scientists we know that the solution to recursive conundrums like the turtle model of reality is to have a *base case*

A hardware instruction set like the ones we have just discussed forms the base case for our stack of translations. Computer architecture research is about finding the best base case. The definitive textbook is Hennessy and Patterson[1] but that is a story for a different module.

What we have seen, though, is that we can expand and contract the capabilities of the hardware without losing generality, and that we can translate programs from, say, our six instruction architecture to our four, two or one instruction architectures, easily and *without changing the upper translation layers at all*.

[1] https://www.elsevier.com/books/computer-architecture/hennessy/978-0-12-811905-1



Bertrand Russell 1872–1970

https://en.wikipedia.org/wiki/Bertrand_Russell

1B Utility and power in languages and their processors

We now have a suspicion that our programming languages might somehow all be the same, in that we can translate between them. If that is so (and we shall return to that question later), why are there so many languages? I think there are four main drivers.

Culture	Human (natural) languages seem to develop alongside cultures - groups of people working and living together share a common language which over time can develop into a separate dialect. ANSI C, ANSI C++ and even Java might be seen as derivatives of the original C.
Commerce	Sometimes manufacturers attempt to lock customers in with proprietary languages. This was common in the early years, and still happens at the fringe: for example Nvidia's CUDA libraries require their hardware.
Coolness	Fashion can dominate decisions - programmers love the latest cool thing, especially if it is not completely different to what they already know.
Conciseness	Verbose programs are tiring programs, and much of programming language history can be seen as the pursuit of elegant notations to express complex notions. Of course, extremely concise notations can be tiring too (which is perhaps why many folk are put off mathematics).

Reuse encourages layering

New software products are built on the foundations of existing systems, since *ab initio* implementation of, say, network protocols, encryption standards and graphics systems in machine code would be immensely costly.

This reuse might be in the form of libraries for use with an existing language, or it might involve a complete new *Domain Specific Language* (DSL)

For instance, a game engine might have its own scripting language. The script interpreter could be written in one architecture's machine code, but then it would only run on that architecture. A portable script engine could be written in Java. Now we have an interpreted language being executed by a program written in Java, which is itself being interpreted...

That game might also save configuration data in XML, access a database using SQL, use HTML5 graphics in the browser, and have a manual written (like these slides) in a typesetting systems such as $\LaTeX$.

We have an entire ecosystem of interacting DSLs

Internal and external DSLs

Libraries such as the Java FX graphics subsystem or the Java MIDI synthesizer offer a well defined set of capabilities that address specific domains - graphics and music-making in these cases.

We can think of each library as a Domain Specific Language in which the features are presented as method calls to Java objects, and we can build applications by writing Java programs that string together lots of calls to these libraries with general purpose Java control flow code.

Alternatively, we could make up a new syntax with its own control flow and data declarations, and hide all the (sometimes verbose) details of the Java scaffolding.

The first approach – a library called from a general purpose language - is sometimes called an *internal* DSL. The full-fat version with its own syntax and processor is called an *external* DSL.

We can view the external DSL as being a wrapper around the underlying internal DSL: the purpose of the external DSL's special language is to provide *concise* and sometimes *cool* ways of specifying applications.

The DSL maintenance trap

I left out one common source of new languages in the first slide: ego.

Programming languages are fascinating artifacts, and many programmers have explored creating their own language. The Python and C++ languages emerged from experiments by individuals.

Reasoning about the interaction of language features is *hard*, especially when trying to extend an existing design.

Sometimes a hot-shot programmer within an organisation is a bit bored and decides to pressure management to allow a new DSL to be developed in which to encode their business processes.

Management wants to retain their expensive hot shot programmer who is thus allowed to build something, usually using tools and techniques that were optimised for 1970s architectures with tiny memories, and with a sticky mess of glue logic to paper over the cracks.

Eighteen months later the hot-shot programmer leaves anyway, and the company is left with an opaque system at the heart of their business that other programmers are frightened to meddle with.

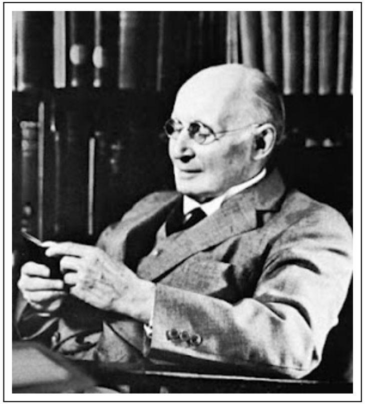
The DSL maintenance solution

There is another way

This course will teach you how to specify languages and their processors with concise very-high-level notations that allow you to:

- experiment with, and reason about, the interactions of language features,
- provide clear specifications to the engineers that have to maintain your work after you have moved on, and
- as a bonus, allow complete language interpreters to be generated automatically from your specifications which you can use for testing, and which might be fast enough for production use.

We hope for concise, complete and directly executable language specifications



Alfred North Whitehead 1861–1947

https://en.wikipedia.org/wiki/Alfred_North_Whitehead

1C What do we mean by *formal*?

The history of science across the twentieth century is one of increasingly rigorous mathematical modelling.

In Physics we have the development of the Bohr atomic model, leading to quantum theory, Quantum Electrodynamics and the standard model of particle physics.

In Biology we have the development of chemical bond modeling leading to the elucidation of the structure of DNA and molecular biology.

In the future, medicine will increasingly be based on the application of statistical machine learning techniques to the unravelling of biochemical pathways, and their impact on the organism leading to new understanding of disease and opportunities for therapeutic intervention.

Formalising mathematics

Mathematics itself is a discipline that benefits from mathematical analysis. Up until suprisingly recently, all mathematics was expressed in prose which is hard to analyse. For instance, the now universally-understood symbol for equality $=$ was only introduced in the mid-sixteenth century, during the reign of the English Queen Elizabeth I, and it was a further 150–200 years before it's use became universal.

As mathematical language moved from (rather fuzzily defined) words to symbols, some people began to wonder whether mathematics is in fact anything more than the manipulation of symbols.

The so-called formalist school of thought has it that all we have are strings of symbols which can be manipulated according to certain rules. The idea is that mathematical statements are simply syntactic forms which have no meaning unless they are given a semantics.

Philosophers of mathematics do not all agree. You can read about some of the debates here `https://en.wikipedia.org/wiki/Formalism_(philosophy_of_mathematics)`

Formalising the notion of computation

Attempts to define computational tasks as well-defined procedures are very old. We shall study Euclid's 2,300 year old Greatest Common Divisor algorithm in a later section. Attempts to actually automate computation (as opposed to designing aids to for human computers) date to Babbage's multiple implementations of the Method of Differences, beginning 200 years ago.

Limits to computation became a topic of great interest in the 1920's. Hilbert, one of the most prominent of the formalists, set a challenge in 1928 called the *Entscheidungsproblem* or *decision problem*. One interpretation of this problem is that it asks whether there are statements in a logic that cannot be deduced from the axioms by any algorithm.

To answer the question, a formal model of 'algorithm' was needed. Alonzo Church's 1935 λ-calculus and Turing's 1936 *Turing machine* are equivalent minimalist models of computation which were used to show that *undecidable* decision problems exist.

More at `https://en.wikipedia.org/wiki/Entscheidungsproblem`

Programming languages and the Entscheidungsproblem

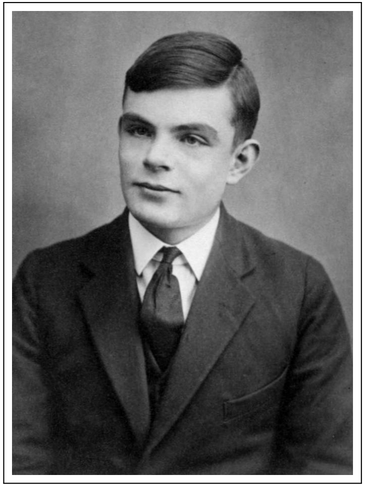
So, a Turing machine is a formal *game* with a configuration and transition rules that, according to the Church-Turing thesis, can perform any computation that a more complex device can. We could view it as an *abstraction* of our real computers, though of course it predates their design.

A key result of Turing's is that there exist *Universal Turing Machines* which can simulate any other Turing machine. They work by essentially reading in a description of the machine to be simulated, and then interpreting the input under those rules.

Many authorities believe that this notion directly inspired Von Neumann description of computers. Others recognise its significance, but believe that pragmatic, engineered stored-program computers developed independently of these theoretical formalisms – certainly Maurice Wilkes told me over dinner one night that his team knew nothing of Turing's formalisms.

Whatever the truth, it is the formal proof of existence of UTMs that 'explains' our insight that programming languages equipped with basic flow control are equivalent, in that one can be translated into another, or simulated in another.

More at https://en.wikipedia.org/wiki/Universal_Turing_machine



Alan Turing 1912–1954

https://en.wikipedia.org/wiki/Alan_Turing

1D Conway's Game of Life, formalised

Conway's Game of Life (CGL) was popularised by Martin Gardner in the October 1970 edition of Scientific American

It is a game with no 'players'. The board is ordinary square ruled graph paper. Each square on the paper may be filled or it may be empty.

The game proceeds in generations: an entire board is evaluated, and the rules of the game specify which squares will switch from filled to empty, which from empty to filled, and which will stay the same.

There is no end point: the generations continue to evolve without limit, although they may converge to constant or repeating patterns that will never develop new behaviour.

CGL is interesting because a set of very simple rules generates complex, semi stable populations. It is an example of *emergent behaviour*

Cellular automata

CGL is an example of a cellular automaton. We'll now give some more precise technical language for talking about such automata.

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 transition rule which specifies the evolution of the cell labels in discrete time steps $\Gamma_0, \Gamma_1, \ldots$ called generations.

It can be useful to think of Γ as having structure, in the sense that some cells are 'adjacent', and for the transition rule to be a function over a 'neighbourhood' $f(p_1, p_2, \ldots, p_k)$ with signature $f : \Sigma \times \Sigma \times \ldots \times \Sigma \rightarrow \Sigma$ where the p_k are the elements of the neighbourhood.

The CGL transition rule

The blue paragraph on the previous slide is *woolly*: we have not explained how the neighbourhoods are defined or used, and as a result we could not automatically generate an implementation.

In CGL, the cells in Γ are arranged as an unbounded two-dimensional rectangular array; Σ is the set $\{0, 1\}$ and the transition rule defines the new state σ' of a cell 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:

$$\sigma' = \begin{cases} 1, & \sigma = 0, P = 3 & \text{(birth)} \\ 0, & \sigma = 1, P < 2 \vee P > 3 & \text{(death through loneliness or overcrowding)} \\ \sigma, & \text{otherwise} & \text{(stasis)} \end{cases}$$

This rule induces a *transition relation*: the set $\{(\Gamma_i, \Gamma_j)\}$ such that the transition rule can construct Γ_j from Γ_i.

The CGL transition relation is a *function* because only one new board can be generated from each Γ_i. Contrast that with, say, chess in which there may be hundreds of moves from each position.

Neighbourhoods

The transition rule is still woolly: what is the meaning of *eight-connected neighbours*? Geometrically, we mean that given the plane tesselated by rectangles, the 8C-neighbours of cell C are the eight cells that touch C, including four that only touch at their corners.

So if we are looking at the cell at coordinates (x, y), then we want to count the total population of the cells at locations (x-1,y-1), (x,y-1), (x+1,y-1), (x-1,y), (x+1,y), (x-1,y+1), (x,y+1), (x+1,y+1)

A more concise way of saying this is

$$P_{x,y} = |\Gamma_{x,y}|$$

where

$$\Gamma_{x,y} = \{(x+i, y+j) \mid (x, y) \in \Gamma, -1 \leq i, j \leq 1, (i \neq 0 \wedge j \neq 0), (x+i, y+j) \in \Gamma\}$$

Translation to Java

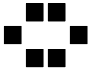
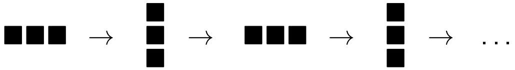
Substituting $\Gamma_{x,y}$ into the expression for $P_{x,y}$, we have

$$P_{x,y} = |\{(x+i, y+j) \mid (x,y) \in \Gamma, -1 \leq i,j \leq 1, (i \neq 0 \wedge j \neq 0), (x+i, y+j) \in \Gamma\}|$$

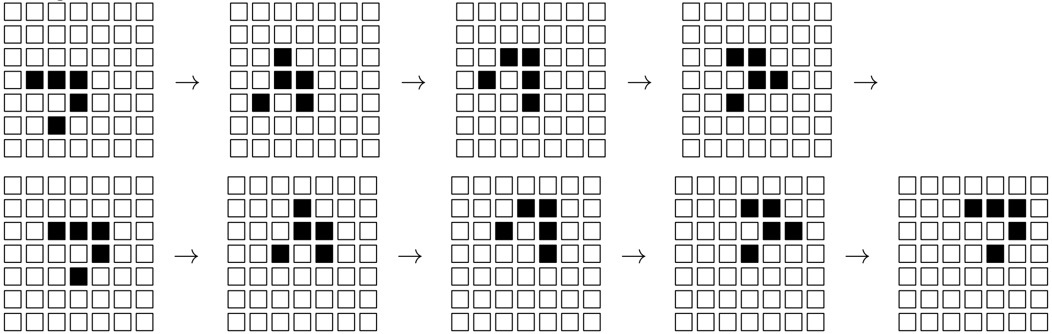
and here is a fairly direct translation from the formal language of set theory, to a formal language of programs (using Java in this case).

```
int P(x,y) {
  int ret = 0;
  for (i = −1, i <= 1, i++)
    for (j= −1, j <= 1, j++)
      if (i != 0 && j != 0) ret += gammaCellState(x + i, y + j);
  return ret;
}

int gammaCellState(x,y) {
  if ( /* The cell (x,y) is in Gamma */)
    return 1;
  return 0;
}
```

Some CGL patterns

Block: Beehive:  Blinker: 

The glider:



Naïve implementation of CGL using an array

So far, our `gammaCellState()` function remains undefined. It depends on the representation we use for the Life board.

An obvious, though flawed, way to implement Conway's Game of Life is to use a two dimensional array of Boolean values or integers. Section 1.7.3 of the book gives a full implementation that you should study.

Here are some weaknesses of the array-based approach

- The size of an array is fixed at the time it is created
- The number of calls per generation-transition to the population function P is the number of cells in the array. This means that the computational demand is $O(XY)$ where X and Y are the board has $X \times Y$ cells.
- Arrays have *edges* because the allowed values of x and y have finite bounds. However, our transition function is defined over the integers, and so we do not know what to do if the neighbourhood function wants to look up an element that is outside of the array. This this implementation is *incorrect*. Consider two gliders that fly outside of the finite bounds before colliding and sending a fragment back in: our implementation will diverge from the formal definition.

A set based implementation without ‘edges’

Life grids are often sparsely populated. We would like an implementation where the processing times is $O(|L|)$ where L is the set of cells that are *live*.

We also want an implementation that avoids the finite bounds of the array. Of course, all real computer memories are finite so eventually we shall run out of memory, but we can do much better then the array representation.

In general, cellular automata can have multiple states, not just ‘filled’ and ‘empty’, so let us represent one generation of an automaton in the plane as a set of triples $\Gamma_i = \{(x, y, \sigma)\}$ with $\sigma \in \Sigma$, the set of possible states. We choose a particular element of Σ to be the ‘background’ and omit those triples from our representation of Γ_i

For CGL, this reduces to representing the state of the board with the set of coordinate pairs of each live cell. You will find an implementation in this style in the book, which you should study. We extract an element from the current Γ, then check its 8-connected neighbours to see if it should live, and if any neighbour births occur.

Using a visited set to avoid recomputations

We can go further. The set based implementation computes Γ_{i+1} by extracting each element (x, y) from Γ_i and checking its eight connected neighbours for two purposes:

1. To decide if $P(x, y)$ is in the range $2 - 4$ in which case (x, y) is entered into Γ_{i+1}
2. To check the eight connected neighbours of cell (x, y) to see if they are not in Γ_i and if so, if their neighbour population is 3 in which case a new cell is 'born'.

Now, an empty cell can be a neighbour of up to eight filled cells, and step 2 will be executed for each of those filled neighbours

We can save this computation by keeping a set C of 'checked' cells and only executing step 2 on cells that are not in C. You will find a full implementation in the book.

This is an example of the technique called *memoisation* in some sources, in which we cache the result of calculations that might otherwise be performed multiple times.

How is this relevant to programming language design and implementation?

There are two lessons from this section.

Firstly, *thinking about things in an abstract way can reveal insights*. By moving away from a geometric understanding, and instead representing each CGL configuration (generation) as a set of tuples we were able to find implementations that were more efficient in both time and space.

Secondly, the idea of a system that has *configurations* which we step between under the control of a *transition relation* is a widely applicable formal notion that is susceptible to mechanisation.

We can use the transition relation idea to think about automata, such as the deterministic and nondeterministic finite automata that are studied in the first year of most computer science degrees, as well as network protocols and other more complex patterns of behaviour.

Programming languages as transition relations

There is a sense in which a programming language, such as Java, defines a transition relation between *configurations* of a computer.

We think about these configurations as the combined state of *all* of the mutable memory elements of the computer by which all of the main RAM memory, the complete state of the disks, the program counter and every other mutable memory element within the system - its a bit like an enormous CGL board with one cell for each memory bit.
As we execute lines of a Java program, the machine configuration changes, and in fact since our configuration includes *all* of the mutable memory elements, the evolving sequence of configuration *completely* defines the behaviour of the machine.
The Java transition relation includes (C_i, C_j) iff C_i is a machine configuration generatable by some Java program, and C_j may be reached from C_i by executing one Java 'step' – say the evaluation of a single operator.

If a program is completely sequential, then there is only ever at most one successor configuration available, that is the transition relation is a partial function. For a concurrent program or a non-deterministic sequential program, then there may be multiple successors. So sequential programs have transition functions as does CGL, and concurrent programs look more like chess with multiple possible 'moves' to multiple possible successor configurations.

John Horton Conway 1937–2020

https://en.wikipedia.org/wiki/John_Horton_Conway

1E The rewrite model: Euclid's Greatest Common Divisor

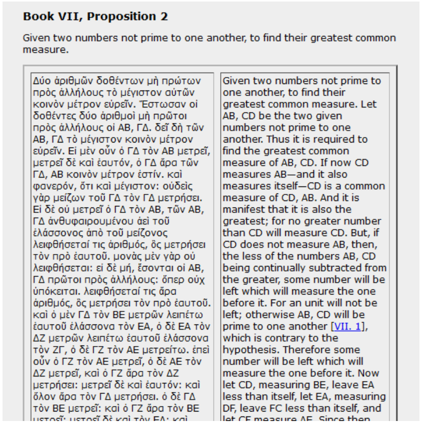
In volume 2 of his *magnum opus* The Art of Computer Programming, Donald Knuth calls Euclid's Greatest Common Divisor the granddaddy of all algorithms, because it is the oldest nontrivial algorithm that has survived to the present day.

The algorithm appears in Book VII of Euclid of Alexandria's *Elements* which was probably written around 2,300 years ago. It is not clear if Euclid collected previous results, or originated them, and in fact it is not entirely clearly that Euclid was a real person... but *Elements* set a pattern for formal presentation of mathematical arguments that still is widely used today.

For our purposes, it has the great merit of being the shortest program I know of that actually does something useful and has non-sequential control flow. We shall use it as a running example.

The text of *Elements*

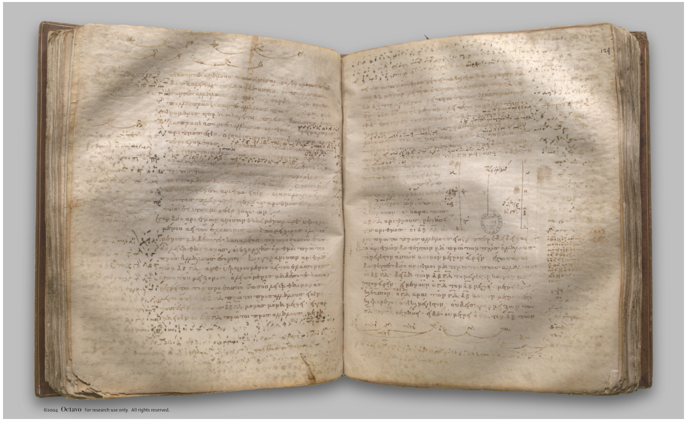
https://www.claymath.org/library/historical/euclid/files/elem.7.2.html



Book VII, Proposition 2

Given two numbers not prime to one another, to find their greatest common measure.

Δύο ἀριθμῶν δοθέντων μὴ πρώτων πρὸς ἀλλήλους τὸ μέγιστον αὐτῶν κοινὸν μέτρον εὑρεῖν. Ἔστωσαν οἱ δοθέντες δύο ἀριθμοὶ μὴ πρῶτοι πρὸς ἀλλήλους οἱ ΑΒ, ΓΔ. δεῖ δὴ τῶν ΑΒ, ΓΔ τὸ μέγιστον κοινὸν μέτρον εὑρεῖν. Εἰ μὲν οὖν ὁ ΓΔ τὸν ΑΒ μετρεῖ, μετρεῖ δὲ καὶ ἑαυτόν, ὁ ΓΔ ἄρα τῶν ΓΔ, ΑΒ κοινὸν μέτρον ἐστίν. καὶ φανερόν, ὅτι καὶ μέγιστον: οὐδεὶς γὰρ μείζων τοῦ ΓΔ τὸν ΓΔ μετρήσει. Εἰ δὲ οὐ μετρεῖ ὁ ΓΔ τὸν ΑΒ, τῶν ΑΒ, ΓΔ ἀνθυφαιρουμένου ἀεὶ τοῦ ἐλάσσονος ἀπὸ τοῦ μείζονος λειφθήσεταί τις ἀριθμός, ὃς μετρήσει τὸν πρὸ ἑαυτοῦ. μονὰς μὲν γὰρ οὐ λειφθήσεται: εἰ δὲ μή, ἔσονται οἱ ΑΒ, ΓΔ πρῶτοι πρὸς ἀλλήλους: ὅπερ οὐχ ὑπόκειται. λειφθήσεταί τις ἄρα ἀριθμός, ὃς μετρήσει τὸν πρὸ ἑαυτοῦ. καὶ ὁ μὲν ΓΔ τὸν ΒΕ μετρῶν λειπέτω ἑαυτοῦ ἐλάσσονα τὸν ΕΑ, ὁ δὲ ΕΑ τὸν ΔΖ μετρῶν λειπέτω ἑαυτοῦ ἐλάσσονα τὸν ΖΓ, ὁ δὲ ΓΖ τὸν ΑΕ μετρείτω. ἐπεὶ οὖν ὁ ΓΖ τὸν ΑΕ μετρεῖ, ὁ δὲ ΑΕ τὸν ΔΖ μετρεῖ, καὶ ὁ ΓΖ ἄρα τὸν ΔΖ μετρήσει: μετρεῖ δὲ καὶ ἑαυτόν: καὶ ὅλον ἄρα τὸν ΓΔ μετρήσει. ὁ δὲ ΓΔ τὸν ΒΕ μετρεῖ: καὶ ὁ ΓΖ ἄρα τὸν ΒΕ μετρεῖ: μετρεῖ δὲ καὶ τὸν ΕΑ: καὶ	Given two numbers not prime to one another, to find their greatest common measure. Let AB, CD be the two given numbers not prime to one another. Thus it is required to find the greatest common measure of AB, CD. If now CD measures AB—and it also measures itself—CD is a common measure of CD, AB. And it is manifest that it is also the greatest; for no greater number than CD will measure CD. But, if CD does not measure AB, then, the less of the numbers AB, CD being continually subtracted from the greater, some number will be left which will measure the one before it. For an unit will not be left; otherwise AB, CD will be prime to one another [VII. 1], which is contrary to the hypothesis. Therefore some number will be left which will measure the one before it. Now let CD, measuring BE, leave EA less than itself, let EA, measuring DF, leave FC less than itself, and let CF measure AE. Since then



The GCD algorithm as a program

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

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

output(a);
```

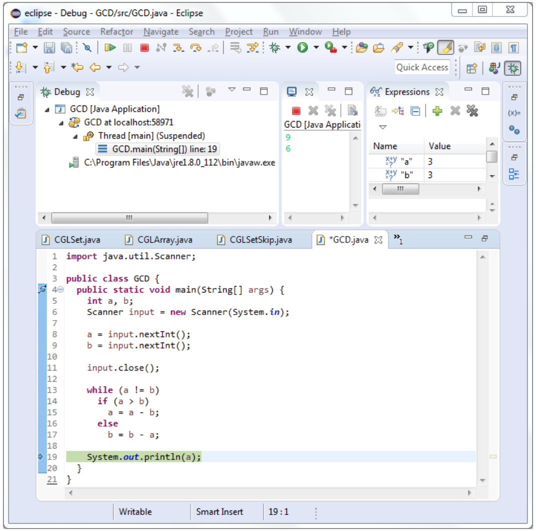
GCD in Java

```
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();

      while (a != b)
         if (a > b)
            a = a - b;
         else
            b = b - a;

      System.out.println(a);
   }
}
```

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. Here is a screenshot of the Java CGD program being run under the debugger within Eclipse.

We have the 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 line 19 of the code window has a small pointer arrow on the left hand side and is highlighted.


eclipse - Debug - GCD/src/GCD.java - Eclipse
File Edit Source Refactor Navigate Search Project Run Window Help
Quick Access
Debug
Expressions
CGLSet.java
CGLArray.java
CGLSetSkip.java
*GCD.java

```
GCD [Java Application]
  GCD at localhost:58971
    Thread [main] (Suspended)
      GCD.main(String[]) line: 19
  C:\Program Files\Java\jre1.8.0_112\bin\javaw.exe
```

```
GCD [Java Applicati
9
6
```

Name	Value
"a"	3
"b"	3

```
 1 import java.util.Scanner;
 2
 3 public class GCD {
 4   public static void main(String[] args) {
 5     int a, b;
 6     Scanner input = new Scanner(System.in);
 7
 8     a = input.nextInt();
 9     b = input.nextInt();
10
11     input.close();
12
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 }
```

Writable | Smart Insert | 19 : 1

Rewriting – another way to think about program execution

Here is a short program.

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

The first thing the program does is output the value 3. We can represent this by constructing an 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

$\langle$`"output(3); output(10+2+4);", []`$\rangle$

means the same thing as

$\langle$`"output(10+2+4);", [3]`$\rangle$

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. We can represent each step of a program's execution by a pair comprising the current output and a rewritten program that represents only what remains to be done.

```
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.

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

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

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

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

```
[3, 16]
```

Rewriting vs. using a program counter

The standard approach – using an unchanging program and a Program Counter that points to the next instruction to be executed – is efficient and effective, and basis of almost all hardware implementations of computers.

The rewriting approach is much more amenable to mathematical analysis. This is because when we want to, say, prove that a program terminates, we can use *proof by induction on the length of the program*. Induction proofs typically need some measure that is increasing or decreasing built into the induction hypothesis.

A closely related technique is *structural induction* in which we show that the effect of a nested structure (perhaps nested control flow in Java) is the union of the effects of its component parts.

We are not going to be proving program properties on this course. However we are going to build syntax analysers and interpreters *compositionally*, for which rewriting can support clear, uncluttered reasoning, producing short, maintainable specifications and for which structural induction proofs could, in principle, be constructed.

A reduction semantics for linear code is straightforward, but we need to think carefully about loops. One approach is to use this *program identity*

```
while booleanExpression do statement;
```

is *always* the same as

```
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

```
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.

A rewrite evaluation of GCD with input [6, 9]

Procedural programs typically have input, output and variables. We use lists of values to represent the input and output, as we have already seen. These lists are called *semantic entities*, and their rôle is to leep track of *side-effects* of the program whilst it is being rewritten away.

To represent the variables, we can use a *map*. A map is a set of bindings. In Java we assign, say, 6 to variable a by writing `a = 6;`. When we come to reduce that piece of code, we create a binding $a \mapsto 6$ in the store, giving the set of bindings $\{a \mapsto 6\}$

A *rewrite configuration* is a tuple comprising the program term to be rewritten and all of the semantic entities with their current values. The set of semantic entities is fixed for a given programming language. For a purely functional programming language, there are *no* semantic entities because program rewriting is all that they allow.

For our GCD program, configurations will be of the form $\langle \alpha, \sigma, \beta, \theta \rangle$ to represent the input, store, output and program term, respectively.

The 36 following steps completely execute GCD for inputs 6 and 9. Each step *roughly* corresponds to one machine instruction.

Start of trace

Initialise variables from input

⟨[6, 9], {}, [ ], a:=input(); b:=input(); **while** a!=b **if** a>b a:=a−b; **else** b:=b−a; output(a);⟩

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

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

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

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

⟨[ ], {$a \mapsto 6, b \mapsto 9$}, [ ],
  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);⟩

⟨[ ], {$a \mapsto 6, b \mapsto 9$}, [ ],
  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);⟩

⟨[ ], {$a \mapsto 6, b \mapsto 9$}, [ ],
  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);⟩

⟨[ ], {$a \mapsto 6, b \mapsto 9$}, [ ],
  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);⟩

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

⟨[ ], {$a \mapsto 6, b \mapsto 9$}, [ ],
  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);⟩

$\langle [\,], \{a \mapsto 6, b \mapsto 9\}, [\,],$ **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);$\rangle$
$\langle [\,], \{a \mapsto 6, b \mapsto 9\}, [\,],$ **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);$\rangle$
$\langle [\,], \{a \mapsto 6, b \mapsto 9\}, [\,],$ **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** $\{a \mapsto 6, b \mapsto 9\}$
$\langle [\,], \{a \mapsto 6, b \mapsto 9\}, [\,],$ b:=b−a; **while** a!=b **if** a>b a:=a−b; **else** b:=b−a; output(a);$\rangle$
$\langle [\,], \{a \mapsto 6, b \mapsto 9\}, [\,],$ b:=b−6; **while** a!=b **if** a>b a:=a−b; **else** b:=b−a; output(a);$\rangle$
$\langle [\,], \{a \mapsto 6, b \mapsto 9\}, [\,],$ b:=9−6; **while** a!=b **if** a>b a:=a−b; **else** b:=b−a; output(a);$\rangle$
$\langle [\,], \{a \mapsto 6, b \mapsto 9\}, [\,],$ b:=3; **while** a!=b **if** a>b a:=a−b; **else** b:=b−a; output(a);$\rangle$
Rewrite using `while p s` $\rightarrow$ `if p { s ; while p s }`
$\langle [\,], \{a \mapsto 6, b \mapsto 3\}, [\,],$ **while** a!=b **if** a>b a:=a−b; **else** b:=b−a;output(a);$\rangle$
Evaluate $a \neq b$ **with store** $\{a \mapsto 6, b \mapsto 3\}$
$\langle [\,], \{a \mapsto 6, b \mapsto 3\}, [\,],$
    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);$\rangle$
$\langle [\,], \{a \mapsto 6, b \mapsto 3\}, [\,],$
    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);$\rangle$
$\langle [\,], \{a \mapsto 6, b \mapsto 3\}, [\,],$
    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);$\rangle$
$\langle [\,], \{a \mapsto 6, b \mapsto 3\}, [\,],$
    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** $\{a \mapsto 6, b \mapsto 3\}$
$\langle [\,], \{a \mapsto 6, b \mapsto 3\}, [\,],$ **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);$\rangle$
$\langle [\,], \{a \mapsto 6, b \mapsto 3\}, [\,],$ **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);$\rangle$
$\langle [\,], \{a \mapsto 6, b \mapsto 3\}, [\,],$ **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` $\to$ `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

Euclid of Alexandria (perhaps) 2300-2200 years ago?

https://en.wikipedia.org/wiki/Euclid

2 Rewriting systems - concepts and terminology

Rewriting is central to much of this course.

In this section we are going to develop the terminology of rewriting systems for both strings and tree-like terms, and see how the choice of *rewriting strategy* affects the results.

Section	Topic
2A	Transition relations and abstract rewriting
2B	String rewriting
2C	Term rewriting
2D	Rewriting strategies
2E	Implementing terms in Java
2F	Types and operations: the ARTValue system

2A Transition relations and abstract rewriting

In this section we are going to develop some technical language and some graphical techniques for thinking about and specifying the application of rewriting under relations to programming language specification.

Our relations will be used to define valid *rewrites* on both strings and trees.

We shall use *string* rewriting to develop *external syntax parsers* for our languages which will check that a user's program is well formed, and then output a tree like representation of the structure of the string in what we call the *internal syntax* of the language. In some places these trees are called *parse trees* or *abstract syntax trees*, both terms that I rather dislike.

We shall then use *tree* rewriting to transform those internal syntax trees into (usually) simpler forms, modelling execution of the program by recording side-effects in *semantic entities*.

Relations

A *binary relation* $\odot$ between two sets P and Q is a set of ordered pairs (p, q) where $p \in P$ and $q \in Q$; hence $\odot$ is a subset of the cartesian product $P \times Q$

P is called the *domain* of $\odot$ and Q is the *codomain*

We often write $x \odot y$ to mean $(x, y) \in \odot$. Hence the familiar Java notation $x > y$ which returns `true` iff x is greater than y, that is (x, y) is in greater-than, $>$. By extension, you will see the notation xRy to mean that x is related to y under R

If P and Q are the same set, then we say that the $\odot$ is a relation *over* P. These kinds of relations are sometimes called *homogeneous*. Our rewriting relations will be homogenous relations

Operations over relations

Since relations are just sets, the usual set operations may be applied to them, so for instance:

the union of two relations R and S written $R \cup S$ is $\{(x, y) \mid xRy \vee xSy\}$ with identity $\emptyset$

the intersection of R and S, $R \cap S$ is $\{(x, y) \mid xRy \wedge xSy\}$ with identity $X \times Y$

The composition of R and S, written $R \circ S$ is $\{(x, z) \mid xRy \wedge ySz\}$

The converse of R, written R^{-1} is $\{(x, y) \in X \times Y \mid yRx\}$

Other names for the converse relation include the transpose and the inverse relation

Properties of relations R between domain X and codomain Y

A relation R over X and Y is:

injective or *left unique* if $\forall x, z \in X, \forall y \in Y$, if $xRy \wedge zRy$ then $x = z$

functional or *right unique* if $\forall x \in X, \forall y, z \in Y$, if $xRz \wedge yRz$ then $y = z$

If the domain X and the codomain Y are specified, then R can also be

surjective, *onto* or *right total* if $\forall y \in Y$, there exists an $x \in X$ such that xRy

serial or *left total* if $\forall x \in X$, there exists a $y \in Y$ such that xRy

A *function* is a binary relation that is functional and serial

An *injection* is a function that is injective

An *surjection* is a function that is surjective

An *bijection* is a function that is injective and surjective

Homogeneous relations over a set X

Where the domain and codomain of R are the same (that is R is homogeneous), then R may have these properties

reflexive	$\forall x \in X, xRx$
symmetric	$\forall x, y \in X$, if xRy then yRx
antisymmetric	$\forall x, y \in X$, if $xRy \wedge yRx$ then $x = y$
transitive	$\forall x, y \in X$, if $xRy \wedge yRz$ then xRz

An equivalence relation is a relation that is reflexive, symmetric and transitive

Equivalence relations are often written $x \sim_R y$ or just $x \sim y$ if the context is clear

The *identity relation* is $\{(x, x) \mid x \in X\}$

Closures of homogeneous relations

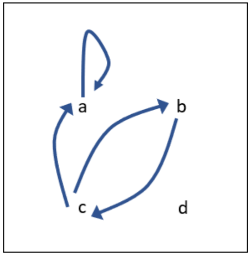
If given an arbitrary homogeneous relation R over X then we can derive these relations from it. The:

reflexive closure	$R^{=}$ is the smallest reflexive relation over X that includes R
transitive closure	R^{*} is the smallest transitive relation over X that includes R

The graph of a relation

A graph is completely characterised by its set of edges, and each edge is completely characterised by its start and end point. Hence a graph is just a set of ordered pairs. Hence, it is clear that graphs and relations are really the same thing.

We have three 'natural' ways to represent a relation: as a set of arrows between set elements, as an adjacency matrix and as a set of ordered pairs. So these three representations are all of the same underlying relation:



	codomain			
domain	a	b	c	d
a	1			
b			1	
c	1	1		
d				

$\{(a, a), (b, c), (c, a), (c, b)\}$

Transition systems and abstract rewriting systems

A *unlabeled transition system* is a pair $(S, \to)$ where S is a set of states and $\to$ is a relation over S. We think of the transition system as defining paths through the relation's graph which may be traced out by following individual transitions.

An *abstract rewriting system* (ARS) is a pair $(A, \to)$ where A is a set of objects to be rewritten and $\to$ is a relation over A. We think of the ARS as defining a sequence of rewrites of some initial object.

Clearly transition systems and abstract rewriting systems are the same thing in terms of the underlying mathematics: the distinction is really only whether we think about a system of states that is being traversed, or an object that is being mutated. Sometimes it is convenient to switch between these two interpretations when considering a single system

Normal forms and termination

In a rewriting system, an object $a \in A$ is *rewritable* if there exists some $b \in A$ such that $a \to b$. If an object is *not* rewritable then it is called a *normal form*.

Now, let's think about sequences of rewrites, or equivalently tracing out a series of transitions. We write $x \xrightarrow{*} y$ to denote 'zero or more' steps of the transition/rewrite relation[2]. In fact $\xrightarrow{*}$ is itself a relation: it is the reflexive-transitive closure of the underlying relation $\to$.

If we have cycles in the relation graph, then we have the potential for never-ending sequences. In many applications this is undesirable since it means that rewriting would not terminate. However, useful programs exist that do not terminate (web servers for instance) so reasoning about termination within the rewrite relation can require care.

[2]This use of $*$ to represent zero or many applications of something is usually called a *Kleene-star* after the logician Stephen Kleene. You may recognise its use as a wildcard in Un*x and Windows command line file name specifications

Normal termination and *stuck* transitions

Often we wish to distinguish between well formed paths and 'bad' paths. To do that we may provide a set of *terminating* or *accepting* states which mark the end of valid paths.

When defining programming languages, we usually expect the final configuration of a rewrite to be either a value such as a number, or a special word that indicates that program execution has been exhausted. In our systems we use the word `done` but `skip` is a traditional alternative in semantics work, and the Python language provides `pass`.

So, for many practical language processors we specify some set of values and `done` as the accepting states. We shall describe the ART value system (which informally speaking is a set of pre-defined types, literals and builtin operations) in a later lecture.

If our system reaches a state with no out-edges that is *not* in the accepting set, then we say that the system is *stuck*

Confluence

In a debugged language specification, programs should never get stuck. The appearance of a stuck state x means that we arrived somewhere we didn't intend since we haven't given any rules for progressing from x, and x is not a terminating state. In practice, this usually means that we have either forgotten to handle one of our cases, or we wrote nonsense somewhere and sent our system off into an unintended place.

Let us assume, then, that stuck states never appear. What else could go wrong? Well relations are rather general things, and it is perfectly possible for a state/object x to have paths out of it that lead to two different terminating states. If we are using rewrite systems to model programs, then that situation would suggest that there might be more than one outcome of a program – a most unhappy situation.

A system is called *confluent* if, given $x \xrightarrow{*} v$ and $x \xrightarrow{*} w$ there exists z such that $v \xrightarrow{*} z$ and $w \xrightarrow{*} z$. Clearly confluence is a desirable property. Unfortunately confluence is undecidable in general.

Keeping things simple

On this course, we are going to model sequential programs only. Modeling concurrent systems using these techniques is viable, but technically more difficult.

Moreover, our systems are going to mostly eschew non-standard termination, by which I mean that arbitrary jumps and goto statements will not be allowed. Again, they *can* be modeled but with some difficulty.

To achieve this simplicity, we are going to (a) use *deterministic* transition systems in which there is at most one transition out of any state, and (b) insist on *structural* descriptions where the specification of semantics exploits the tree-like structure of the internal syntax representation emitted by our parsers.

Once you have completed this course you will be able to move on to more sophisticated features of real programming languages, such as multi-threading, data level concurrency and arbitrary jumps. Suggestions for further reading will be provided.

Alonzo Church 1903–1995

https://en.wikipedia.org/wiki/Alonzo_Church

2B String rewriting

Let us think about manipulating character strings using rewriting.

Actually this is quite a familiar idea – after all we all spend a lot of time in text editors changing text and that is just rewriting. Of course, we haven't defined a formal rewrite relation: we're relying on the human to use their skill and judgement to knock the text into the desired form.

Let's concentrate on one simple sub-problem, and think about it formally.

Upper-casing of a piece of text involves locating all instances of lower case letters in the range $\boxed{\text{a}}$ to $\boxed{\text{z}}$ and rewriting them into their corresponding upper case character in the range $\boxed{\text{A}}$ to $\boxed{\text{Z}}$.

Rules for upper casing

The uppercasing relation $x \xrightarrow{U_1} y$ for strings of length 1 has these 26 pairs

$$\{(a, A), (b, B), \ldots, (z, Z)\}$$

which we could also write as

$$a \xrightarrow{U_1} A, b \xrightarrow{U_1} B, \ldots, z \xrightarrow{U_1} Z$$

This is a good start, but we want to change all of the characters in a string of arbitrary length, and this rewrite relation only modifies strings of length 1 containing a single lower case character. All other strings are normal forms under this rewrite.

Rule schemas

Now we are faced with a significant problem. We cannot reasonably be expected to write out a separate rewrite rule for every possible case: that would entail writing something of the form $x \xrightarrow{U} y$ for every string containing lower case alphabetic characters. Even if we limited ourselves to strings of length 5, there would be 26^5 such rewrite rules, which is 11,881,376 separate rewrite rules. (Or is it $26^5 - 1$?)

A *rule schema* is a shorthand for a set of rewrite rules. A schema contains *variables* which in this case are placeholders for arbitrary bits of string. We shall use variables α and β to represent substrings of any length, including the empty string which has length zero. This allows us to write out the more useful relation $\xrightarrow{U}$ as follows

$$\alpha a \beta \xrightarrow{U} \alpha A \beta, \quad \alpha b \beta \xrightarrow{U} \alpha B \beta, \quad \ldots, \quad \alpha z \beta \xrightarrow{U} \alpha Z \beta$$

Mathematically, this is just an abbreviated recipe for the *infinite* set of rewrite rules that handle the individual strings.

More subtle rewrites

Consider this rewrite scheme which is superficially similar to the upper-casing schema

$$\alpha a \beta \xrightarrow{U} \beta A \alpha, \quad \alpha b \beta \xrightarrow{U} \beta B \alpha, \quad \ldots, \quad \alpha z \beta \xrightarrow{U} \beta Z \alpha$$

The modification here is that α and β are swapped over on the right hand sides, so the effect of the rewrite is to both to convert lower to upper case letters, and to swap over the ends of the string using the position of the lower case letter as a pivot.

Can you see that actually this is a rather hard rewrite system to reason about? Consider the string $AxyZ$. This has two lower case letters, and so two rewrites will be induced. But depending on the order we perform the rewrites, we get different results:

$$AxyZ \to yZXA \to ZXAY$$

$$AxyZ \to ZYAx \to XZYA$$

Thus we can see that this relation is non-confluent, which we want to avoid!

Avoiding implicit equality tests

This innocuous looking rewrite scheme has a hidden constraint:

$$\alpha a \alpha \xrightarrow{U} \alpha A \alpha$$

The left hand side of the rewrite $\alpha a \alpha$ will only match strings in which the letter a has identical strings on each side, such as $xyaxy$ and $zzazz$. The point is that by using α twice in a pattern we implicitly demand that both instances match the same substring.

In general this extra equality test can create a fixed overhead on real implementations of term rewriting. One way of avoiding the overhead is require each variable in an open expression to be only used once, and that is the approach we shall take in this course.

Avoiding contiguous variables

Here is another rewrite schema that is mathematically perfectly acceptable but which can cause real term rewriting implementations indigestion (and thus should perhaps be avoided):

$$\alpha\beta a \xrightarrow{U} \alpha A\beta$$

Now this is interesting because the left hard expression $\alpha\beta a$ will match any string ending with an a, and for strings of length greater than one there are *multiple ways to do the matching*. Consider the string xya. The concatenation of α and β can match the prefix xy in these three ways, leading to three rewrites (so the rewrite relation is not a function):

α	β	Rewrite
xy	ϵ	xyA
x	y	aAy
ϵ	xy	Axy

Here we have used *epsilon* ϵ to represent the empty string in the variable columns

Matching, binding and substituting

An expression with variables in it is called an *open* expression . A *substutution* is used to 'fill in' the variables. An expression whose variables have all been substituted (that is an expression with no variables in it) is called a *closed* expression. A *pattern* can be either a closed or open expression.

In rewriting, the expression to the left of the rewrite symbol is in a *match* context and the expression to the right is in a s *substitute* context:

$$\text{match-pattern} \rightarrow \text{substitute-pattern}$$

One way to think about this is that we are given some input string which we then attempt to fit the match-pattern to. Literals must match exactly, and pattern variables are then fitted into the remaining substrings to create a set of *bindings* as shown in the table on the previous slide. We then create the rewritten results by taking the substitute-pattern and generating all of the strings that we can make by replacing variables with their bound values.

John Barkley Rosser Sr. 1907–1989

https://en.wikipedia.org/wiki/J._Barkley_Rosser

2C Term rewriting

We are now going to consider how to represent programs inside the computer, leading to the notion of tree-like *terms*.

We shall then extend our notions of constants, variables, patterns and substitutions to terms, and look at one limited form of pattern matching, giving pseudo-code algorithms for matching and substition.

In later sections we shall look at concrete implementations of this form of pattern matching, examining both mutable and immutable representations; the first using the Java API functions and the second using a hand crafted hash-coded representation based on arrays of integers.

Infix syntax

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.

Infix notation does not extent comfortably operations with more than two arguments (although consider the C and Java `p?e1:e2` conditional operator).

Prefix and postfix syntax

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.

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 its evaluation which is 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 fragment

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

might be written

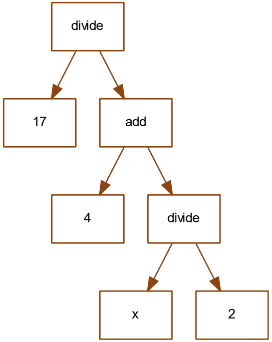
seq(assign(x,2), while(greaterThan(x,5), seq(assign(y, mul(y, y)), assign(x, add(x, 1)))))

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

Prefix notation and trees

Prefix notation has the great merit of uniformity: the wide variety of syntactic styles which are use 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 this tree



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 list of terms is a term.

We are very permissive about what constitutes a symbol. When thinking about computer based tools, the name of a symbol can be any sequence of characters, not just the usual alphanumeric identifiers, numbers and punctuation symbols of conventional programming languages. The only characters we need to be careful of are the parentheses (and) and the comma , because they are used in the definition of terms: we say they are *meta symbols*. If we do want a parenthesis or a comma within a symbol, we usually write it with a preceding back-slash \(. 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 \\.

The recursive nature of terms

Rule one on the previous slide 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.

Just as every subtree of a tree is itself a tree, every subterm of a term is itself a term. We shall exploit this observation in our implementation of *immutable terms*.

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.

Term patterns

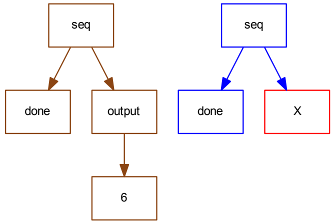
As with string rewriting, a term pattern is a term which may contain term variables. A term which has no term variables in it is called a *closed* term. *Pattern matching* is the process of comparing a closed term to a pattern to decide if they match and if they do, constructing a table of term variables 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*.

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.

Matching of terms

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.



Notation for the matching operation

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 $\perp$, or a set of bindings. So:

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

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

seq(done, output(6)) $\triangleright$ seq(X, done) returns $\perp$ 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 $\perp$ if they differ.

Pseudo-code for term matching

```
match(M: set of term variables, E: environment, t: term, p: term)
        returns environment OR bottom
  if label(p) in M then add p |−> t to E
  else if label(p) != label(t) then return bottom
  else for ct in children(t), cp in children(p) do add match(E, ct, cp) to E
  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.

Substitution of terms

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

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

returns

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

Pseudo-code for term substitution

Here is a recursive function to perform substitution

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

Some simple rewrites

In our implementation, we usually use the $\rightsquigarrow$ symbol for unconditional rewrites. (We shall encounter conditional rewrites and inductively defined rewrites in section 3; we usually use symbols such as *rightarrow* and $\Rightarrow$ for these conditional rewrites.)

We also distinguish between constant symbols and variables by using a leading underscore for variable names.

Following those conventions, and using our prefix-style internal syntax, we can express the while-expansion trick as:

while(_predicate, _body) $\rightsquigarrow$ if(_predicate, seq(_body, while(_predicate, _body)))

Note the use of the seq() binary operator to sequence elements together.

What about our two instruction code, for which we wanted to convert x+y into x-(0-y)? Well in prefix style that rewrite can be expressed as:

add(_x,_y) $\rightsquigarrow$ sub(_x, sub(0, _y))

Haskell Curry 1900–1992

https://en.wikipedia.org/wiki/Haskell_Curry

2D Rewriting strategies

Let's assume that we have a set of rewrite rules. Now, formally, rewrites are defined over the whole of the term to be matched, which is equivalent to saying that rewrites can only be performed at the root term.

When people think about rewriting, though, they often think about local transformations that can be applied deep in the tree.

It turns out that it is possible to write rules which have the effect of, say, finding the deepest instance of an operation and rewriting it into a different one. But using *pure* rewriting, this can get quite messy and the traversal rules can obscure the underlying simplicity of the specification.

We sometimes, therefore, imagine an outer *traversal machine* which is walking the term in some well-defined order, stopping at each node and trying to apply the rules. The traversal machine implements a *rewriting strategy*

Subsetting rules and using phased strategies to manage termination

Practical applications of term rewriting often involve rules which are badly behaved, at least from a mathematical point of view. For instance, we might want to, in different contexts, both multiply-out $a * (b + c)$ to $a * b + a * c$ and factorise $x * y + x * z$ to $x * (y + z)$. However these two operations obviously are in tension with each other: if fact they are effectively inverse with rules

$$\mathsf{mul}(X, \mathsf{add}(Y, Z)) \to \mathsf{add}(\mathsf{mul}(X, Y), \mathsf{mul}(X, Z))$$

$$\mathsf{add}(\mathsf{mul}(X, Y), \mathsf{mul}(X, Z)) \to \mathsf{mul}(X, \mathsf{add}(Y, Z))$$

This is obviously nonterminating. A practical solution is to put these rules into different subsets (sometimes called *modules*) and the use a strategy that applies these rules appropriately in separate phases.

Subsetting rules and using phased strategies to manage confluence

Similar approaches can help with confluence. Recall our earlier string rewriting example:

$$\alpha a \beta \xrightarrow{U} \beta A \alpha, \quad \alpha b \beta \xrightarrow{U} \beta B \alpha, \quad \ldots, \quad \alpha z \beta \xrightarrow{U} \beta Z \alpha$$

When applied to the string $AxyZ$, depending on the order we perform the to rewrites, we get different results:

$$AxyZ \rightarrow yZXA \rightarrow ZXAY$$

$$AxyZ \rightarrow ZYAx \rightarrow XZYA$$

Here again we can effectively specify the order of application of rules by putting them into different subsets, and using them in two separate phases.

Expressions naturally need inside out evaluation

Consider an arithmetic expression such as

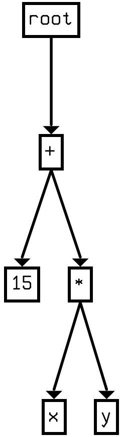
$$15 + x * y$$

Conventionally, the tree for such an expression has the *highest* priority operations at the *lowest* part of the tree, thus we would represent this as

```
root(+(15,*(x,y)))
```

If we wanted to evaluate this, we would first rewrite x and y to the values in those variables, then we would rewrite the $*$ subtree to the product of those values, and then we would rewrite the $+$ subtree to the sum of 15 and the product.

The rewriting needs to proceed outwards from the innermost bracket



Simple Java class for binary term constructors

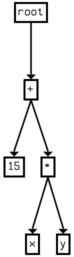
Here is a minimalist Java class for representing terms of fixed arity 2 which we shall use to remind ourselves of the three natural traversal orders

```
class StringTree
{
   private StringTree leftChild;
   private StringTree rightChild;
   private String payload;
   ...
}
```

Preorder traversal

```
void printPreorder()
{
    System.out.println(payload);
    if (leftChild != null) leftChild.printPreorder();
    if (rightChild != null) rightChild.printPreorder();
}
```

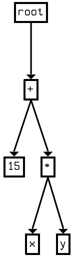
```
root
+
15
*
x
y
```



Inorder traversal

```
void printInorder()
{
    if (leftChild != null) leftChild.printInorder();
    System.out.println(payload);
    if (rightChild != null) rightChild.printInorder();
}
```

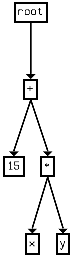
```
15
+
x
*
y
root
```



Postorder traversal

```
void printPostorder()
{
    if (leftChild != null) leftChild.printPostorder();
    if (rightChild != null) rightChild.printPostorder();
    System.out.println(payload);
}
```

```
15
x
y
*
+
root
```



Sibling order

Pre-order is sometimes called *top down* and post-order *bottom up*. I am not keen on these terms as they both demand a traversal from the root to the leaves and back up again, and the top-down/bottom-up labels obscure that reality.

In more detail, the traversals we have described might better be called *left-preorder*, *left-inorder* and *left-postorder* because they visit the childen in left to right order.

We could just as easily have visited the children in right-to-left order, leading to *right-preorder* and so on.

The pre- and post- definitions extend naturally to arities greater than two since they simply require an action as we recurse *into* a node (preorder) or immediately before exiting (postorder).
Inorder traversals for arities other than two require more care. Should an arity one node be printed out at all? The usual convention is that we get $k - 1$ printouts for k children, which is useful for, say, comma delimited lists.

Incomplete traversals

For expressions, a postorder application of arithmetic evaluation rewrites is convenient and will reduce an entire expression in one phase.

However, for some applications, we can abort a phase as soon as some condition is met. For instance, if we want to test for the existence of a certain node, we can use a *one shot* strategy which returns as soon as locate an instance of that node.

A more subtle example might abort if no applicable rewrites are found amongst a set of sibling nodes, or more plausibly continue running a phase until that condition arises.

Real systems

The engineering of term rewriting tools that allow complex strategy definition is a current research area. The *Stratego* system implements a so-called calculus of rewriting strategies, where strategy primitives may be combined to produce very complex control flows. You can read about Stratego at
`http://www.metaborg.org/en/latest/source/langdev/meta/lang/stratego`

An alternative approach is discussed in this paper:
`https://dl.acm.org/doi/10.1145/941566.941568`

An encouraging formal basis for embedding strategies in pure rewriting is described at
`https://arxiv.org/abs/1705.08632`
(Note that you may need to be on campus, or to have tunneled in with the College's VPN to access some of these sources.)

Mosses Schonfinkel 1888–1942

https://en.wikipedia.org/wiki/Moses_Schonfinkel

2E Implementing terms

Naïve implementations of rewriting systems can be very slow. Good implementations of rewriting systems are usually just plain slow. So rewriting based interpreters are usually for *prototyping*. However, on a modern machine, sometimes a rewrite interpreter is fast enough for basic scripting languages.

A subsidiary problem is that rewrite interpretations of programs are *very* wasteful of memory. Unlike a conventional implementation, in which the static program is traversed via a program counter, a rewrite interpreter continuously makes new trees, and of course each tree is itself a rather inefficient representation of a program compared to the tight binary encodings used for conventional programs.

On the other hand, modern computers provide gargantuan memories compared to those of, say, the 1970s, and the formal advantages of the rewriting approach in terms of correctness-by-construction and conciseness are not to be lightly dismissed.

This section is about building efficient infrastructure implementations for rewriting systems that we shall use in later sections to build language interpreters.

Naïve term representation

From a user's perspective, a term is simply a tree in which the nodes are labeled with a piece of text. Most terms have arity less than three, and by using scaffolding nodes labeled **andAlso** we can handle higher arities.

A straightforward implementation of terms, then, might involve a binary tree class such as

```
class Term
{
  private Term leftChild;
  private Term rightChild;
  private String symbolString;

   void printPreorder() {
       System.out.println(symbolString);
       if (leftChild != null) leftChild.printPreorder();
       if (rightChild != null) rightChild.printPreorder();
   }
}
```

Better term representation

Limiting the user to a binary tree is workable, but represents an uncomfortable constraint, and it clutters the internal syntax as well as requiring extra rewriting steps. There is a nice trick which allows us to use what are effectively binary tree nodes internally but which present as variable arity trees externally.

```
class Term
{
  private Term child;
  private Term sibling;
  private String symbolString;

   void printPreorder() {
        System.out.println(symbolString);
        if (child != null) child.printPreorder();
        if (sibling != null) sibling.printPreorder();
    }
}
```

The immense cost of term rewriting

A term rewriting interpreter, such as the eSOS system that we shall study in Section 3, constructs vast numbers of term trees during the execution of even small programs.

In a staightforward implementation this involves the creation of vast numbers of small tree nodes, and potentially a lot of work for the garbage collector.

However we see a lot of repetition in the trees. Firstly, most of the tree is made up of constructors which are drawn from the set of internal syntax elements, which will be very small compared to the number of term nodes. So, we should map the set of tree labels onto the naturals and label term nodes with a single integer rather than a reference to a string in memory which is represented over and over again. We can perform this mapping by building a `HashMap<String, Integer>` to map from strings to naturals and a `HashMap<Integer, String>` to perform the reverse mapping.

DAGs and immutable terms

We also see the same subtrees appearing over and over again. Every time an **if** statement is evaluated we might expect to see a terms like `if(true, _2, _3)` and `if(true, _2, _3)` appearing and then being rewritten to the substitution for either _2 or _3. Similarly, we may find subtrees such as `__int32(1)` and `__int32(0)` appearing over and over again, sometimes within the same large tree.

A key idea, then, is to *cache* terms and to represent the forest of trees that we are working on as a forest of Directed Acyclic Graphs (DAGs) with subtrees being shared wherever possible.

A necessary corollary is that shared terms must be *immutable*. If we have one internal representation for two instances of a topologically identical subtree, then any changes to the internal representation would be expressed in both instances.

Hash tables of terms: the term is its hash table address

If we simply store every subtree we ever see, and link to it if it is required again, then we get a much more compact representation. We also get very fast term equality testing.

The problem is that we need to be able to quickly find an old subtree, or decide that it has not been cached and so create a new one.

A good way to do this is to use a hash table; each unique subtree (term) then is mapped onto a natural number by a hash function which is then used as an address into a table of terms.

How much memory does each unique subtree take in this case? Well the label of the root of the subtree is one natural number, and then we need one natural number for each child.

It is very often the case that our terms are fixed arity: for instance internal syntax node `ifThenElse(_P, _T, _F)` will always have arity three. In other cases, such as collection data types (set, map and so on) the arity is variable.

For fixed arity nodes, we need a map from node labels to aritiesm and then $1 + \textit{arity}$ naturals per node. For variable arity nodes, we need to store the arity in the node so we need $2 + \textit{arity}$ naturals.

Aspects of ART's implementation

ART includes a class `ITerms` which implements immutable terms using hash maps. `ITerms` is an abstract class which defines all of the user accessible public methods and maintains a separate hash table for strings. `ITerms` also implements various pattern matching and substitution operations, as well as containing subclasses that implement the value system that we shall discuss in the next section.

The intention is that an ITerms application (such as the eSOS system) can be built using Java or C in one of two modes that I call API and Pool; and Pool can be used for direct hardware implementations.

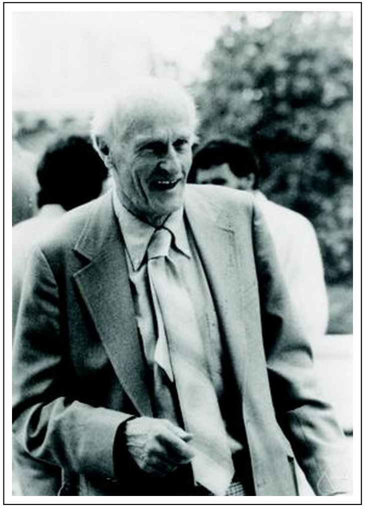
In API style, the host language's hash table implementation is used, with each unique term being represented by an object. This is easy to engineer but slow.

In Pool style (almost) no objects are used – instead blocks of memory are allocated as large arrays of `int` and the terms are represented as contiguous subsequences within those arrays. Allocation is fast because there is no garbage collection so we just increment the free boundary at each allocation.

The CWI ATerm datatype

The ideas underpinning ART's term rewriter originate with the ASF+SDF system built by Mark van den Brand and others under Paul Klint's leadership at CWI Amsterdam in the 1980s and 1990s (`https://dl.acm.org/doi/10.1145/567097.567099`)

The core datastructure is the *annotated term* which has a low level type system built in, and is designed for information interchange between applications with maximal term sharing . It was used for communications between applications (much as XML is used today) and for efficient term rewriting in ASF+SDF and Stratego (`https://research.tue.nl/en/publications/efficient-annotated-terms`)



Stephen Kleene 1909–1994

https://en.wikipedia.org/wiki/Stephen_Cole_Kleene

2F Types and operations: the ARTValue system

Programs compute *values* using control-flow compositions over builtin *operations*.

At machine level, values are simply bit strings with no more semantics than that. If the program counter is addressing a bit string, then that bit-string will be treated as an instruction and brought into the processor for execution.

On nearly all commercially available computers[3], the operands for the instruction are themselves just bit strings with no type information at all. The instruction itself contains implicit type information: for instance there will be separate addition instructions for integer and floating point operands. If we accidentally ask a floating point add instructon to fetch its operands from the wrong part of memory, it will cheerfully generate a new floating point number anyway.

[3]There were experiments in the 1970s into so-called *capabilities* architectures which tagged memory with datatyping information. Recently, security concerns have brought some of these ideas back into the research agenda. (`https://en.wikipedia.org/wiki/Capability-based_addressing`)

The rise of typing in programming languages

The earliest compilers (such as the original FORTRAN system) projected the underlying hardware constraints into the high-level program. For instance the ICT 1500 FORTRAN compiler that I learnt to program with disallowed so-called mixed-mode arithmetic, that is an operation that involved an integer and a floating point operand. In Java the integer would be *coerced* to a float or double when necessary before performing the operation.

The language Algol-60 (which is the mother-lode for most modern languages) introduced the idea of block structure and our now familiar control flow statements.

In the 1960s languages began to be equipped with *user defined types* constructed from, primitive types and a variety of aggregation mechanisms such as Pascal's tt record and C's `struct`. These languages usually also provided heap based allocation allowing dynamic data structures to be created at run time.

A readable and scholarly survey from 1972 by Tony Hoare is available here: `http://www.informatik.uni-bremen.de/agbkb/lehre/programmiersprachen/artikel/hoare-data-structuring.pdf`

Typeful programming

Equipping data with user-specifiable type information significantly increases programmer productivity since it enables nonsensical operations to be detected. A strongly typed system will not allow, for instance, a character value to be divided by a real number.

More significantly, in some systems we can use typing and subrange typing to generate frameworks of correctness checking. For instance, in a 3D CAD system we might want to catch programmer errors such as adding a length to an area. Most likely these would both be fundamentally implemented as real numbers, but they represent physically heterogeneous and incompatible quantities.

There is a school of thought, perhaps typified by some Haskell programmers, that believes that system design should proceed from a type analysis of the problem, through the definition of necessary function signatures and then finally coding of function bodies. This approach, which is rather algebraic in nature, comes naturally to some folk and feels alien to others.

Coercion vs conversion

Implicit coercion can be a programmer convenience, but can cause unpleasant bugs. The alternative is to require explicit conversions using casts or conversion functions such as `toString`, `parseInt` and collection constructors.

Here is the classic coercion nasty surprise which has caught me more than once:

```
System.out.println(1/3 + 2.1);
```

Discouragingly, this prints out 2.1 because the division has higher priority and is performed as integer division before the result (zero) is coerced to a double and added to 2.1.

One of the most complete implicit conversion regimes is to be found in LISP family languages, in particular Scheme, whose *numeric tower* attempts to replicate the conventional mathematical hierarchy of numbers, in both exact and rounding styles.

`https://groups.csail.mit.edu/mac/ftpdir/scheme-7.4/doc-html/scheme_5.html`

Strong and weak typing

A strongly typed language will never allow a nonsensical operation. A weakly typed language will allow nonsensical operations, which may cause the program to fail and may also be a security risk.

For instance, in C a `void*` pointer has no type, and can be cast to a pointer to any type (even code in a `gcc` compiled program!) To make matters worse, C also allows pointer arithmetic, so that an address can be offset.

Strong type checking must include collection extents. Even simple array updates can be sufficient to wreck a program, or wreak havoc with system security if bounds are not checked. *Buffer overrun* exploits are a well known source of security failures.
`https://en.wikipedia.org/wiki/Morris_worm`

The Rust programming language
`https://en.wikipedia.org/wiki/Rust_(programming_language)` probably represents the current state of the art in strongly typed safe languages.

Static and dynamic typing

There is sometimes confusion between the strong/weak typing axis and the static/dynamic axis. In reality they are completely orthogonal: strongly typed dynamic languages like Python exist, as do weakly typed static languages (such as any assembler).

Fully dynamic typing requires all values to carry their type with them, that is a value at run time is an ordered pair (type, data). This naturally incurs an overhead, both in space and in runtime performance since type checks must be executed interleaved with the operations specified by the program.

As a general rule, the more dynamic a language is, the more flexible it is from a user perspective, but probably the lower its performance.

In a fully static language, all checks are performed at compile time. This means that any data-dependent type properties, such deciding which form of addition to perform, must be independent of the runtime data.

In reality, most languages sit somewhere on spectrum. Consider array bounds, for instance. In Pascal, they are checked at compile time which means that arrays have a fixed size. In Java arrays are allocated at run time which means that bounds checks must be performed dynamically. However the base type is fixed at compile time.

Name equivalence vs structural equivalence

How should type correctness be defined? What does it mean for two types to have equal properties? In languages in the Pascal tradition, the name of the type is all that is used to decide equivalence – so called *name equivalence*. This is useful because Pascal supports *subrange* types which define an interval over the integers and sometimes one wishes to have two type-incompatible subranges which actually have the same bounds. In additon it allows us to distinguish between an ordered pair which is a cartesian coordinate and an ordered pair which represents, say, the start and end times of a lecture.

Structural equivalence allows types with the same 'shape' to be equivalent, so our two kinds of ordered pair would be valid in either context.

Duck typing extends structural equivalence to dynamic object oriented programs in that a object is type compatible as long as it supports method calls which are actually made to it in a particular run of a program. This implies that type compatibility is a function of the particular data inputs. This is very flexible, but perhaps dangerous. Reflection allows Java to be technically duck typed though the language does not encourage that practice.

Name equivalence, statically typed languages tend to be safe but tiresomely rigid requiring special case code to be written for mixed type computations. Duck typing is flexible leading to compact code which may induce nasty surprises.

ART's types

ART provides a fixed set of atomic (in the sense of having no internal structure) types and a fixed set of collection types from which richer structures may be constructed.

Primitive types

```
__bottom, // failure
__done, // terminal for program terms
__empty, // uninitialised data
__boolean, // boolean type
__char, // character type
__intAP, __int32, // integer types - AP means arbitrary precision (BigNum)
__realAP, __real64, // real types
```

Collection types

```
__list, __tuple, __array, __string, __map, __mapOrdered, __set
```

ART's operations

ART provides a fixed set of operations, any of which may be implemented into a type

comparisons __gt, __lt, __ge, __le, __eq, __ne
logic and bitwise __not, __and, __or, __xor, __cnd
shift and rotate __lsh, __rsh, __ash, __rol, __rolc, __ror, __rorc
arithmetic __neg, __add, __sub, __mul, __div, __mod, __exp
collection size (returns 1 for atomic types) __cardinality
lists __prepend, __append, __head, __tail
sets and maps
__insert, __delete, __update, __contains, __union, __intersection, __difference
network and disk resources __open, __input, __output, __flush, __close
user backend hook for Domain Specific Languages __user

Dynamic type checking of ART values

The ART value system uses a simple form of dynamic type checking. We may wish to model dynamic languages, and so having a fully dynamic type systems is essentially the only option. We can, however, add rules to our language specification that cause types to be checked before program execution: the so-called *static semantics* of the language. If our type system was static, then it is very hard to see how we could comfortably model all aspects of dynamic type checking.

The implementation uses a simple inheritance trick to handle the dynamic type checking. When you call a builtin function such as `__add(10.4, 3.0)` ART looks at the type of the left operand (10.4) and attempts to call method `__add()` in type class `__real64`. All type classes are subclasses of `Value`, and `Value` has a method in it for every operation. The methods in `Value` uniformly issue the error message

```
value type typename does not provide operation operation
```

Hence, an attempt to, say perform a logical `__and` operation on a `__string` will generate an interpretation-time error.

This yields an extremely compact yet flexible value system. However operations and types can only be added by extending and recompiling the `Value` classes.

The V3 and V4 ART value systems

There have been two versions the value system called, perhaps confusingly, V3 and V4. (The value system and term rewriting were only added to ART in its third version.)

In V3, variable names are inferred, built in operations have names such as `addOp(x,y)`, type predicates have names such as `isInt(x)` and operations may only appear on the right hand side of a $\triangleright$ operator. The operations implement automatic coercions via a Scheme-like numeric tower; hence there are supertypes such as `number` available.

In V4, the value system was simplified and flattened, so there are no implicit coercions at all. The rationale is that, as with the dynamic vs static typing problem, a system that automatically coerces cannot easily model a system that does not. Since ART is used to model languages, we need a very plain value system with no 'cleverness' – that should instead be implemented in the eSOS rules.

Operations now have names like `__add`, variable names must have a leading underscore and operations can appear in any substitution context. Individual values are represented as two level subtrees: `__int32(24)` and so types can be checked directly with clauses such as `_X |> __int32(_)`, sp we don't need operations such as `isInt`

Atomic types

Atomic types in programming languages begin with various sizes of integers and floating point numbers, whose behaviour is a direct expression of the underlying machine code instructions.

Whilst integer arithmetic is usually compatible across architectures up to word length, real number arithmetic has only been fully standardised in the 1980s, and it is still the case that some machines have multiple floating point modes.

Character sets have been another fertile source of incomatibility between architectures. Early machines often used six-bit character sets which were compact, but often had no provision for lower case letters. The 7-bit ASCII code became very widely distributed during the microcmputer boom of the late 1970s and 1980s, but many important glyphs including currency signs for sterling (and the Euro, which had not been invented when ASCII was defined) were missing. Java and later systems use Unicode which allows for an essentially unbounded set of glyphs.

Enumeration types

An *enumeration* is a mapping from a set of symbols onto the naturals.

The idea was popularised by Pascal, and subseqently added to C where previously the preprocessor was used to define symbolic constants.

A weakness of enumerations in most statically typed languages is that the alphanumeric form of the enumeration element is discarded, and so when printed an enumeration value only appears as the associated natural number. In interpreters, the symbol table is available at runtime, so the alphanumeric values can be retrieved.

Java introduced `enums` as (almost) first class objects, with associated `toString()` and `value()` methods that allowed programs to interrogate the enumeration as if it were a map. There are still limitations though, and if you do not statically know all the members of an `enum` then using a class which mimics `enum` functionality with a `Map` might be useful. However one then cannot exploit `switch` statements since their labels must evaluate to compile time constants.

A closely related idea is the definition of individual constant integers which then become, effectively, enumerations with only one element. Again, Java allowed a more *dynamic* flavour of constants with its `final` keyword: in Java a constant is a write-once variable.

Subrange types

An oddity of the original BCPL and C languages which has been inherited by later designs such as Java is that array indices are always based at zero. Now, an array is effectively an implementation of a partial function from an interval of the naturals to some domain.

Why should that interval always be based at zero? Well, it reflects the underlying hardware implementation in which array index expressions represent the *offset* from some array base address. However, high level lamnguages are intended to be, well, high level and hiding hardware quirks is an important part of that principle.

Pascal family languages provide declarations of the form

```
myArray:  ARRAY[13..25] of int;
```

which allow both the lower and the upper bound to be specified. The compiler can easily generate code for accesses to `myArray` by subtracting $13\times$ sizeof(int) from the index expression before using it as an offset. Subranges can also be used as first class types.

Dimensions

Physicists and engineers are taught to perform *dimensional analysis* on formulae to ensure that results come from combining values of the right kind. For instance an area is measured in L^2 where L is the local unit of length. It is an error to add an area (with kind L^2) to a volume with kind L^3. Similarly, time, cost and length are all one dimensional measures, but adding a time value to a sterling price cannot make sense.

Expressing these constraints using conventional type systems can be rather clumsy, and there have been many attempts over the years to add a natural syntax to programming languages to handle this notion, notably in Ada and the Boost libraries for C++. Probably the most mature example today is the F# 'units of measure' feature

`https://docs.microsoft.com/en-us/dotnet/fsharp/language-reference/units-of-measure`

developed by Andrew Kennedy (now at Facebook), who talks about their applications in this video:

`https://channel9.msdn.com/Blogs/Charles/Andrew-Kennedy-F-Units-of-Measure`.

Adding non-general but application specific units of measure is a very useful feature in Domain Specific Languages, where Volume and Area could be resrved type keywords, with their own resolution algorithm.

Polymorphism through overloading and named arguments

In a statically typed language, verbose code may be needed to handle type-rich programs. In a language such as C, the signature of a function includes the name, and the type of the arguments and must be specified at compile time.

In C++ and Java, the signature of a function may be *overloaded* so that a family of functions with the same name but different parameter lists may be created.

This facility is particularly useful for constructors, where we want to have default behaviour for many parameters, but also allow the user to have access to the *full fat* constructor in which initial values for all public fields may be specified concisely.

For complex classes, the number of constructors can become very large (in detail 2^k for k arguments) and in any case the usual implementation involves an extra call. *Named arguments* allow a function call to specify some subset of the functions parameters, with defaults being used for unnamed arguments. Python and OpenSCAD provides this feature.

Polymorphism through flexible arities

Often, especially with collection data types, we should like to pass an arbitrary sequence of values to a function. C introduced *variadic* functions whose paramater list has one or more conventional typed parameters followed by the notation `...` to indicate an open sequence of arguments.

The implementation in C is ugly and unsafe: essentially the user is able to directly walk the functions data which is stored on the stack in a *stack frame*. By continuing to walk the stack, functions can gain access to the data for enclsoing functions.

The Java implementation is both conceptually simpler and safer. The arguments corresponding to the `...` section are loaded into an array which is then delivered to the function as a single argument.

A more radical approach is taken by ML, in which formally *all* functions take one argument which is a tuple. This provides a very clean solution.

Polymorphsim through inheritance

In dynamic languages, values carry their types, and functions can branch dependent on the actual type supplied at runtime. This can allow a large set of signatures to be implemented with one piece of code.

A half way house is to allow types to have supertypes, just as integers have supertype number in the Scheme numerical tower, but to allow the use to specify these relationships. Object oriented languages naturally provide this sort of polymorphism.

In the earliest OO languages such as Smalltalk, a single inheritance hierarchy is created. C++ allows multiple inheritance, but that brings with it the risk of type ambiguity - the so-called *diamond of death*. Java interfaces were a technical response to this limitation.

Polymorphism through parameterised types

A more difficult kind of polymorphism arises in the context of *collection* data types such as sets, maps and lists. We should like to have a single container type which can adapt to whatever kinds of values are put into it. In an interpreted environment with dynamic typing where all values carry their types with them this is quite easy. In a statically typed language, much more machinery is required.

The main solution is to allow some form of *type variable*, that is an argument to a type or even a first class variable (as in ML) capable of recording a type.

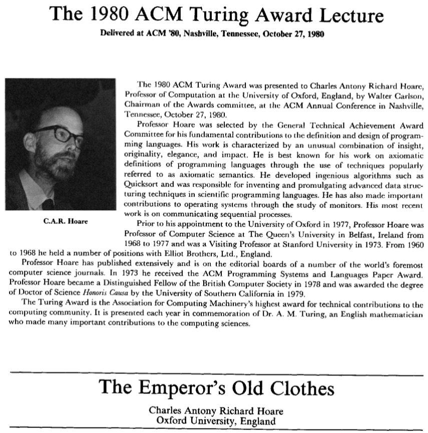
Java originally had no such feature, but type parameterisation via the $\langle$ and $\rangle$ brackets was introduced at version 1.5. The implementation is constrained, though, by the JVM which has no simple facility for type representation at run time (so-called *type erasure*) and as a consequence, you cannot declare an array of generic types in Java.

Andrew Kennedy at Microsoft designed the equivalent facility in C#, and that is well behaved.

Professor Sir Tony Hoare 1934–

https://en.wikipedia.org/wiki/Tony_Hoare

The emporer's old clothes: https://dl.acm.org/doi/pdf/10.1145/358549.358561



The 1980 ACM Turing Award Lecture

Delivered at ACM '80, Nashville, Tennessee, October 27, 1980

C.A.R. Hoare

The 1980 ACM Turing Award was presented to Charles Antony Richard Hoare, Professor of Computation at the University of Oxford, England, by Walter Carlson, Chairman of the Awards committee, at the ACM Annual Conference in Nashville, Tennessee, October 27, 1980.

Professor Hoare was selected by the General Technical Achievement Award Committee for his fundamental contributions to the definition and design of programming languages. His work is characterized by an unusual combination of insight, originality, elegance, and impact. He is best known for his work on axiomatic definitions of programming languages through the use of techniques popularly referred to as axiomatic semantics. He developed ingenious algorithms such as Quicksort and was responsible for inventing and promulgating advanced data structuring techniques in scientific programming languages. He has also made important contributions to operating systems through the study of monitors. His most recent work is on communicating sequential processes.

Prior to his appointment to the University of Oxford in 1977, Professor Hoare was Professor of Computer Science at The Queen's University in Belfast, Ireland from 1968 to 1977 and was a Visiting Professor at Stanford University in 1973. From 1960 to 1968 he held a number of positions with Elliot Brothers, Ltd., England.

Professor Hoare has published extensively and is on the editorial boards of a number of the world's foremost computer science journals. In 1973 he received the ACM Programming Systems and Languages Paper Award. Professor Hoare became a Distinguished Fellow of the British Computer Society in 1978 and was awarded the degree of Doctor of Science *Honoris Causa* by the University of Southern California in 1979.

The Turing Award is the Association for Computing Machinery's highest award for technical contributions to the computing community. It is presented each year in commemoration of Dr. A. M. Turing, an English mathematician who made many important contributions to the computing sciences.

The Emperor's Old Clothes

Charles Antony Richard Hoare
Oxford University, England

3 Structural Operational Semantics

SOS is a formal game that models a computer's operation as *conditional* term rewriting

3A Conditional term rewriting

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 general effects that we used to evaluate GCD at the end of the first section.

There is one major gap though, and that is the *selection* of sub-phrases to rewrite.

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.

In this section we are going to learn how to control the application of rewrites so as to mimic the sequential execution of a program.

Configurations

When modeling 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 **t**erm θ, and possibly including a **s**tore term σ, an envi**r**onment ρ, an out**p**ut stream ϕ, a **d**ata input stream δ and some sig**n**als ν.

Notice how the Greek letters for each semantic entity have some mnemonic relationship to their rôle.

Note that in this rewriting world *everything* is a term, so the tupling operation itself is a term tuple(,). To avoid clutter, we often write tuple(x,y) using angle brackets as $\langle x, y\rangle$ but that is just a syntactic convention.

In our first examples we shall use configurations of the limited form $\langle\theta, \phi\rangle$ comprising a program term θ and an output term ϕ. We shall refer to the *current configuration* as $\langle\Theta, \Phi\rangle$ (using the capitalised versions of those greek letters).

Semantics by enumeration

For very simple languages, it might be practical to define their semantics by enumeration. Consider a language **add02** 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 $\langle\theta,\phi\rangle$ there are only nine possible programs that can be written in this language each of which we could evaluate directly using these nine rules:

$$\langle \text{output(add(0, 0))}, [\,]\rangle \rightarrow \langle \text{done}, [0]\rangle$$
$$\langle \text{output(add(0, 1))}, [\,]\rangle \rightarrow \langle \text{done}, [1]\rangle$$
$$\langle \text{output(add(0, 2))}, [\,]\rangle \rightarrow \langle \text{done}, [2]\rangle$$
$$\langle \text{output(add(1, 0))}, [\,]\rangle \rightarrow \langle \text{done}, [1]\rangle$$
$$\langle \text{output(add(1, 1))}, [\,]\rangle \rightarrow \langle \text{done}, [2]\rangle$$
$$\langle \text{output(add(1, 2))}, [\,]\rangle \rightarrow \langle \text{done}, [3]\rangle$$
$$\langle \text{output(add(2, 0))}, [\,]\rangle \rightarrow \langle \text{done}, [2]\rangle$$
$$\langle \text{output(add(2, 1))}, [\,]\rangle \rightarrow \langle \text{done}, [3]\rangle$$
$$\langle \text{output(add(2, 2))}, [\,]\rangle \rightarrow \langle \text{done}, [4]\rangle$$

A schema for add02

This is, of course, a tiny language and we are effectively trying to define a lookup table which captures the languages complete semantics.

Enumeration is clearly not a very practical approach. What we need to do is to be able to express the pattern of additions more concisely. Recall that 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, z and p be term variables
if the current configuration matches output(add(x,y)) with current output p 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
    return done with output set to p followed by z
else return fail
```

Conditional rewriting expressed using operators

More concisely, using the notations we have developed (and assuming the existence if a function *__is012*(_) that returns a boolean, we might say

$$
\begin{array}{c}
\text{if} \quad \rho_1 = (\langle \Theta, \Phi \rangle \triangleright \langle \text{output}(\text{plus}(X\,, Y\,)\,)\,, \phi \rangle) \\
\text{and} \quad \_\_is012(X) \triangleright \text{True} \\
\text{and} \quad \_\_is012(Y) \triangleright \text{True} \\
\text{and} \quad \rho_2 = ((\_\_add(X\,,Y) \triangleleft \rho_1) \triangleright Z) \\
\text{then} \quad \langle \_\_done, \ \_\_append(\phi, \ Z) \rangle \quad \text{else} \quad \perp
\end{array}
$$

Here we are not just substituting whenever we see a match as in Section 2. In this example we are matching, then testing some conditions, and only substituting if those condition tests succeed. This is *conditional* rewriting.

Built in functions

We have also introduced an important new mechanism here: *built in* functions that take terms and return terms. In ART, these functions have names prefaced by two underscores.

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 we must believe that there is a pure rewriting definition for each function.

If we want to use ART to try out our rules, then we can use any of the operations and types discussed at the end of Section 2. That naturally means that we must also follow the typing conventions and constraints described there, so that (for instance) integer 100 is represented by the term *__int32*(100), but the ART front end can automatically generate those forms from simple constants, so we can (for instance) write `__add(3,4)` and it will generate the term *__add*(*__int32*(3), *__int32*(4))

Inference rules

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:

$$\frac{C_1 \quad C_2 \quad \dots \quad C_n}{\langle \theta, \phi \rangle \to \langle \theta', \phi' \rangle}$$

The elements above the line (the C_i) are called *conditions* (or sometimes *premises*). 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* $\langle \theta, \phi \rangle$*,*
> *and* C_1 *succeeds and* C_2 *succeeds and* $\dots$ *and* C_n *succeeds*
> *then transition to new configuration* $\langle \theta', \phi' \rangle$

Round the clock and recursion

In SOS, one reads these inference rules 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'.

Conditions may also be simple matches against the return value of a function, in which case they are called *side conditions*.

Most importantly, conditions can themselves be transitions, and this implies that the matching process can recursively descends into the subterms of the top level term whilst we check to see if a transition is allowed.

A function call can appear in any substitution context, that is on the left of a $\triangleleft$ operator, and this can reduce the number of variables and side conditions in a rule

Inference rules in detail

The detailed inference rule representation of our schema is

$$\frac{(\_\_is012(X) \lhd \rho_1) \rhd \text{True} \quad (\_\_is012(Y) \lhd \rho_1) \rhd \text{True} \quad \rho_2 = ((\_\_add(X, Y) \lhd \rho_1) \rhd Z)}{\rho_1 = (\langle \Theta, \Phi \rangle \rhd \langle \text{output}(\text{plus}(X, Y)), \phi \rangle) \to \langle \_\_done, \_\_append(\phi, Z) \rangle \lhd (\rho_2 \cup \rho_1)}$$

We have been careful here to represent all of the pattern matching and substitution operations explicitly.

The variables in the above rule are Θ, Φ, ϕ, X, Y and Z. We develop two sets of bindings ρ_1 and ρ_2, and at bottom right we substitute both into the right hand side of the conclusion.

Note that each rule has its own set of term variables even if the same term variable name is used in multiple rules (that is, the *scope* of a variable is its enclosing rule)

Abbreviated rules

It turns out that we only really need a single local environment ρ which has bindings added into it as we perform pattern matches.

This works because we are not allowed to bind variables more than once in a rule. Thus we can also be sure that the first time (in the round-the-clock sense) that we meet a term variable it is being used in a match context to create a binding; and that any subsequent appearances of a term variable must be substitution contexts. The ART tool checks that variables meet this constraint and will issue an error message if you breach it.

We also know that the left hand side of the conclusion is *always* matched against the current configuration $\langle \Theta, \Phi \rangle$. This allows us to abbreviate our inference rule to:

$$\frac{\_\_is012(\mathrm{X}) \triangleright \mathrm{True} \quad \_\_is012(\mathrm{Y}) \triangleright \mathrm{True} \quad \_\_add(\mathrm{X}\ ,\mathrm{Y}) \triangleright Z}{\langle \mathrm{output}(\mathrm{plus}(X\,,Y\,)\,)\,, \phi \rangle \rightarrow \langle \_\_done, \_\_append(\phi, Z\,) \rangle}$$

and this is the style that we shall use in future

Examples in ART

We shall now look at a sequence of examples using the ART tool, building in complexity.

At each stage we shall show the ART-style elided input along with the fully fleshed out and typeset rules, along with some example rewrite sequences.

The ART front end allows us to write constants like this `True, False, 123, 123.4` which will be expanded into an appropriate typed term. You can also write { `a=b` } for a map, `[x,y]` for a list and so on. Finally, you can elide references to configuration elements that are unchanged across a rule

The front end generates a file called `ARTSpecification.tex` each time it runs, and this can be processed by LaTeX to produce the expanded typeset version of the rules

Most of the following examples can be found in the `slelabs/esos` directory

Here is a rule (not a rule schema!) that will increment 3 to 4, but nothing else...

```
---
increment(3) -> 4
```

$$\frac{}{\langle \text{increment}(\_\_int32(3))\rangle \rightarrow \langle \_\_int32(4)\rangle} \quad [\text{R1}]$$

```
** !try 3 <increment(__int32(3))>
Step 1
  Rewrite call 1 <increment(__int32(3))>, ->
  [R1]   --- <increment(__int32(3))> -> <__int32(4)>
  [R1] rewrites to <__int32(4)>
Normal termination on <__int32(4)> after 1 step and 1 call to rewrite()
```

Now let's look at the same rule, but try (and fail) to increment 4

```
---
increment(3) -> 4
```

$$\frac{}{\langle \text{increment}(\_\_\mathit{int32}(3))\rangle \to \langle \_\_\mathit{int32}(4)\rangle} \quad [\text{R1}]$$

```
** !try 3 <increment(__int32(4))>
Step 1
  Rewrite call 1 <increment(__int32(4))>, ->
  [R1]   --- <increment(__int32(3))> -> <__int32(4)>
  [R1] Theta match failed: seek another rule
Stuck on <increment(__int32(4))> after 1 step and 1 call to rewrite()
```

We can generalise the rule to a *rule schema* by using *variables* (prefixed by _ in the ART source) along with the builtin function __*add*(_,_) and thus increment any integer

```
---
increment(_X) -> __add(_X,1)
```

$$\frac{}{\langle \text{increment}(X)\rangle \to \langle \_\_\mathit{add}(X, \_\_\mathit{int32}(1))\rangle} \quad [\text{R1}]$$

```
** !try 3 <increment(__int32(4))>
Step 1
  Rewrite call 1 <increment(__int32(4))>, ->
  [R1]   --- <increment(_X)> -> <__add(_X, __int32(1))>
  [R1] rewrites to <__int32(5)>
Normal termination on <__int32(5)> after 1 step and 1 call to rewrite()
```

However we must work harder if we want to handle nuumbers other than integers, because no automatic coercions are applied.

```
---
increment(_X) -> __add(_X,1)
```

$$\frac{}{\langle \text{increment}(X)\rangle \rightarrow \langle \_\_add(X, \_\_int32(1))\rangle}\quad [R1]$$

```
** !try 3 <increment(__real64(5.5))>
Step 1
  Rewrite call 1 <increment(__real64(5.5))>, ->
  [R1]   --- <increment(_X)> -> <__add(_X, __int32(1))>
!! Function error: __add(__real64,__int32) - operands must be of same type
  [R1] rewrites to <__empty>
Normal termination on <__empty> after 1 step and 1 call to rewrite()
```

We can suppress the fatal library error with a side condition to check the type of the operand by matching against the term $\_\_int32(\_)$

```
_X |> __int32(_)
---
increment(_X) -> __add(_X, 1)
```

$$\frac{X \triangleright \_\_int32(\_)}{\langle \text{increment}(X)\rangle \rightarrow \langle \_\_add(X, \_\_int32(1))\rangle} \quad [\text{R1}]$$

```
** !try 3 <increment(__real64(5.5))>
Step 1
  Rewrite call 1 <increment(__real64(5.5))>, ->
  [R1] _X |> __int32(_)  --- <increment(_X)> -> <__add(_X, __int32(1))>
Stuck on <increment(__real64(5.5))> after 1 step and 1 call to rewrite()
```

Now the computation sticks rather than generating an $\_\_empty$ result (which is better!)

We add an extra rule to handle real number increments

```
-integerInc  _X |> __int32(_) --- increment(_X) -> __add(_X, 1)
-realInc    _X |> __real64(_) --- increment(_X) -> __add(_X, 1.0)
```

$$\frac{X \triangleright \_\_int32(\_)}{\langle \text{increment}(X)\rangle \rightarrow \langle \_\_add(X, \_\_int32(1))\rangle} \quad [\text{integerInc}]$$

$$\frac{X \triangleright \_\_real64(\_)}{\langle \text{increment}(X)\rangle \rightarrow \langle \_\_add(X, \_\_real64(1.0))\rangle} \quad [\text{realInc}]$$

```
** !try 3 <increment(__real64(5.5))>
Step 1
  Rewrite call 1 <increment(__real64(5.5))>, ->
  [integerInc] _X |> __int32(_)  --- <increment(_X)> -> <__add(_X, __int32(1))>
  [realInc] _X |> __real64(_)  --- <increment(_X)> -> <__add(_X, __real64(1.0))>
  [realInc] rewrites to <__real64(6.5)>
Normal termination on <__real64(6.5)> after 1 step and 1 call to rewrite()
```

ART's eSOS trace levels

eSOS rewrites are initiated in ART with the `!try` directive.

`!try` *`optional-level input-term transition optional-test-term`*

`optional-level` is an integer that specifies the amount of tracing information supplied: each level includes all output from its lower levels. If omitted, then level 3 is assumed.

0 – silent
1 – final result
2 – top level rewrites
3 – all rewrites
4 – conditions
5 – bindings

`optional-test-term` is compared to the final configuration with the result logged for testing

$$\frac{X \triangleright \_\_int32(\_)}{\langle \mathrm{increment}(X)\rangle \to \langle \_\_add(X, \_\_int32(1))\rangle} \quad [\mathrm{integerInc}]$$

$$\frac{X \triangleright \_\_real64(\_)}{\langle \mathrm{increment}(X)\rangle \to \langle \_\_add(X, \_\_real64(1.0))\rangle} \quad [\mathrm{realInc}]$$

```
** !try 5 <increment(__real64(5.5))>

Step 1
  Rewrite call 1 <increment(__real64(5.5))>, ->
  [integerInc] _X |> __int32(_)  --- <increment(_X)> -> <__add(_X, __int32(1))>
  [integerInc] bindings after Theta match { _X=__real64(5.5) }
  [integerInc].SC1 _X |> __int32(_)
  [integerInc].SC1 failed: seek another rule
  [realInc] _X |> __real64(_)  --- <increment(_X)> -> <__add(_X, __real64(1.0))>
  [realInc] bindings after Theta match { _X=__real64(5.5) }
  [realInc].SC1 _X |> __real64(_)
  [realInc] bindings after condition{ _X=__real64(5.5) }
  [realInc] rewrites to <__real64(6.5)>
Normal termination on <__real64(6.5)> after 1 step and 1 call to rewrite()}
```

Gordon Plotkin 1946–

https://en.wikipedia.org/wiki/Gordon_Plotkin

3B Recursive inference rules

So far we have only used conditions to *filter* the application of a rule by ensuring that different types are handled appropriately.

We have also used the builtin function __*add*(_,_) to perform addition efficiently outside of the rewriting system, and seen that there are in fact separate __*add*(_,_) functions for the __*int32* and __*real64* types.

As well as simple matches, we are allowed to put *transitions* above the line. As part of the checking process, the rewriter will recursively call itself on a conditional transition, and that way we can descend into the tree, and handle nested operations.

We shall explore this capability by designing rules to handle nested addition.

Addition, and compound addition

Here is a rule to perform integer addition over a pair of numbers

$$\frac{X \triangleright \_\_int32(\_) \quad Y \triangleright \_\_int32(\_)}{\langle \text{add}(X, Y)\rangle \rightarrow \langle \_\_add(X, Y)\rangle} \qquad [R1]$$

```
** !try 3 <add(__int32(3), __int32(4))>
Step 1
  Rewrite call 1 <add(__int32(3), __int32(4))>, ->
  [R1] _X |> __int32(_)  _Y |> __int32(_)  --- <add(_X, _Y)> -> <__add(_X, _Y)>
  [R1] rewrites to <__int32(7)>
Normal termination on <__int32(7)> after 1 step and 1 call to rewrite()
```

This rule can only handle two numeric operands, and does not work with nested operations. If we write `!try add(add(3,4),5) ->` we get stuck

$$\frac{X \triangleright \_\_int32(\_) \quad Y \triangleright \_\_int32(\_)}{\langle \mathrm{add}(X, Y)\rangle \rightarrow \langle \_\_add(X, Y)\rangle} \quad [R1]$$

```
** !try 5 <add(add(__int32(3), __int32(4)), __int32(5))>
Step 1
  Rewrite call 1 <add(add(__int32(3), __int32(4)), __int32(5))>, ->
  [R1] _X |> __int32(_)  _Y |> __int32(_)  --- <add(_X, _Y)> -> <__add(_X, _Y)>
  [R1] bindings after Theta match { _X=add(__int32(3), __int32(4)), _Y=__int32(5) }
  [R1].SC1 _X |> __int32(_)
  [R1].SC1 failed: seek another rule
Stuck on <add(add(__int32(3), __int32(4)), __int32(5))>
  after 1 step and 1 call to rewrite()
```

So the problem is that $\_X$ is not bound to an integer: we need to evaluate the subexpression to an integer before we can proceed

Here are two rules that work together to handle nested expressions. The first rule handles operands which are both integers. The second rule handles nested expressions, by recursively call itself. The first rule is the inductive base case.

```
-add _X |> __int32(_)  _Y |> __int32(_)  __add(_X, _Y) |> _Z
---
add(_X,_Y) -> _Z

-addExpr _E1 -> _V1 _E2 -> _V2
---
add(_E1, _E2) -> add(_V1, _V2)
```

$$\frac{X \triangleright \_\_int32(\_) \quad Y \triangleright \_\_int32(\_) \quad \_\_add(X, Y) \triangleright Z}{\langle \mathrm{add}(X, Y) \rangle \rightarrow \langle Z \rangle} \quad \text{[add]}$$

$$\frac{\langle E_1 \rangle \rightarrow \langle V_1 \rangle \quad \langle E_2 \rangle \rightarrow \langle V_2 \rangle}{\langle \mathrm{add}(E_1, E_2) \rangle \rightarrow \langle \mathrm{add}(V_1, V_2) \rangle} \quad \text{[addExpr]}$$

When presented with a compound expression, the second rule will call itself repeatedly until the base case can take over. Note that in eSOS, the *order* of the rules is (unhappily) significant.

First we shall demonstrate that the base case still works with !try 4 add(3,4) ->

$$\frac{X \triangleright \_\_int32(\_) \quad Y \triangleright \_\_int32(\_) \quad \_\_add(X, Y) \triangleright Z}{\langle \mathrm{add}(X, Y) \rangle \to \langle Z \rangle} \quad [\mathrm{add}]$$

$$\frac{\langle E_1 \rangle \to \langle V_1 \rangle \quad \langle E_2 \rangle \to \langle V_2 \rangle}{\langle \mathrm{add}(E_1, E_2) \rangle \to \langle \mathrm{add}(V_1, V_2) \rangle} \quad [\mathrm{addExpr}]$$

```
** !try 4 <add(__int32(3), __int32(4))>
Step 1
  Rewrite call 1 <add(__int32(3), __int32(4))>, ->
  [add] _X |> __int32(_)  _Y |> __int32(_)  __add(_X, _Y) |> _Z  --- <add(_X, _Y)> -> <_Z>
  [add].SC1 _X |> __int32(_)
  [add].SC2 _Y |> __int32(_)
  [add].SC3 __add(_X, _Y) |> _Z
  [add] rewrites to <__int32(7)>
Normal termination on <__int32(7)> after 1 step and 1 call to rewrite()
```

Note that this is a level 4 trace so the individual condition evaluations are listed

Now a compound: `!try 4 add(add(3,4),5) ->`

```
** !try 4 <add(add(__int32(3), __int32(4)), __int32(5))>
Step 1
  Rewrite call 1 <add(add(__int32(3), __int32(4)), __int32(5))>, ->
  [add] _X |> __int32(_)  _Y |> __int32(_)  __add(_X, _Y) |> _Z  --- <add(_X, _Y)> -> <_Z>
  [add].SC1 _X |> __int32(_)
  [add].SC1 failed: seek another rule
  [addExpr] <_E1> -> <_V1>  <_E2> -> <_V2>  --- <add(_E1, _E2)> -> <add(_V1, _V2)>
  [addExpr].C1 <_E1> -> <_V1>
    Rewrite call 2 <add(__int32(3), __int32(4))>, ->
    [add] _X |> __int32(_)  _Y |> __int32(_)  __add(_X, _Y) |> _Z  --- <add(_X, _Y)> -> <_Z>
    [add].SC1 _X |> __int32(_)
    [add].SC2 _Y |> __int32(_)
    [add].SC3 __add(_X, _Y) |> _Z
    [add] rewrites to <__int32(7)>
  [addExpr].C2 <_E2> -> <_V2>
    No rewrite required as already terminating state<__int32(5)>
  [addExpr] rewrites to <add(__int32(7), __int32(5))>
Step 2
  Rewrite call 3 <add(__int32(7), __int32(5))>, ->
  [add] _X |> __int32(_)  _Y |> __int32(_)  __add(_X, _Y) |> _Z  --- <add(_X, _Y)> -> <_Z>
  [add].SC1 _X |> __int32(_)
  [add].SC2 _Y |> __int32(_)
  [add].SC3 __add(_X, _Y) |> _Z
  [add] rewrites to <__int32(12)>
Normal termination on <__int32(12)> after 2 steps and 3 calls to rewrite()
```

The expression is evaluated in two *steps*, each roughly corresponding to one execution of a machine code instruction: integer ADD in this case

A step is a top level rewrite which causes the actual program to change. There may be many other rewrites required to decide whether a step can take place

The second step simply applies the old [add] rule to an expression over integers and has a trace that is similar to the previous example but with values 7 and 5

```
** !try 4 <add(add(__int32(3), __int32(4)), __int32(5))>

...

Step 2
  Rewrite call 3 <add(__int32(7), __int32(5))>, ->
   [add] _X |> __int32(_)  _Y |> __int32(_)  __add(_X, _Y) |> _Z  --- <add(_X, _Y)> -> <_Z>
   [add].SC1 _X |> __int32(_)
   [add].SC2 _Y |> __int32(_)
   [add].SC3 __add(_X, _Y) |> _Z
   [add] rewrites to <__int32(12)>
Normal termination on <__int32(12)> after 2 steps and 3 calls to rewrite()
```

The first step is more interesting: it first attempts to use the [add] rule and then [addexpr].

```
** !try 4 <add(add(__int32(3), __int32(4)), __int32(5))>
Step 1
  Rewrite call 1 <add(add(__int32(3), __int32(4)), __int32(5))>, ->
  [add] _X |> __int32(_)  _Y |> __int32(_)  __add(_X, _Y) |> _Z  --- <add(_X, _Y)> -> <_Z>
  [add].SC1 _X |> __int32(_)
  [add].SC1 failed: seek another rule
  [addExpr] <_E1> -> <_V1>  <_E2> -> <_V2>  --- <add(_E1, _E2)> -> <add(_V1, _V2)>
  [addExpr].C1 <_E1> -> <_V1>
    Rewrite call 2 <add(__int32(3), __int32(4))>, ->
    [add] _X |> __int32(_)  _Y |> __int32(_)  __add(_X, _Y) |> _Z  --- <add(_X, _Y)> -> <_Z>
    [add].SC1 _X |> __int32(_)
    [add].SC2 _Y |> __int32(_)
    [add].SC3 __add(_X, _Y) |> _Z
    [add] rewrites to <__int32(7)>
  [addExpr].C2 <_E2> -> <_V2>
    No rewrite required as already terminating state<__int32(5)>
  [addExpr] rewrites to <add(__int32(7), __int32(5))>
...
Normal termination on <__int32(12)> after 2 steps and 3 calls to rewrite()
```

Condition [addexpr].C1 is a transition and this induces a new call to the rewrite() function. Since it is a recursive call, the trace indents the trace to show a sub-rewrite. Only top level rewrites make steps.

On the previous slide, condition [addexpr].C2 did not induce a recursive call because it was already a *value* and nested rewriting always stops when we reach a value. This example has compound additions on both the left and right operands, so we get two nested rewrites

```
!try 3 add(add(3,4),add(5,6)) ->
```

```
** !try 3 <add(add(__int32(3), __int32(4)), add(__int32(5), __int32(6)))>
Step 1
  Rewrite call 1 <add(add(__int32(3), __int32(4)), add(__int32(5), __int32(6)))>, ->
  [add] _X |> __int32(_)  _Y |> __int32(_)  __add(_X, _Y) |> _Z  --- <add(_X, _Y)> -> <_Z>
  [addExpr] <_E1> -> <_V1>  <_E2> -> <_V2>  --- <add(_E1, _E2)> -> <add(_V1, _V2)>
    Rewrite call 2 <add(__int32(3), __int32(4))>, ->
    [add] _X |> __int32(_)  _Y |> __int32(_)  __add(_X, _Y) |> _Z  --- <add(_X, _Y)> -> <_Z>
    [add] rewrites to <__int32(7)>
    Rewrite call 3 <add(__int32(5), __int32(6))>, ->
    [add] _X |> __int32(_)  _Y |> __int32(_)  __add(_X, _Y) |> _Z  --- <add(_X, _Y)> -> <_Z>
    [add] rewrites to <__int32(11)>
  [addExpr] rewrites to <add(__int32(7), __int32(11))>
Step 2
  Rewrite call 4 <add(__int32(7), __int32(11))>, ->
  [add] _X |> __int32(_)  _Y |> __int32(_)  __add(_X, _Y) |> _Z  --- <add(_X, _Y)> -> <_Z>
  [add] rewrites to <__int32(18)>
Normal termination on <__int32(18)> after 2 steps and 4 calls to rewrite()
```

Note that this is a level 3 trace so that we only see the calls to rewrite() and not the individual conditions

As we nest more deeply, we get extra steps: remember one step is required per machine operation. `!try 3 add(add(3,add(4,5)),6) ->`

```
** !try 3 <add(add(__int32(3), add(__int32(4), __int32(5))), __int32(6))>
Step 1
  Rewrite call 1 <add(add(__int32(3), add(__int32(..), __int32(..))), __int32(6))>, ->
  [add] _X |> __int32(_)  _Y |> __int32(_)  __add(_X, _Y) |> _Z  --- <add(_X, _Y)> -> <_Z>
  [addExpr] <_E1> -> <_V1>  <_E2> -> <_V2>  --- <add(_E1, _E2)> -> <add(_V1, _V2)>
    Rewrite call 2 <add(__int32(3), add(__int32(4), __int32(5)))>, ->
    [add] _X |> __int32(_)  _Y |> __int32(_)  __add(_X, _Y) |> _Z  --- <add(_X, _Y)> -> <_Z>
    [addExpr] <_E1> -> <_V1>  <_E2> -> <_V2>  --- <add(_E1, _E2)> -> <add(_V1, _V2)>
      No rewrite required as already terminating state<__int32(3)>
      Rewrite call 3 <add(__int32(4), __int32(5))>, ->
      [add] _X |> __int32(_)  _Y |> __int32(_)  __add(_X, _Y) |> _Z  --- <add(_X, _Y)> -> <_Z>
      [add] rewrites to <__int32(9)>
    [addExpr] rewrites to <add(__int32(3), __int32(9))>
    No rewrite required as already terminating state<__int32(6)>
  [addExpr] rewrites to <add(add(__int32(3), __int32(9)), __int32(6))>
Step 2
  Rewrite call 4 <add(add(__int32(3), __int32(9)), __int32(6))>, ->
  [add] _X |> __int32(_)  _Y |> __int32(_)  __add(_X, _Y) |> _Z  --- <add(_X, _Y)> -> <_Z>
  [addExpr] <_E1> -> <_V1>  <_E2> -> <_V2>  --- <add(_E1, _E2)> -> <add(_V1, _V2)>
    Rewrite call 5 <add(__int32(3), __int32(9))>, ->
    [add] _X |> __int32(_)  _Y |> __int32(_)  __add(_X, _Y) |> _Z  --- <add(_X, _Y)> -> <_Z>
    [add] rewrites to <__int32(12)>
    No rewrite required as already terminating state<__int32(6)>
  [addExpr] rewrites to <add(__int32(12), __int32(6))>
Step 3
  Rewrite call 6 <add(__int32(12), __int32(6))>, ->
  [add] _X |> __int32(_)  _Y |> __int32(_)  __add(_X, _Y) |> _Z  --- <add(_X, _Y)> -> <_Z>
  [add] rewrites to <__int32(18)>
Normal termination on <__int32(18)> after 3 steps and 6 calls to rewrite()
```

Compact traces

We can see that traces quickly become quite dense. Once you have understood the principles of recursive rules, it might be helpful to switch to level 2 traces which just show the outcomes of the rewrites

```
!try 2 add(add(3,add(4,5)),6) ->
```

```
** !try 2 <add(add(__int32(3), add(__int32(4), __int32(5))), __int32(6))>
Step 1
  [addExpr] rewrites to <add(add(__int32(3), __int32(9)), __int32(6))>
Step 2
  [addExpr] rewrites to <add(__int32(12), __int32(6))>
Step 3
  [add] rewrites to <__int32(18)>
Normal termination on <__int32(18)> after 3 steps and 6 calls to rewrite()
```

Peter Mosses 1948 –

https://en.wikipedia.org/wiki/Peter_Mosses

3C Expression nesting and congruence rules

The two-rule version of addition works in ART, but presents some mathematical discomfort. The problem is that, whilst the eSOS interpreter does indeed process conditions left-to-right across a rule, from a mathematical point of view we should really be trying all possible orderings.

If the two arguments to a subtraction have side effects, then the order of evaluation affects the output. Consider, for instance, an expression containing post-increment operators such as $(a + 1\ ) - (a{+}{+} + 1)$ which in prefix form is sub(add(a,1),add(postInc(a),1))

The contents of variable a are being manipulated on the fly during expression evaluations. Assume that before the expression is evaluated, a is bound to 2.

If the right hand addition is performed first, it yields 3 and a is set to 3 before evaluating the left hand addition. The overall expression then yields 1, since $(3 + 1) - 3 = 1$

If the left hand addition is performed first, it yields 3 and the value of a remains at 2. The overall expression then yields 0, since $3 - (2 + 1) = 0$. In either case, a ends up as 3.

The outcome is dependent on evaluation order: our two-rule specification style is then non-confluent

The three-rule style for binary operators

The solution is to use three rules which together enforce an evaluation ordering. We retain the base rule for evaluating actual numbers, but then have a rule that forces left evaluation first, and a rule that then performs right evaluation. These two are called *congruence 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)
```

$$\frac{X \triangleright \_\_int32(\_) \quad Y \triangleright \_\_int32(\_) \quad \_\_add(X, Y) \triangleright Z}{\langle \text{add}(X, Y)\rangle \to \langle Z\rangle} \quad [\text{add}]$$

$$\frac{n \triangleright \_\_int32(\_) \quad \langle E_2\rangle \to \langle I_2\rangle}{\langle \text{add}(n, E_2)\rangle \to \langle \text{add}(n, I_2)\rangle} \quad [\text{addRight}]$$

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

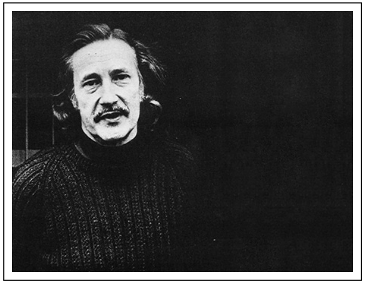
```
** !try 4 <add(add(__int32(3), __int32(4)), add(__int32(4), __int32(5)))>
Step 1
  Rewrite call 1 <add(add(__int32(3), __int32(4)), add(__int32(4), __int32(5)))>, ->
  [add] _X |> __int32(_)  _Y |> __int32(_)  __add(_X, _Y) |> _Z  --- <add(_X, _Y)> -> <_Z>
  [add].SC1 _X |> __int32(_)
  [add].SC1 failed: seek another rule
  [addRight] _n |> __int32(_)  <_E2> -> <_I2>  --- <add(_n, _E2)> -> <add(_n, _I2)>
  [addRight].SC1 _n |> __int32(_)
  [addRight].SC1 failed: seek another rule
  [addLeft] <_E1> -> <_I1>  --- <add(_E1, _E2)> -> <add(_I1, _E2)>
  [addLeft].C1 <_E1> -> <_I1>
    Rewrite call 2 <add(__int32(3), __int32(4))>, ->
    [add] _X |> __int32(_)  _Y |> __int32(_)  __add(_X, _Y) |> _Z  --- <add(_X, _Y)> -> <_Z>
    [add].SC1 _X |> __int32(_)
    [add].SC2 _Y |> __int32(_)
    [add].SC3 __add(_X, _Y) |> _Z
    [add] rewrites to <__int32(7)>
  [addLeft] rewrites to <add(__int32(7), add(__int32(4), __int32(5)))>
```

```
Step 2
  Rewrite call 3 <add(__int32(7), add(__int32(4), __int32(5)))>, ->
  [add] _X |> __int32(_)  _Y |> __int32(_)  __add(_X, _Y) |> _Z  --- <add(_X, _Y)> -> <_Z>
  [add].SC1 _X |> __int32(_)
  [add].SC2 _Y |> __int32(_)
  [add].SC2 failed: seek another rule
  [addRight] _n |> __int32(_)  <_E2> -> <_I2>  --- <add(_n, _E2)> -> <add(_n, _I2)>
  [addRight].SC1 _n |> __int32(_)
  [addRight].C2 <_E2> -> <_I2>
    Rewrite call 4 <add(__int32(4), __int32(5))>, ->
    [add] _X |> __int32(_)  _Y |> __int32(_)  __add(_X, _Y) |> _Z  --- <add(_X, _Y)> -> <_Z>
    [add].SC1 _X |> __int32(_)
    [add].SC2 _Y |> __int32(_)
    [add].SC3 __add(_X, _Y) |> _Z
    [add] rewrites to <__int32(9)>
  [addRight] rewrites to <add(__int32(7), __int32(9))>
```

```
Step 3
 Rewrite call 5 <add(__int32(7), __int32(9))>, ->
  [add] _X |> __int32(_)  _Y |> __int32(_)  __add(_X, _Y) |> _Z  --- <add(_X, _Y)> -> <_Z>
  [add].SC1 _X |> __int32(_)
  [add].SC2 _Y |> __int32(_)
  [add].SC3 __add(_X, _Y) |> _Z
  [add] rewrites to <__int32(16)>
Normal termination on <__int32(16)> after 3 steps and 5 calls to rewrite()
```

Or more compactly...

```
** !try 2 <add(add(__int32(3), __int32(4)), add(__int32(4), __int32(5)))>
Step 1
  [addLeft] rewrites to <add(__int32(7), add(__int32(4), __int32(5)))>
Step 2
  [addRight] rewrites to <add(__int32(7), __int32(9))>
Step 3
  [add] rewrites to <__int32(16)>
Normal termination on <__int32(16)> after 3 steps and 5 calls to rewrite()
```



Christopher Strachey 1916–1975

https://en.wikipedia.org/wiki/Christopher_Strachey

3D The store, program variables, output and simple sequencing

Imperative, procedural programs rely on mutable variables and implicit sequencing from one statement to another. We also want to be able to observe program effects by sending them to some output.

Variable s which change their contents and program outputs are examples of *side effects* and we use *semantic entities* to capture their state as the rewriting process proceeds.

A natural structure to represent mutable variables is a *store* which is a set of bindings from variable names to values. We use a *__map* type which is traditionally called sigma (σ).

For output, we use a *__list* called phi (ϕ)

Assignment of an integer to a variable

```
-assign
_n |> __int32(_)  __put(_sig, _X, _n) |> _sig1
---
assign(_X, _n), _sig -> __done, _sig1
```

$$\frac{n \triangleright \_\_int32(\_) \quad \_\_put(\sigma, X, n) \triangleright \sigma_1}{\langle \text{assign}(X, n), \sigma\rangle \rightarrow \langle \_\_done, \sigma_1\rangle} \quad \text{[assign]}$$

```
** !try 3 <assign(tmp, __int32(32)), {}>
Step 1
  Rewrite call 1 <assign(tmp, __int32(32)), {}>, ->
  [assign] _n |> __int32(_)  __put(_sig, _X, _n) |> _sig1  ---
                 <assign(_X, _n), _sig> -> <__done, _sig1>
  [assign] rewrites to <__done, {tmp=__int32(32)}>
Normal termination on <__done, {tmp=__int32(32)}> after 1 step and 1 call to rewrite()
```

Assignment resolution

```
-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
```

$$\frac{n \triangleright \_\_int32(\_) \quad \_\_put(\sigma, X, n) \triangleright \sigma_1}{\langle \text{assign}(X, n), \sigma \rangle \to \langle \_\_done, \sigma_1 \rangle} \quad [\text{assign}]$$

$$\frac{\langle E, \sigma \rangle \to \langle I, \sigma' \rangle}{\langle \text{assign}(X, E), \sigma \rangle \to \langle \text{assign}(X, I), \sigma' \rangle} \quad [\text{assignResolve}]$$

```
** !try 2 <assign(a, add(__int32(3), __int32(4))), {}>
Step 1
  [assignResolve] rewrites to <assign(a, __int32(7)), {}>
Step 2
  [assign] rewrites to <__done, {a=__int32(7)}>
Normal termination on <__done, {a=__int32(7)}> after 2 steps and 3 calls to rewrite()
```

Sequencing

```
-sequenceDone   ---
seq(__done, _C) -> _C

-sequence   _C1 -> _C1P
---
seq(_C1, _C2) -> seq(_C1P, _C2)
```

$$\frac{}{\langle \mathsf{seq}(\_\_done, C)\rangle \rightarrow \langle C\rangle} \quad [\mathsf{sequenceDone}]$$

$$\frac{\langle C_1\rangle \rightarrow \langle {C_1}'\rangle}{\langle \mathsf{seq}(C_1, C_2)\rangle \rightarrow \langle \mathsf{seq}({C_1}', C_2)\rangle} \quad [\mathsf{sequence}]$$

```
** !try 3 <seq(__done, __int32(7))>
Step 1
  Rewrite call 1 <seq(__done, __int32(7))>, ->
  [sequenceDone]   --- <seq(__done, _C)> -> <_C>
  [sequenceDone] rewrites to <__int32(7)>
Normal termination on <__int32(7)> after 1 step and 1 call to rewrite()
```

Sequenced store operations using previous rules

tmp1 := 32; tmp2 := 64; tmp1 := 128;

```
!try 2 seq(seq(assign(tmp1, 32),assign(tmp2, 64)),assign(tmp1,128)), __map() ->

** !try 2 <seq(seq(assign(tmp1, __int32(32)), assign(tmp2, __int32(64))), assign(tmp1, __int32(128))
Step 1
  [sequence] rewrites to <seq(seq(__done, assign(tmp2, __int32(..))), assign(tmp1, __int32(128))),
                                                {tmp1=__int32(32)}>
Step 2
  [sequence] rewrites to  <seq(assign(tmp2, __int32(64)), assign(tmp1, __int32(128))),
                                                 {tmp1=__int32(32)}>
Step 3
  [sequence] rewrites to  <seq(__done, assign(tmp1, __int32(128))),
                                                {tmp2=__int32(64), tmp1=__int32(32)}>
Step 4
  [sequenceDone] rewrites to  <assign(tmp1, __int32(128)),
                                               {tmp2=__int32(64), tmp1=__int32(32)}>
Step 5
  [assign] rewrites to <__done,
                                               {tmp2=__int32(64), tmp1=__int32(128)}>
Normal termination on <__done, {tmp2=__int32(64), tmp1=__int32(128)}>
after 5 steps and 9 calls to rewrite()
```

Dereferencing a variable

```
-deref
__get(_sig, _R) |> _Z
---
deref(_R),_sig -> _Z, _sig
```

$$\frac{\_\_get(\sigma, R) \triangleright Z}{\langle \text{deref}(R), \sigma\rangle \to \langle Z, \sigma\rangle} \quad \text{[deref]}$$

-

```
Step 3
  Rewrite call 4 <assign(b, deref(a)), {a=__int32(3)}>, ->
  [assign] _n |> __int32(_)  __put(_sig, _X, _n) |> _sig1  --- <assign(_X, _n), _sig> -> <_
  [assignResolve] <_E, _sig> -> <_I, _sigP>  --- <assign(_X, _E), _sig> -> <assign(_X, _I)
    Rewrite call 5 <deref(a), {a=__int32(3)}>, ->
    [deref] __get(_sig, _R) |> _Z  --- <deref(_R), _sig> -> <_Z, _sig>
    [deref] rewrites to <__int32(3), {a=__int32(3)}>
  [assignResolve] rewrites to <assign(b, __int32(3)), {a=__int32(3)}>
```

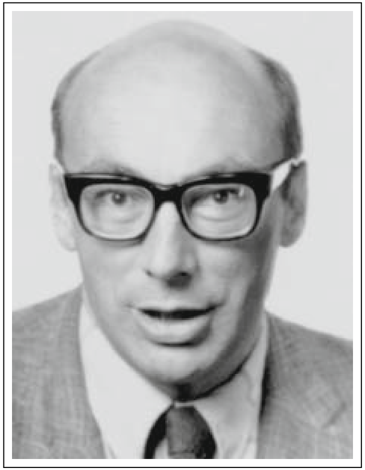
Output

```
-output _x,_phi1 ->_y,_phi2
---
output(_x),_phi1 -> __done, __append(_phi2,_y)
```

$$\frac{\langle x, \phi_1\rangle \rightarrow \langle y, \phi_2\rangle}{\langle \text{output}(x), \phi_1\rangle \rightarrow \langle \_\_done, \_\_append(\phi_2, y)\rangle} \quad \text{[output]}$$

```
** !try 3 <seq(output(__int32(4)), output(__int32(5))), []>
Step 1
  Rewrite call 1 <seq(output(__int32(4)), output(__int32(5))), []>, ->
  [sequenceDone]   --- <seq(__done, _C), _phi> -> <_C, _phi>
  [sequenceDone] Theta match failed: seek another rule
  [sequence] <_C1, _phi1> -> <_C1P, _phi2>  --- <seq(_C1, _C2), _phi1> -> <seq(_C1P, _C2), _phi2>
    Rewrite call 2 <output(__int32(4)), []>, ->
    [output] <_x, _phi1> -> <_y, _phi2>  --- <output(_x), _phi1> -> <__done, __append(_phi2, _y)>
      No rewrite required as already terminating state<__int32(4), []>
    [output] rewrites to <__done, [__int32(4)]>
  [sequence] rewrites to <seq(__done, output(__int32(5))), [__int32(4)]>
```

```
Step 2
  Rewrite call 3 <seq(__done, output(__int32(5))), [__int32(4)]>, ->
  [sequenceDone]   --- <seq(__done, _C), _phi> -> <_C, _phi>
  [sequenceDone] rewrites to <output(__int32(5)), [__int32(4)]>
Step 3
  Rewrite call 4 <output(__int32(5)), [__int32(4)]>, ->
  [output] <_x, _phi1> -> <_y, _phi2>  --- <output(_x), _phi1> -> <__done, __append(_phi2, _y)>
    No rewrite required as already terminating state<__int32(5), [__int32(4)]>
  [output] rewrites to <__done, [__int32(4), __int32(5)]>
Normal termination on <__done, [__int32(4), __int32(5)]> after 3 steps and 4 calls to rewrite()
```



Peter Landin 1930–2009

https://en.wikipedia.org/wiki/Peter_Landin

3E Control flow

Early programming languages provided control flow commands that mirrored the underlying hardware: unconditional jumping to a labeled statement (`GOTO L`), and conditional branching to a labeled statement (`IF` *`predicate`* `GOTO L`).

During the 1960s it became increasingly clear that unrestricted control flow changes (so-called spaghetti programming) reduced the comprehensibility of programs. In 1968 Edsger Dijkstra wrote his famous letter *Go to statement considered harmful* to the editor of Communications of the ACM which you can read at `https://dl.acm.org/doi/10.1145/362929.362947`

Java is one of the first languages to outlaw `GOTO` altogether (in detail, Java has syntax for `GOTO` but the static semantics are to generate an error message!) Instead, we program using control flow structures which have a *single* predecessor statement and a single *successor* statement. These so-called *D-structures* (after Dijkstra) can nest. We are still allowed `break` and `continue` statements because they do not breach the single-predecessor/successor rule.

D-structures fit nicely with a reduction model of execution because they rewrite way in a nicely nested fashion. `GOTO` requires copying of chunks of program which breaches the structural constraint.

if-then-else

```
-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
```

$$\frac{}{\langle \text{if}(\_\_\textit{boolean}(\text{True}), C_1, C_2), \sigma\rangle \rightarrow \langle C_1, \sigma\rangle} \quad [\text{ifTrue}]$$

$$\frac{}{\langle \text{if}(\_\_\textit{boolean}(\text{False}), C_1, C_2), \sigma\rangle \rightarrow \langle C_2, \sigma\rangle} \quad [\text{ifFalse}]$$

$$\frac{\langle E, \sigma\rangle \rightarrow \langle E', \sigma'\rangle}{\langle \text{if}(E, C_1, C_2), \sigma\rangle \rightarrow \langle \text{if}(E', C_1, C_2), \sigma'\rangle} \quad [\text{ifResolve}]$$

```
** !try 3 <if(gt(__int32(3), __int32(4)), __int32(1), __int32(0)), {}>
Step 1
  Rewrite call 1 <if(gt(__int32(3), __int32(4)), __int32(1), __int32(0)), {}>, ->
  [ifTrue]   --- <if(__boolean(True), _C1, _C2), _sig> -> <_C1, _sig>
  [ifTrue] Theta match failed: seek another rule
  [ifFalse]   --- <if(__boolean(False), _C1, _C2), _sig> -> <_C2, _sig>
  [ifFalse] Theta match failed: seek another rule
  [ifResolve] <_E, _sig> -> <_EP, _sigP>  --- <if(_E, _C1, _C2), _sig> -> <if(_EP, _C1, _C2), _sigP>
    Rewrite call 2 <gt(__int32(3), __int32(4)), {}>, ->
    [gt] _n1 |> __int32(_)  _n2 |> __int32(_)  --- <gt(_n1, _n2), _sig> -> <__gt(_n1, _n2), _sig>
    [gt] rewrites to <__boolean(False), {}>
  [ifResolve] rewrites to <if(__boolean(False), __int32(1), __int32(0)), {}>
Step 2
  Rewrite call 3 <if(__boolean(False), __int32(1), __int32(0)), {}>, ->
  [ifTrue]   --- <if(__boolean(True), _C1, _C2), _sig> -> <_C1, _sig>
  [ifTrue] Theta match failed: seek another rule
  [ifFalse]   --- <if(__boolean(False), _C1, _C2), _sig> -> <_C2, _sig>
  [ifFalse] rewrites to <__int32(0), {}>
Normal termination on <__int32(0), {}> after 2 steps and 3 calls to rewrite()
```

```
** !try 3 <if(gt(__int32(4), __int32(3)), __int32(1), __int32(0)), {}>
Step 1
  Rewrite call 1 <if(gt(__int32(4), __int32(3)), __int32(1), __int32(0)), {}>, ->
  [ifTrue]   --- <if(__boolean(True), _C1, _C2), _sig> -> <_C1, _sig>
  [ifTrue] Theta match failed: seek another rule
  [ifFalse]   --- <if(__boolean(False), _C1, _C2), _sig> -> <_C2, _sig>
  [ifFalse] Theta match failed: seek another rule
  [ifResolve] <_E, _sig> -> <_EP, _sigP>  --- <if(_E, _C1, _C2), _sig> -> <if(_EP, _C1, _C2), _sigP>
    Rewrite call 2 <gt(__int32(4), __int32(3)), {}>, ->
    [gt] _n1 |> __int32(_)  _n2 |> __int32(_)  --- <gt(_n1, _n2), _sig> -> <__gt(_n1, _n2), _sig>
    [gt] rewrites to <__boolean(True), {}>
  [ifResolve] rewrites to <if(__boolean(True), __int32(1), __int32(0)), {}>
Step 2
  Rewrite call 3 <if(__boolean(True), __int32(1), __int32(0)), {}>, ->
  [ifTrue]   --- <if(__boolean(True), _C1, _C2), _sig> -> <_C1, _sig>
  [ifTrue] rewrites to <__int32(1), {}>
Normal termination on <__int32(1), {}> after 2 steps and 3 calls to rewrite()
```

while-do

We unwrap while-do into if-then using the rewrite trick we described earlier. This translates into a simple unconditonal rewrite rule

```
-while
---
while(_E, _C),_sig -> if(_E, seq(_C, while(_E,_C)), __done), _sig
```

$$\frac{}{\langle \text{while}(E, C), \sigma\rangle \rightarrow \langle \text{if}(E, \text{seq}(C, \text{while}(E, C)), \_\_\mathit{done}), \sigma\rangle}\;[\text{while}]$$

Now, by their nature, traces from while-do loops can be rather long.

```
** !try 2 <seq(assign(a, __int32(2)), while(gt(deref(a), __int32(0)),
     assign(a, sub(deref(a), __int32(1))))), {}>
Step 1
  [sequence] rewrites to <seq(__done, while(gt(deref(..), __int32(..)),
     assign(a, sub(.., ..)))), {a=__int32(2)}>
Step 2
  [sequenceDone] rewrites to <while(gt(deref(a), __int32(0)),
     assign(a, sub(deref(..), __int32(..)))), {a=__int32(2)}>
Step 3
  [while] rewrites to <if(gt(deref(a), __int32(0)), seq(assign(a, sub(.., ..)),
     while(gt(.., ..), assign(.., ..))), __done), {a=__int32(2)}>
Step 4
  [ifResolve] rewrites to <if(gt(__int32(2), __int32(0)),
     seq(assign(a, sub(.., ..)), while(gt(.., ..), assign(.., ..))), __done), {a=__int32(2)}>
Step 5
  [ifResolve] rewrites to <if(__boolean(True), seq(assign(a, sub(.., ..)),
     while(gt(.., ..), assign(.., ..))), __done), {a=__int32(2)}>
Step 6
  [ifTrue] rewrites to <seq(assign(a, sub(deref(..), __int32(..))),
     while(gt(deref(..), __int32(..)), assign(a, sub(.., ..)))), {a=__int32(2)}>
Step 7
  [sequence] rewrites to <seq(assign(a, sub(__int32(..), __int32(..))),
     while(gt(deref(..), __int32(..)), assign(a, sub(.., ..)))), {a=__int32(2)}>
Step 8
  [sequence] rewrites to <seq(assign(a, __int32(1)), while(gt(deref(..), __int32(..)),
     assign(a, sub(.., ..)))), {a=__int32(2)}>
```

```
Step 9
  [sequence] rewrites to <seq(__done, while(gt(deref(..), __int32(..)),
     assign(a, sub(.., ..)))), {a=__int32(1)}>
Step 10
  [sequenceDone] rewrites to <while(gt(deref(a), __int32(0)),
     assign(a, sub(deref(..), __int32(..)))), {a=__int32(1)}>
Step 11
  [while] rewrites to <if(gt(deref(a), __int32(0)), seq(assign(a, sub(.., ..)),
     while(gt(.., ..), assign(.., ..))), __done), {a=__int32(1)}>
Step 12
  [ifResolve] rewrites to <if(gt(__int32(1), __int32(0)),
     seq(assign(a, sub(.., ..)), while(gt(.., ..), assign(.., ..))), __done), {a=__int32(1)}>
Step 13
  [ifResolve] rewrites to <if(__boolean(True), seq(assign(a, sub(.., ..)),
     while(gt(.., ..), assign(.., ..))), __done), {a=__int32(1)}>
Step 14
  [ifTrue] rewrites to <seq(assign(a, sub(deref(..), __int32(..))),
     while(gt(deref(..), __int32(..)), assign(a, sub(.., ..)))), {a=__int32(1)}>
Step 15
  [sequence] rewrites to <seq(assign(a, sub(__int32(..), __int32(..))),
      while(gt(deref(..), __int32(..)), assign(a, sub(.., ..)))), {a=__int32(1)}>
Step 16
  [sequence] rewrites to <seq(assign(a, __int32(0)),
     while(gt(deref(..), __int32(..)), assign(a, sub(.., ..)))), {a=__int32(1)}>
```

```
Step 17
  [sequence] rewrites to <seq(__done, while(gt(deref(..), __int32(..)),
     assign(a, sub(.., ..)))), {a=__int32(0)}>
Step 18
  [sequenceDone] rewrites to <while(gt(deref(a), __int32(0)),
     assign(a, sub(deref(..), __int32(..)))), {a=__int32(0)}>
Step 19
  [while] rewrites to <if(gt(deref(a), __int32(0)),
     seq(assign(a, sub(.., ..)), while(gt(.., ..), assign(.., ..))), __done), {a=__int32(0)}>
Step 20
  [ifResolve] rewrites to <if(gt(__int32(0), __int32(0)),
     seq(assign(a, sub(.., ..)), while(gt(.., ..), assign(.., ..))), __done), {a=__int32(0)}>
Step 21
  [ifResolve] rewrites to <if(__boolean(False),
     seq(assign(a, sub(.., ..)), while(gt(.., ..), assign(.., ..))), __done), {a=__int32(0)}>
Step 22
  [ifFalse] rewrites to <__done, {a=__int32(0)}>
Normal termination on <__done, {a=__int32(0)}> after 22 steps and 44 calls to rewrite()
```

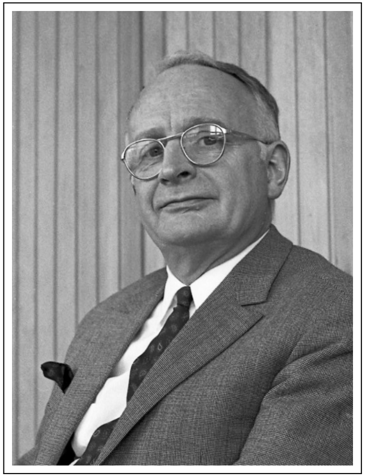
Other structures

We have seen how to do selection and iteration using `if-then` and `while-do`, and in fact `while-do` only needs a simple rewrite.

Other control flow statements that we should consider include the `switch` statement (also known as a `case` statement), `for` loops, `do-while`, `repeat-until` and the `break` and `continue` early-exit jumps.

Now, it turns out that structures can be implemented in terms of our basic `if-then` rewrite rule, and thus we could rewrite using the same approach as with `while-do`.

Try it



Adriaan van Wijngaarden 1916–1987
https://en.wikipedia.org/wiki/Adriaan_van_Wijngaarden

3F Big step semantics

$$\frac{n_1 \triangleright \_\_int32(\_) \quad n_2 \triangleright \_\_int32(\_)}{\langle \mathsf{sub}(n_1, n_2), \sigma\rangle \rightarrow \langle \_\_sub(n_1, n_2), \sigma\rangle} \quad [\mathsf{sub}]$$

$$\frac{n \triangleright \_\_int32(\_) \quad \langle E_2, \sigma\rangle \rightarrow \langle I_2, \sigma'\rangle}{\langle \mathsf{sub}(n, E_2), \sigma\rangle \rightarrow \langle \mathsf{sub}(n, I_2), \sigma'\rangle} \quad [\mathsf{subRight}]$$

$$\frac{\langle E_1, \sigma\rangle \rightarrow \langle I_1, \sigma'\rangle}{\langle \mathsf{sub}(E_1, E_2), \sigma\rangle \rightarrow \langle \mathsf{sub}(I_1, E_2), \sigma'\rangle} \quad [\mathsf{subLeft}]$$

$$\frac{\_\_get(\sigma, R) \triangleright Z}{\langle \mathsf{deref}(R), \sigma\rangle \rightarrow \langle Z, \sigma\rangle} \quad [\mathsf{deref}]$$

$$\frac{n_1 \triangleright \_\_int32(\_) \quad n_2 \triangleright \_\_int32(\_)}{\langle \mathsf{gt}(n_1, n_2), \sigma\rangle \rightarrow \langle \_\_gt(n_1, n_2), \sigma\rangle} \quad [\mathsf{gt}]$$

$$\frac{n \triangleright \_\_int32(\_) \quad \langle E_2, \sigma\rangle \rightarrow \langle I_2, \sigma'\rangle}{\langle \mathsf{gt}(n, E_2), \sigma\rangle \rightarrow \langle \mathsf{gt}(n, I_2), \sigma'\rangle} \quad [\mathsf{gtRight}]$$

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

$$\frac{}{\langle \mathsf{if}(\_\_boolean(\mathsf{True}), C_1, C_2), \sigma\rangle \to \langle C_1, \sigma\rangle} \quad [\mathsf{ifTrue}]$$

$$\frac{}{\langle \mathsf{if}(\_\_boolean(\mathsf{False}), C_1, C_2), \sigma\rangle \to \langle C_2, \sigma\rangle} \quad [\mathsf{ifFalse}]$$

$$\frac{\langle E, \sigma\rangle \to \langle E', \sigma'\rangle}{\langle \mathsf{if}(E, C_1, C_2), \sigma\rangle \to \langle \mathsf{if}(E', C_1, C_2), \sigma'\rangle} \quad [\mathsf{ifResolve}]$$

$$\frac{n_1 \triangleright \_\_int32(\_) \quad n_2 \triangleright \_\_int32(\_)}{\langle \mathsf{ne}(n_1, n_2), \sigma\rangle \to \langle \_\_ne(n_1, n_2), \sigma\rangle} \quad [\mathsf{ne}]$$

$$\frac{n \triangleright \_\_int32(\_) \quad \langle E_2, \sigma\rangle \to \langle I_2, \sigma'\rangle}{\langle \mathsf{ne}(n, E_2), \sigma\rangle \to \langle \mathsf{ne}(n, I_2), \sigma'\rangle} \quad [\mathsf{neRight}]$$

$$\frac{\langle E_1, \sigma\rangle \to \langle I_1, \sigma'\rangle}{\langle \mathsf{ne}(E_1, E_2), \sigma\rangle \to \langle \mathsf{ne}(I_1, E_2), \sigma'\rangle} \quad [\mathsf{neLeft}]$$

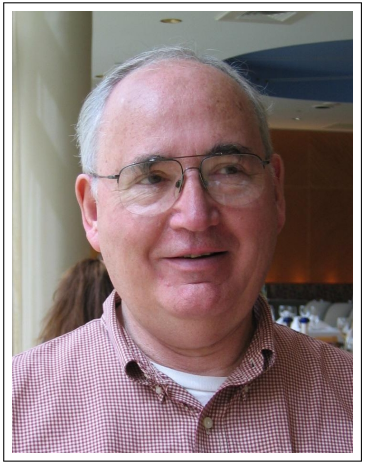
$$\frac{}{\langle \mathsf{seq}(\_\_done, C), \sigma\rangle \rightarrow \langle C, \sigma\rangle} \quad \text{[sequenceDone]}$$

$$\frac{\langle C_1, \sigma\rangle \rightarrow \langle {C_1}', \sigma'\rangle}{\langle \mathsf{seq}(C_1, C_2), \sigma\rangle \rightarrow \langle \mathsf{seq}({C_1}', C_2), \sigma'\rangle} \quad \text{[sequence]}$$

$$\frac{}{\langle \mathsf{while}(E, C), \sigma\rangle \rightarrow \langle \mathsf{if}(E, \mathsf{seq}(C, \mathsf{while}(E, C)), \_\_done), \sigma\rangle} \quad \text{[while]}$$

$$\frac{n \triangleright \_\_int32(\_)}{\langle \mathsf{assign}(X, n), \sigma\rangle \rightarrow \langle \_\_done, \_\_put(\sigma, X, n)\rangle} \quad \text{[assign]}$$

$$\frac{\langle E, \sigma\rangle \rightarrow \langle I, \sigma'\rangle}{\langle \mathsf{assign}(X, E), \sigma\rangle \rightarrow \langle \mathsf{assign}(X, I), \sigma'\rangle} \quad \text{[assignResolve]}$$



Dana Scott 1932–
https://en.wikipedia.org/wiki/Dana_Scott

4 Syntax specification and analysis

Section	Topic
4A	Internal and external syntax
4B	Limited but easy parsing: Ordered Singleton Backtrack Recursive Descent
4C	From strings to tokens - lexicalisation
4D	The space of parsing algorithms
4E	Generalised context free parsing
4F	Generalised context free and regular lexical analysis

4A Internal and external syntax

We can identify four main parts of a language processor

1. The *front end* (F) translates from external syntax to internal syntax
2. The *back end* (B) translates from internal syntax to target syntax
3. The *middle end* (M) translates from internal syntax to internal syntax, transforming the program in some way
4. The *execution end* (X) executes the program, consuming data and generating the program's effects

There are at least three representations of the user program in use here: external (E), internal (I) and target (T). In the case of `javac`, the Java compiler, syntax E is the Java language itself, syntax T is Java byte code and syntax I is a property of the particular java compiler implementation. Some compiler toolkits, for instance the gcc toolchain, have multiple internal syntaxes, each optimised for a particular task.

In this section we focus on the design and specification of the front end

Middle end first

As programmers, it is natural for us to focus on the external ‘look’ of a language, but on this course you have been strongly encouraged to take a ‘feature-first’ approach in which we identify the underlying semantic notions before constructing a human-comfortable external notation.

In our model, we use a strictly unambiguous prefix-form to describe program terms inside the compiler. Terms are trees, and we can thus capture the compositional, hierarchical structure of programs directly

The emphasis on *one* form for each language feature eases interpreter and compiler implementation, allowing us to generate code or interpret actions as a single flat case analysis.

I call this the *middle end first* design strategy

Thinking about programs and their transformations

A nest of prefix function instances is not the only way to think about program texts.

Alternatives include

1. *three-address code* (3A) which looks a little like an idealised assembly language
2. *Static Single Assignment* (SSA) form which is a control-flow graph based representation in which variables are never redefined
3. *Continuation Passing Style* (CPS) in which each operation is represented as a function which includes as a parameter the rest of the program term to be executed.

However, it turns out that CPS is naturally expressible as nested functions; SSA is a subset of CPS and 3A can be transformed to SSA. So our internal syntax has broad utility, although the way different researchers think about what is encoded into internal syntax differs.

These issues are very important for the writers of optimising compilers, where many important transformation algorithms can be expressed elegantly in SSA and CPS. However, we are focusing solely on interpreters, which have only a vestigial middle-end.

Syntax for humans

Designing a comfortable and expressive programming syntax is hard, and there are no clear rules.

Most programming language design appeal strongly to notations with which the users may already be familiar, including arithmetic operators that we learn at school, basic English language words to describe conditionals and loops, and in particular existing widely used languages. As such, programming language syntax development is probably best viewed as an evolutionary process.

There is always a tension between conciseness for the experienced used and clarity for the neophyte: consider Algol-68's two equivalent notations for `if` statements:

if a=b **then** s1() **else** s2 **fi**

(a=b | s1() | s2)

There is also always a tension between enumeration and orthogonality.
Layout is critical to understanding too – especially when trying to comprehend deeply nested structures

Resources for thinking about syntax

One of the most interesting places to read about the zoo of programming notations is the Rosetta Code site at `http://rosettacode.org/wiki/Rosetta_Code` which attempts to show implementations of standard programs in as many languages as possible (at the time of writing, just over 800).

Another good way to escape the cultural blinkers that we all seem to acquire from our first attempts at programming is to look at deliberately unhelpful languages, many of which are designed as parodies or satires on conventional languages. You can read more at `https://en.wikipedia.org/wiki/Esoteric_programming_language`[4] and `https://esolangs.org`

Some of these attempts contain useful ideas that challenge our fundamental notions. I particularly like the **Whitespace** programming language in which all semantically meaningfull programs are entirely composed of whitespace, and the printable characters are treated as comments. Though obviously satirical in intent, this allows Whitespace programs to be embedded within conventional programs.

[4] *Trigger alert: some bad language may be encountered on these esoteric language pages*

Taming the syntax monster

The front ends for early languages, including FORTRAN, were essentially written in an *ad hoc* way, and it shows. To this day, FORTRAN syntax presents a non-trivial challenge for many front end toolsets. At the time of writing, FORTRAN registers as the 23rd most used programming language world-wide (above ADA, Lisp, Scala, Lua and Haskell) so the difficulty of processing FORTRAN source code is of practical importance. For instance, the Met Office UK weather model comprises around one million lines of FORTRAN.

John Backus, who led the FORTRAN effort said that the compiler required 18 person-years of effort to create: much of that was consumed by middle end code improvers.

FORTRAN has some interesting syntactic quirks: for instance all spaces may be omitted from a FORTRAN program without channging its meaning. At the time, this seemed like a desirable feature. As the Sun FORTRAN reference manual purportedly said: *Consistently separating words by spaces became a general custom about the tenth century A. D., and lasted until about 1957, when FORTRAN abandoned the practice.*

Another 'good' idea which was not used in many other languages was association of type with particular identifier names. In FORTRAN, variables beginning with **I** through **N** are by default integer. Other identifiers have type real. A declaration can override this, leading to the old joke that with FORTRAN, God is real unless declared integer.

Backus and Naur and their notation

In 1958 a meeting was held at ETH Zurich between European computer scientists and some prominent American programming language experts (including Backus) to discuss proposals for a new *International Algebraic Language* which developed into the proposal for Algol-58.

Algol-58 is noted more for its influence on later languages than for widespread takeup. It introduced the concept of block structure, used $:=$ for assignment. The languages immediate successor, Algol-60, was a huge success and is the direct ancestor of most of the block structured languages we use today.

Backus became interested in developing a systematic approach to external syntax analysis as a way of reducing front end implementation to a clerical process. He developed *Backus Normal Form* to describe Algol-58's external syntax.

BNF was revised and expanded by Peter Naur, and at the suggestion of Donald Knuth, the notation is now known as Backus-Naur Form (BNF).

Mathematical games for lexicalisation and parsing

The original motivation for BNF was to provide a concise and precise specification of the allowable Algol programs.

Once that had been achieved, there was an explosion of research into algorithms that could take a BNF specification and automatically generate a program which could perform syntax analysis.

The formal roots of this work liek in the work of Axel Thue (1863–1922) who investigated string rewriting systems as part of early efforts to formalise mathematics and enable automated theorem proving – what we now call a String Rewriting System was historically called a semi-Thue system.

Backus was aware of work in linguistics, in which theoreticians such as Noam Chomsky had been investigating mathematical models of natural (human) speech, and the equivalence of Algol's BNF to formal grammars was soon noted.

Formal grammars

A *formal grammar* constitutes a convenient way to describe string rewriting in a way that emphasises the compositional structure of phrases. It is inspired by the *informal grammars* used to teach the structure of human languages.

A formal grammar is a 4-tuple $< N, T, S, P >$ with $S \in N$ and $N \cap T = \emptyset$. N is a finite set of nonterminal symbols. T is a finite set of terminal symbols. $S \in N$ is a distinguished member of N called the *start symbol*. P is a set of *productions* which in their most general form are rewrite rules from a nonempty string in $(N \cup T)^+$ to a possibly empty string in $(N \cup T)^*$.

We use a set of conventions to avoid having to repeatedly write the same definitions. In most of the formal grammar literature, lower case letters $a \dots g$ denote terminals, lower case letters $u \dots z$ denote (possibly empty) strings of terminals, upper case letters denote nonterminals and Greek letters $\alpha \dots \delta$ denote strings over both terminals and nonterminals.

The empty string (of length zero) is denoted by ϵ

Symbols are simply atomic mathematical objects. Typically we think of symbols as either being individual letters, or as being *tokens* which may represent families of substrings.

Generating sentences

We can think of a grammar as a *language generating device* which operates by starting with just the start symbol S and then rewriting strings over $(N \cup T)$ until all of the nonterminals have been removed.

A *sentential form* is a (possibly empty) string in $(N \cup T)^*$ A sentential form containing no nonterminals is called a *sentence* – it will thus be a member of T^*. The set of sentences generated by a formal grammar Γ is the *language* of that grammar, $L(\Gamma)$.

The rewriting relation defined by the productions is in a slightly different form to the ones that we looked at in the string rewriting part of Section 2 in that with a grammar, we look for occurrences of a production left hand side anywhere in the sentential form whereas in Section 2, the left hand side of a rewrite had to match the entire string. Technically, then, the rewrite relation from a production such as $\beta \rightarrow \delta$ has to be *extended* to the string rewrite $\alpha\beta\gamma \rightarrow \alpha\delta\gamma$ to express the notion that can have arbitrary strings of elements from $N \cup T$ on either side of the rewrite site.

The Chomsky hierarchy

In 1956 Noam Chomsky published the paper *Three models for the description of language* in which he gave *hierarchy* of grammar types whose associated languages nest.

We can define this hierarchy using constraints on the form of productions in a formal grammar.

Type	Languages	Automaton	Constraints
Type-0	Recursively enumerable	Turing machine	$\gamma \to \delta, \gamma \neq \epsilon$
Type-1	Context sensitive	Linear bounded ND Turing machine	$\alpha A \beta \to \alpha \gamma \beta, \gamma \neq \epsilon$
Type-2	Context free	ND push down automaton	$A \to \alpha$
Type-3	Regular	Finite state automaton	$A \to a, A \to aB$
Type-4	*Lookup table*	*Finite map*	$A \to u$

The Type-4 (lookup table) languages are not usually included in the Chomsky hierarchy, but I find it useful to have the extra level. For instance, the **add012** language that I used to introduce the semantics section was a Type-4 language

BNF specifications generate Type-2 (context free) languages, and BNF can be thought of as an external syntax for Context Free Grammars

Derivation steps and context free derivations

For the remainder of this course, we shall assume that all of our formal grammars are Context Free unless otherwise noted

In formal grammar terminology, a single rewrite step is called a *derivation step* and is written $\Rightarrow$. A complete *derivation sequence* (or just a *derivation*) might be written

$$S \Rightarrow aXYc \Rightarrow aYc \Rightarrow abc$$

if we had context free productions $S \rightarrow aXYc, X \rightarrow \epsilon, Y \rightarrow b$

We use the symbol $\overset{*}{\Rightarrow}$ to represent a sequence of zero or more derivation steps, and $\overset{+}{\Rightarrow}$ to represent a sequence of one or more derivation steps. So, for instance, in the example above abc is in the language and hence $S \overset{*}{\Rightarrow} abc$. We can also say that

$$L(N, T, S, P) = \{u \mid S \overset{*}{\Rightarrow} u, u \in T^*\}$$

Derivation trees

If we represent a CF derivation step as a tree whose root is labeled with the left hand side nonterminal, and whose children are the right hand side elements, then the derivation sequence induces a tree, which for this example is $S(a, X(), Y(b), c)$

Note that the order in which we expand the X and Y nonterminals does not affect this tree, so the tree in fact represents the two derivation sequences

$$S \Rightarrow aXYc \Rightarrow aYc \Rightarrow abc(\text{expand X first})$$

$$S \Rightarrow aXYc \Rightarrow aXbc \Rightarrow abc(\text{expand Y first})$$

It turns out that for CF grammars, the rewrites are always confluent so we need never be concerned about the rewrite order, and the derivation tree thus captures all of the available information. Without loss of generality, we choose to only consider *leftmost* derivations in which the leftmost nonterminal is expanded at each derivation step.

Properties of derivation trees

Since all derivations must start with the start symbol, derivation trees will always have a root node labeled with the start symbol.

Terminals are symbols that cannot appear on the right hand side of a production, and so cannot have children in the derivation tree. Hence terminals in the derivation tree will appear as labels on leaf nodes.

Internal nodes of the tree correspond to nonterminals that will at some point in the derivation have been rewritten, hence nonterminals will appear as the labels of internal nodes in the tree.

Productions of the form $X \to \epsilon$ are represented by trees of the form $X(\epsilon)$

The *yield* of a derivation tree is the sequence of leaf node labels in left to right order. For any derivation of string u in grammar Γ, the yield of the corresponding derivation tree after deletion of ϵ instances is u

Unit productions, epsilon-free grammars and uselessness

A *unit production* is of the form $X \to Y$ with $X, Y \in N$.

An *epsilon free* grammar is one which has no rules of the form $X \to \epsilon$. There exists an ϵ-removal algorithm which will derive an epsilon-free Γ' from any grammar Γ with a new rule $S \to \epsilon$ needing to be added if $S \overset{*}{\Rightarrow} \epsilon$.

The grammar formalism does not preclude having elements in $X \in N$ for which there are no productions but such an X would be *useless*. Clearly any production $Y \to \alpha X \beta$ would also be useless in that any sentential form it generated would be stuck since there is no way to expand the instance of X into terminals.

A more subtle form of useless nonterminal is an X for which the only production is $X \to XX$. This will generate an infinite set of sentential forms but cannot generate any sentences. By extension, productions of the form $Y \to \alpha X \beta$ would also be useless.

We can see that it is possible to build entire codependent sets of productions which are useless. A grammar which has no useless rules in it is said to be *sanitised*

Finite and infinite languages

A nonterminal Z (and by extension its enclosing grammar) is *directly recursive* if the grammar includes a rule of the form $Z \rightarrow \alpha Z \beta$ and it is *recursive* if $Z \stackrel{*}{\Rightarrow} \gamma Z \delta$.

Z is *left recursive* if $\gamma \stackrel{*}{\Rightarrow} \epsilon$ and *right recursive* if $\delta \stackrel{*}{\Rightarrow} \epsilon$. A nonterminal Y *depends* on X iff the grammar $X \stackrel{*}{\Rightarrow} Y$.

If a grammar is not recursive, then a nonterminal can only appear at most once in any derivation. Since N is a finite set, that means that the language of a non-recursive grammar must be finite. Thus the only way we can specify infinite languages is with recursive nonterminals.

The grammar

$$S \rightarrow D \quad S \rightarrow D(A) \quad A \rightarrow S \quad A \rightarrow A, S \quad D \rightarrow a \quad D \rightarrow b \ldots$$

generates the infinite language of terms over single letter internal syntax names. The grammar is context free but *not* regular from which we see that languages with nested brackets cannot be specified using regular grammars (and thus not by regular expressions) – the slogan version is *regular expressions can't count*

Ambiguity and grammar equivalence

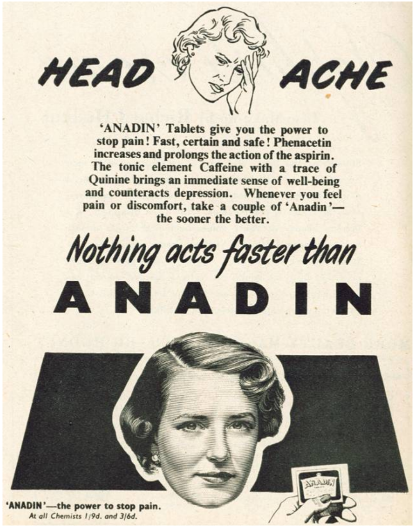
Two grammars Γ and Γ' are *equivalent* if $L(\Gamma) = L(Gamma')$

It is in general undecidable whether two context free grammars are equivalent. Hence asking for ‘the’ grammar for Java is incompetent. There are several grammars available, and we cannot be sure that they are equivalent. This is rather uncomfortable.

A grammar Γ is *ambiguous* is there exists some $u \in L(\Gamma)$ such that is more than one derivation sequence $S \overset{*}{\Rightarrow} u$, and thus two or more distinct derivation trees for u. This situation is common in real applications: the standard formulation of **if** - **then** - **else** is ambiguous.

It is in general undecidable whether a context free grammar is ambiguous. This is rather uncomfortable.

Semantics and derivation trees



'ANADIN' Tablets give you the power to stop pain! Fast, certain and safe! Phenacetin increases and prolongs the action of the aspirin. The tonic element Caffeine with a trace of Quinine brings an immediate sense of well-being and counteracts depression. Whenever you feel pain or discomfort, take a couple of 'Anadin'—the sooner the better.

'ANADIN'—the power to stop pain.
At all Chemists 1/9d. and 3/6d.

In both natural linguistics and formal language theory the *meaning* of a phrase is attached to the composition of the derivation tree. In English, many jokes rely on ambiguity (sometimes unintentionally as with the advert for a headache remedy here).

In formal languages (and especially the term based languages that we use as internal syntax) the *meaning of the program* is usually assumed to be a pre-order composition of the actions associated with each derivation tree node.

Generators, recognisers and parsers

A *generating procedure* for a context free grammar Γ is a procedure for enumerating $L(\Gamma)$. Since in general the language is infinite, it will not always terminate. Essentially we need to generate sentential foms, and then output the ones that have no nonterminals in them (the sentences). The order in which we generate sentential forms affects the result: if we go depth first then the exploration will be limited to sentential forms of leftmost recursive nonterminals. Hence we usually generate the sentential forms *breadth first*.

A *general context free recogniser* for Γ is a function $R(\Gamma, t \in T^*) : \mathit{boolean}$ which returns true iff $t \in L(\Gamma)$

A *general context free parser* for Γ is a function $R(\Gamma, t \in T^*) : \mathit{Set\ of\ Derivation\ Tree}$ which returns the (possibly infinite) set of derivations for t in $L(\Gamma)$.

A *parser* for Γ is a function $R(\Gamma, t \in T^*) : \mathit{DerivationTree}$ which returns one derivation tree for $t \in L'(\Gamma) \subseteq L(\Gamma)$. Note that this implies that some elements of $L(\Gamma)$ may not be recognised (that is no tree is returned for a valid member of the language) and that, further, for ambiguous elements $t \in L(\Gamma)$, at most one derivation tree will be returned. Very, very few production parsers used for real language processors are general. This is rather uncomfortable.

Alphabets, whitespace, and the benefits of lexicalisation

The *alphabet* of a grammar is simply the set T, that is the elements of the strings in $L(\Gamma)$. When we are looking at theoretical properties of grammars we do not usually pay much attention to the alphabet: the elements are just atomic mathematical objects. However in real languages, both natural and formal, it can be useful to *lexicalise* the character level strings into strings over *tokens* which have associated sets of lexemes, which are the actual textual substrings.

In free format programming languages such as Java it is conventional to split the input at any point where whitespace or a comment might be inserted. We then end up with lexemes such as `import`, `temp`, 123 and so on, and each class of lexemes is given its own token: so all integers are represented by a single token $\boxed{\text{int}}$, all identifiers by $\boxed{\text{id}}$ and so on. Keywords are usually (but not always) each given a unique token. This globbing together of substrings in the character input can significantly reduce the size of a context free grammar, and speeds up many of the internal processes of real parsers by allowing us to, for instance, immediately distinguish the Pascal keywords **do** and **downto** without needing to scan the input character string.

Lexicalisation also allows us to throw away white space and comments before parsing begins, reducing the size of the grammar and speedingup parsing.

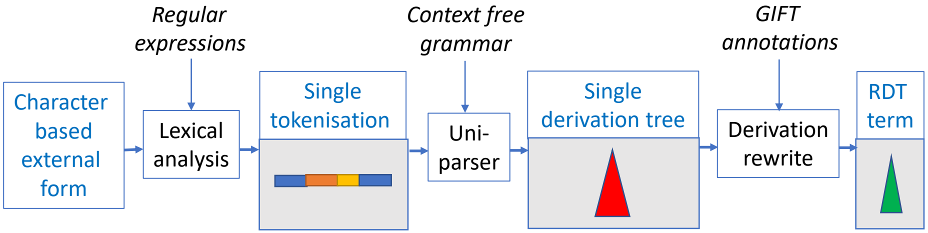
Why context free parsing?

Almost all production language systems perform non-general lexicalisation using regular specifications, and context free specification of non-general parsers. This is initially surprising since many features of real programming languages cannot be expressed using context free rules, and even the context free parts cannot all be handled by these ‘broken’ parsers.

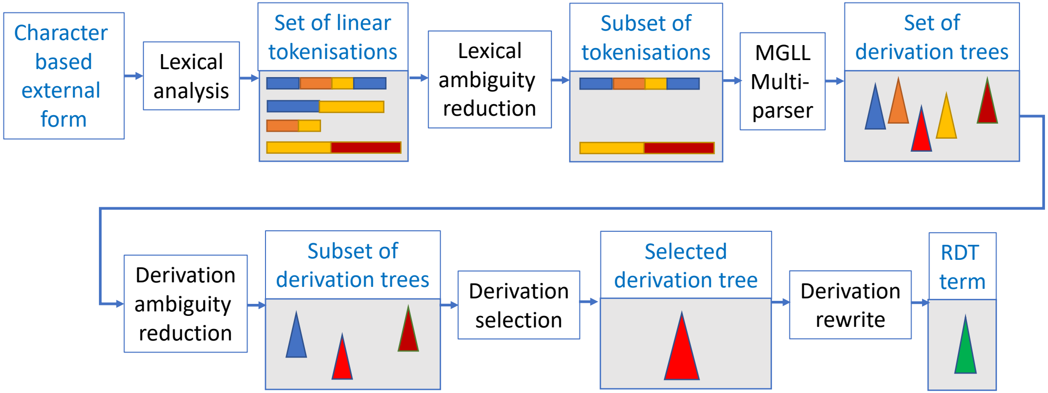
In particular, even simple type checking in languages with declarations cannot be specified using context free grammars. Just as ‘regular languages can’t count’, context free languages cannot check long-range properties of a string (the productions do not use *context*). So we cannot, within the grammar itself, discover that the Java fragment $\{$ String str; $\ldots$; str $=$ str $/$ 10.0;$\}$ is invalid (because we cannot divide a string by a real).

As we shall see later, this culture has grown our of the hardware limitations of the 1960s and 1970s-era computers that were current when compilers for languages such as Pascal and C were being developed. Today we can use polynomial-time and space general lexers and context free parsers. However we are unlikely to ever make use of context sensitive parsing because in general that requires exponential time. (And in addition, grammars turn out not to be a very comfortable notation for specifying type systems.)

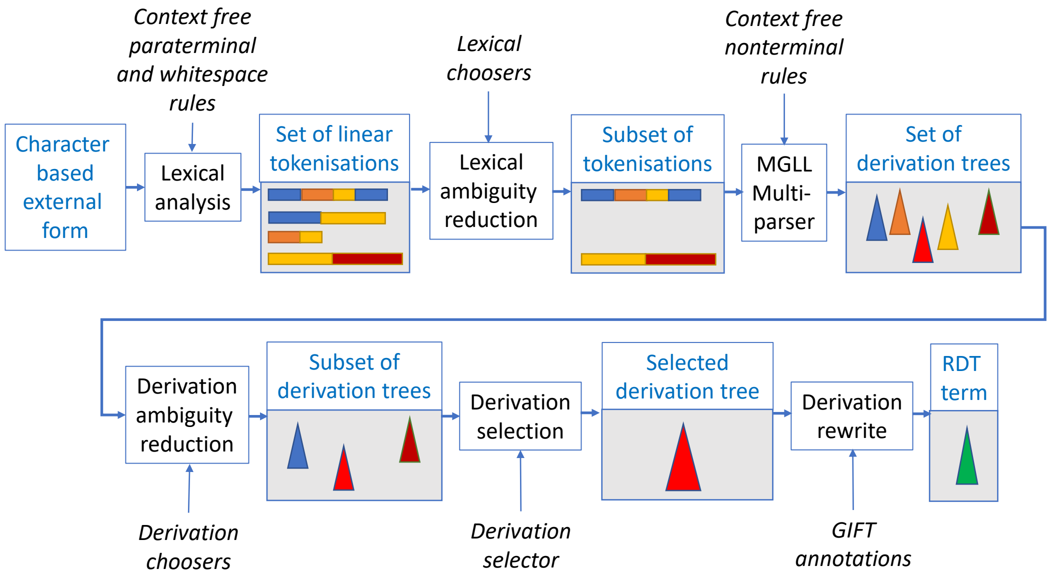
The traditional Lex/Yacc style external to internal syntax pipeline

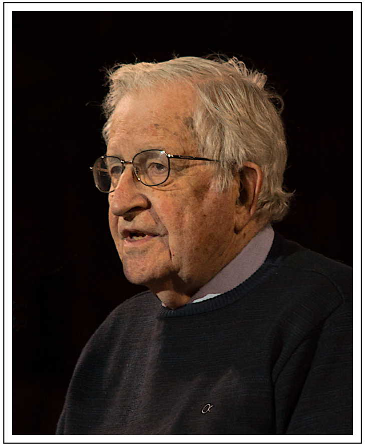


The ‘Holloway’ pipeline



The ‘Holloway’ pipeline with specification inputs





Noam Chomsky 1928–
https://en.wikipedia.org/wiki/Noam_Chomsky

4 B Limited but easy parsing: Ordered Singleton Backtrack Recursive Descent

This section illustrates the basic principles of parsing, semantics evaluation and term construction using the simplest useful parsing technique I could think of.

We shall also look at the limitations and failure modes of this technique.

This technique is called OSBRD – Ordered Singleton Backtrack Recursive Descent

Ordered – rules are checked in specification order (so valid inputs may be rejected)
Singleton – one match returned from each branch (so valid inputs may be rejected)
Backtrack – if a parser branch fails, then backtrack (with worst case exponential time)
Recursive Descent – a recursive nest of function calls (which may not terminate)

Given that OSBRD is broken in multiple ways, why learn about it?

The pedagogic benefits of OSBRD

OSBRD is not a serious contender for use as a production parser, but it has pedagogic benefits

1. Recursive Descent parsers may be written by hand – they are effectively just a specialised form of grammar pretty-printing. They are both easy to write and easy to understand
2. Recursive Descent parsers can be easily traced using a conventional code debugger, and that can be very helpful when trying to debug a grammar specification
3. Recursive descent parsers may be tamed: if a grammar is both left factored and follow determined then an RD parser will run in linear time and not reject valid strings. Grammars in this format are called LL(1). However the LL(1) format does not admit some useful grammatical idioms, in particular left recursion.
4. The refereed research literature on parsing is littered with over-strong claims. It is good to study broken algorithms as it will help you improve your critical reasoning skills.

So we shall use OSBRD to learn about *the internals of* parsing, syntax-directed translation and a limited form of attribute grammars before using fully engineered production parsing algorithms for the real work

The graph of leftmost derivation steps

The string rewriting relation that is induced by an unambiguous Context Free Grammar has a transition graph that is a tree. So, for instance, the grammar

$$\Gamma = S \to b \mid aXz \quad X \to xX \mid \epsilon$$

has language

$$L(\Gamma) = \{b, az, axz, axxz, axxxz, \ldots\}$$

and the following leftmost derivations:

$$S \Rightarrow b$$

$$S \Rightarrow aXz \Rightarrow az$$

$$S \Rightarrow aXz \Rightarrow axXz \Rightarrow axz$$

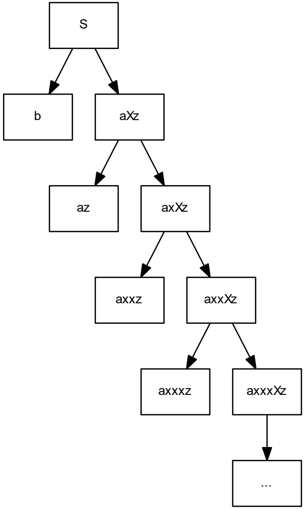
$$S \Rightarrow aXz \Rightarrow axXz \Rightarrow axxXz \Rightarrow axxz$$

$$S \Rightarrow aXz \Rightarrow axXz \Rightarrow axxXz \Rightarrow axxxXz \Rightarrow axxxz$$

$$S \Rightarrow axXz \Rightarrow axxXz \Rightarrow axxxXz \ldots$$

An unambiguous transition graph is a tree

The (leftmost) transition graph of the derivation relation is constructed by making a node for each unique sentential form, labelled with that sentential form. There is then an edge from the node labelled α to the node labelled β if $\alpha \overset{\text{left}}{\Rightarrow} \beta$



An ambiguous transition graph is not a tree

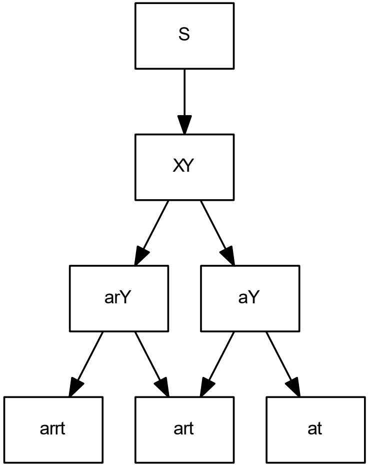
$$S \rightarrow XY \quad X \rightarrow ar \mid a \quad Y \rightarrow rt \mid t$$

$$S \Rightarrow XY \Rightarrow arY \Rightarrow arrt$$

$$S \Rightarrow XY \Rightarrow arY \Rightarrow art$$

$$S \Rightarrow XY \Rightarrow aY \Rightarrow art$$

$$S \Rightarrow XY \Rightarrow aY \Rightarrow at$$

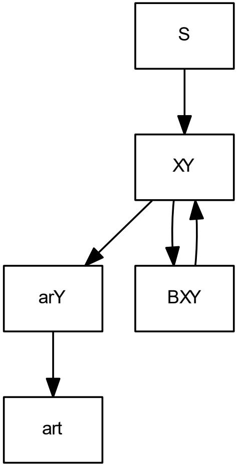


Unhygenic transition graphs are not even DAGs

$$S \to XY \quad X \to ar \mid BX \quad Y \to t \quad B \to \epsilon$$

$$S \Rightarrow XY \Rightarrow arY \Rightarrow art$$

$$S \Rightarrow XY \Rightarrow BXY \Rightarrow XY \Rightarrow \ldots$$



Parsing strategies

The *recognition* problem is: given a string σ, is there a path through the derivation transition graph that takes us from the node labelled with the start symbol to a node labelled with σ

The *parsing* problem is: given a string σ, for every path through the derivation transition graph the node labelled with the start symbol to a node labelled with σ, given the sequence of sentential forms linked by that path

There are two core operations we can use when trying to find derivations: so-called *top down* or *predictive* parsing where we select the (usually leftmost) nonterminal X in a sentential form, and then rewrite X using the i productions $X \rightarrow \alpha_i$. In general we therefore have a set of sentential forms that we are expanding at each step.

Alternatively we can look for production *right hand sides* within sentential forms, and replace them with their corresponding left hand side nonterminal. These *reductions* are making use of the *inverse* derivation relation: one in which all of the rewrite rules have their arrow reversed. *Bottom up* parsers work this way.

Our approach

So, top down parsers move forwards through the derivation relation graph, and bottom up parsers move backwards in the graph. In both cases the goal is the same (to find a path from the start symbol's node to the node containing the string we are parsing), and indeed mixed approaches (such as *left corner parsing* are also available.

We might also characterise algorithms in terms of how many sentential forms they are looking at. Deterministic parsers can be written that run in linear time (that is, $O(n)$ where n is the length of the input) but they usually only consider one sentential form at a time. General parsers typically maintain a set of parser configurations which are being processed together.

In this section, our OSBRD parsers are, at any given time, working on a single sentential form. However, if we reach a dead end, the parser can backtrack and try an alternative. Actually even out backtracking is inadequate, because OSBRD is *greedy* and will pursue the first successful production in a nonterminal that matches. More genrality is achieved if each nonterminal can return a set of production matches to pursue.

Two external syntaxes for BNF

Recall the set of conventions that we use when discussing the theory of conext free grammars: lower case letters $a \dots g$ denote terminals, lower case letters $u \dots z$ denote (possibly empty) strings of terminals, upper case letters denote nonterminals and Greek letters $\alpha \dots \delta$ denote strings over both terminals and nonterminals. The empty string (of length zero) is denoted by ϵ. These conventions are useful for writing small examples, compactly

For 'real world' examples, we need a different set of conventions that distinguish terminals and nonterminals using stropping, and which use only characters that appear on standard keyboards. we only need to specify rules, not reason about sentential forms, so we can drop the string conventions.

In ART, we use the following external syntax for context free productions:

nonterminals are alphanumeric identifiers such as `this`
case sensitive terminals are denoted by single quote delimited strings like `'this'`
the $\rightarrow$ 'produces' symbol is denoted by `::=`
the $|$ alternation symbol is denoted by `|`
the empty string is denoted by `#`

Thus we have two external syntaxes for the mathematical object 'Context Free Grammar'

Extended Backus Naur form – EBNF

ART does also provide the standard *postfix* regular expression operators and parentheses:

`X?` one or zero occurrences of `X`
`X+` one or more occurrences of `X`
`X*` zero or more occurrences of `X`
`(X | Y)*` zero or more occurrences of `X` or `Y`

These operators do not extend the set of languages that may be defined since in terms of the generated language, they are shorthands for BNF expressions:

```
X ::= Y?    ≡   X ::= Y | #
X ::= Y+    ≡   X1 ::= Y X1 | Y        X ::= X1
X ::= Y*    ≡   X1 ::= Y X1 | #        X ::= X1 (right recursive) or
X ::= Y*    ≡   X1 ::= X1 Y | #        X ::= X1 (left recursive)
X ::= (Y | Z)?   ≡   X1 ::= Y | Z        X ::= X1
```

where ≡ means *equivalent generated language*

Some EBNF subtleties

EBNF is widely supported in parser generator tools, but there are some subtleties to watch out for.

1. It is easy to accidentally introduce ambiguities. For instance, a well known language standard contains the idiom

   ```
   X ::= (Y | Z?)*
   ```

   Here, X generate the empty string in more than one way since `Z?` can generate ϵ and so can the star operator. And anyway, what does `#*` actually mean? Is it, in fact an error to apply an EBNF operator to the empty string?

2. What should derivation trees for EBNF look like? If our semantics is defined over derivation trees, we need a sound formal definition of the tree that appears from an EBNF parse

Lexical matters

Dividing the lexer and parser as described earlier reduces complexity in the parser, and enables automatic suppression of whitespace.

Formally, ART parsers use productions defined at character level. You can denote these explictly with ART productions using a *character level terminal* which is a **backquote** ‘ followed by a single printable character or an escape sequence.

Formally, then, the terminal `'adrian'` is shorthand the sequence

```
`a`d`r`i`a`n WS
```

where `WS` is the name of a nonterminal that defines the whitespace convention.

ART V3 also provides a set of *built in* lexical elements whose names are precede by an ampersand: `&ID`, `&INTEGER`, `&REAL`, `&STRING_DQ`, `&STRING_SQ` and others.

Some lexical subtleties

The partitioning of a grammar into lexical and phrase level productions is often convenient, but does present some theoretical challenges.

1. Traditional lexers require regular specifications, and this disallows nested bracketed comments
2. Traditional lexers provide only a single lexicalisation for each input, and in general this is not correct. Think about `a+++b` in Java, for instance
3. Traditional parsers would not know what to do with multiple lexicalisations even if they were available.
4. Sometimes one wants to parse mixed languages where more than one whitespace convention is in use. A global `WS` nonterminal is likely to be inadequate

ART V3 and your project work

We have developed ways to manage these EBNF and lexical difficulties, and the new ART V4 incorporates those ideas.

The eSOS interpreter and value system that you are using is part of ART V4.

However, for the *parsing* part of your project work in AY 2020-21, you will be using ART V3 which can run EBNF parses, but whose attribute evaluator is BNF only. As such, you need to stick to BNF specifications for your project work, and only use the builtin lexical elements.

Do not use EBNF constructs or character level terminals in your project work Do use BNF constructs and the builtin `&...` terminals in your project work

The OSBRD idea

OSBRD parsers come in three sections which run as separate phases:

1. A set of mutually recursive functions that read the input string to find a single derivation, and return it as a sequence of integers called the *oracle*
2. A set of mutually recursive functions that read the oracle to follow the trace of the successful derivation in step 1, executing embedded Java actions as a side effect
3. A set of mutually recursive functions that read the oracle to follow the trace of the successful derivation in step 1, and construct a *Rewritten Derivation Tree* under the control of the ^ and ^^ GIFT annotations.

Phase 1 tries to find the derivation using a backtracking search. Phase 2 allows *semantic actions* to be embedded within the parser. Phase 3 uses a special set of pre-written semantic actions to build trees suitablke for input to eSOS.

We need the oracle because there is no way to undo embedded semantic actions, so we need to find the derivation first (allowing backtracking) and then re-run the parser using the oracle to tell it 'magically' where to go.

OSBRD for pedagogy, MGLL for production

Although OSBRD is slow and in general incorrect, it can do much useful work, and the ideas and notations carry straight across to the production parsers in ART, such as MGLL.

The internals of OSBRD are easy to understand, and in fact sufficiently simple that we can write parsers for BNF specifications by hand. So we can explore the ideas around parsing and attribute grammars in this ‘sandbox’ and then use more sophisticated general algorithms in a black box manner.

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

```
Input: 'axx@ '
Accepted
Oracle: 2 1 1 2

Semantics phase

Tree construction phase

Derivation term
S('a',X('x',X('x',X(#))),'@'))

Derivation tree
9: S
1:   'a'
7:   X
2:     'x'
6:     X
3:       'x'
5:       X
4:         #
8:   '@'
```

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

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

  /* Nonterminal S, alternate 1 */
  cc = rc; co = ro; oracleSet(1);
  if (match("b")) {return true; }

  /* Nonterminal S, alternate 2 */
  cc = rc; co = ro; oracleSet(2);
  if (match("a")) {
  if (parse_X()) {
  if (match("@")) {return true; }}}

  return false;
}

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

  /* Nonterminal X, alternate 1 */
  cc = rc; co = ro; oracleSet(1);
  if (match("x")) {
  if (parse_X()) {return true; }}

  /* Nonterminal X, alternate 2 */
  cc = rc; co = ro; oracleSet(2);
  /* epsilon */ return true;
}
```

Informal parser construction rules

1. One function per nonterminal with no parameters and return type boolean
2. Global variables `cc` and `co` which index the current input character and current oracle output slot
3. Local variables `rc` and `ro` which remember the values of `cc` and `co` at entry
4. For each alternate production:
    - 4.1 Reset `cc` and `co` to their values at entry
    - 4.2 Write the *number* of the alternate into the oracle
    - 4.3 A *nest* of **if** statements for each alternate, one element per terminal and nonterminal, wrapping **return true;**
    - 4.4 For each element of the alternate production
        - 4.4.1 If a terminal, then call match
        - 4.4.2 If a nonterminal, then call the parse function for that nonterminal
        - 4.4.3 If ϵ (#) then **return true;**
5. If there is no epsilon alternative, at the end of the function **return false;**

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

```
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
```

Adding in semantic actions

There is a style of translation called *syntax directed* in which actions are executed as a side effect of the parse. Deterministic Recursive Descent parsers naturally lend themselves to this style since the actions can just be copied straight into the generated parsers.

Nondeterministic parsers (such as OSBRD, which backtracks) need to find a derivation before executing semantic actions. The Oracle is our mechanism for communicating between the parse and semantic phases.

In ART, actions in Java can be written between curly braces anywhere on the right hand side of a production:

```
S ::= 'b' | 'a' X '@'
X ::= 'x' { System.out.println("Found an x at location " + cc); } X | #
```

Syntax directed translation output

```
Input: 'axx@ '
Accepted
Oracle: 2 1 1 2

Semantics phase
Found an x at location 2
Found an x at location 3
```

The Oracle records the number of the alternate to be executed for each nonterminal.

So the Oracle $\boxed{\texttt{2 1 1 2}}$ says:
Use the second alternate for the start symbol (S)
For the leftmost nonterminal in the resulting expansion (X) use the first production
For the leftmost nonterminal in the resulting expansion (X) use the first production
For the leftmost nonterminal in the resulting expansion (X) use the second production

OSBRD semantic functions

```
class Attribute_S {}
void semantics_S(Attribute_S S) {
   switch(oracle[co++]) {
      case 1: {
         match("b");
         break; }

      case 2: {
         Attribute_X X1 = new Attribute_X();
         match("a");
         semantics_X(X1);
         match("@");
         break; }
   }
}

class Attribute_X {}
void semantics_X(Attribute_X X) {
   switch(oracle[co++]) {
      case 1: {
         Attribute_X X1 = new Attribute_X();
         match("x"); System.out.println("Found an x at location " + cc);
         semantics_X(X1);
         break; }

      case 2: {
         /* epsilon */
         break; }
   }
}
```

OSBRD main() and parse() functions

```
void parse(String filename) throws FileNotFoundException {
  input = readInput(filename);

  System.out.printf("Input: '%s'\n", input);
  cc = co = 0; builtIn_WHITESPACE();
  if (!(parse_S() && input.charAt(cc) == '\0')) {System.out.print("Rejected\n"); return; }

  System.out.print("Accepted\n");
  System.out.print("Oracle:"); for (int i = 0; i < co; i++) System.out.printf(" %d", oracle[i]); System.out.printf("\n");
  System.out.print("\nSemantics phase\n"); cc = 0; co = 0; builtIn_WHITESPACE(); Attribute_S S = new Attribute_S(); semantics_S(S);
  System.out.print("\nTree construction phase\n"); cc = 0; co = 0; builtIn_WHITESPACE();
  TreeNode dt = new TreeNode("S", tree_S(), null, TreeKind.NONTERMINAL, GIFTKind.NONE);
  dt.dot("dt.dot"); System.out.print("\nDerivation term\n"); dt.printTerm(0);
  System.out.print("\n\nDerivation tree\n"); dt.printTree(0);

  ...

}

public static void main(String[] args) throws FileNotFoundException{
    if (args.length < 1) {
      System.err.println("No input file name supplied");
      System.exit(1);
    } else
      new ARTGeneratedParser().parse(args[0]);
  }
```

Using attributes to transmit information within the tree

Adding a print statement to a rule is a rather limited form of side-effect: in general we want to be able to perform computations over data drawn from several rules.

The mechanism by which we move information around is the *attribute*. We are allowed to attach arbitrary data elements to trees in the node

Now, the internal (non-leaf) nodes of a derivation tree are created by instances of a nonterminal. We specify the data to be attached to these nodes within angle brackets, and we can then manipulate those elements in Java actions@

```
X <a:int b:double> ::= 'x' X X { X.a = X1.b + X2.b; }
```

The instances of nonterminals are numbered across the rule so that we can distinguish them; in Java we say `X.a` for the left hand side attribute, `X1.a` for the first right hand side X's attribute, and so on. We are not allowed to look further into the tree - all data accesses are local in this sense.

The leaf nodes are created by instances of terminals. In our ART system, leaf nodes have automatically generated attributes that allow us to retrieve the substring of the input that was matched by this token.

An attributed example

Let us now write an attributed grammar that does not just report the appearance of an `x`, but counts the number of `x` characters in a string:

```
S ::= 'b' | 'a' X '@'  { System.out.println("Xcount: " + X1.x); }

X < x:int> ::= 'x' X { X.x = X1.x+1; }
             |  #    { X.x = 0; }
```

```
Input: 'axx@ '
Accepted
Oracle: 2 1 1 2

Semantics phase
Xcount: 2
```

Implementing attributes in OSBRD

Each nonterminal has its own unique set of attributes. In the generated OSBRD parser, for each nontermina X we create a class called `Attribute_X` Java declarations corresponding to the attribute definitions in angle brackets

On entry to a parse function for Y, we create separarate instances of the attribute classes for every nonterminal that appears on a right hand side of Y.

We can use actions to put values into these attributes *before* calling the associated parse function - this allows a parent node to pass information down the tree. Such attributes are called *inherited* attributes in the literature.

We can also put new values into the attribute block that was passed to us, thus allowing information to flow back to our parent. Such attributes are called *synthesized* attributes in the literature.

```
class Attribute_S {}

void semantics_S(Attribute_S S) {
   switch(oracle[co++]) {
      case 1: {
         match("b");
         break; }

      case 2: {
         Attribute_X X1 = new Attribute_X();
         match("a");
         // Instance X1
         semantics_X(X1);
         match("@"); System.out.println("Xcount: " + X1.x);
         break; }
   }
}

class Attribute_X { int x; }

void semantics_X(Attribute_X X) {
   switch(oracle[co++]) {
      case 1: {
         Attribute_X X1 = new Attribute_X();
         match("x");
         // Instance X1
         semantics_X(X1); X.x = X1.x+1;
         break; }

      case 2: {
         /* epsilon */ X.x = 0;
         break; }
   }
}
```

Tree construction as specialised semantics

In OSBRD grammars, the attributes and actions allow us to directly implement interpreters for languages up to, but not including, flow control involving loops since the OSBRD semantics functions only visit each tree node once.

(We shall handle using *delayed attributes* in a more powerful attribute evaluator to be discussed in Section 6).

One important use case for these interpreters is the creation of terms for interpretation *via* SOS rules (and other styles of formal semantics). In the research literature these are usually called *abstract syntax trees*: the idea being that anything a human writes is a *concrete* syntax, and the internal forms are all *abstractions* of the human-written text.

I prefer to use the terms external syntax and internal syntax in recognition of the multi-layered nature of many real systems, since the internal syntax of an outer layer is used as the external syntax of the next layer down. We shall have more to say about this in section 5.

Representing trees

We have already studied a purpose-built library for efficiently representing trees in the form of ART's `ITerms` class hierarchy, and of course we could use that with OSBRD generated parsers. However, we want OSBRD to be stand-alone and simple to understand, so instead we offer a simple, traditional tree class which uses the child/sibling model.

Each tree node has a `String label` which is the textual representation of the tree node's label. In addition, for pedagogic and debugging purposes, we include an integer field `nodeNumber` which holds a unique integer. This allows us to trace through trees more easily since we can distinguish between multiple instances of the same nonterminal or terminal.

The tree nodes also have a type field to distinguish between epsilon-nodes, terminals, nonterminals and builtins (since a terminal, nonterminal and builtin could all have the same text label) and a *GIFT* annotation of which more later.

```
public enum TreeKind {
    EPSILON, TERMINAL, BUILTIN, NONTERMINAL
};

public enum GIFTKind {
    NONE, FOLD_UNDER, FOLD_OVER, FOLD_ABOVE
};

public class TreeNode {
    String label;
    int nodeNumber;
    TreeNode child;
    TreeNode sibling;
    TreeKind kind;
    GIFTKind giftOp;

...
```

Tree construction and rendering

In OSBRD, trees are created bottom-up using a postorder traversal of the derivation, and so our tree nodes can be created via the usual constructor which is passed child and sibling values.

```
public TreeNode(String label, TreeNode child, TreeNode previousSibling, TreeKind kind, GIFTKind giftOp) {
        if (previousSibling != null) previousSibling.sibling = this;
        this.label = label;
        this.child = child;
        this.sibling = null;
        this.kind = kind;
        this.giftOp = giftOp;
        nodeNumber = nextNode++;
    };
```

The `TreeNode` class also contains code to render trees as `.dot` files for visual display, and for two kinds of textual rendering of these trees: as an indented listing, and as a parenthesized term suitable for input to eSOS.

```
public void printTree(int indent) {
  System.out.printf("%d: ", nodeNumber);
  for (int temp = 0; temp < indent; temp++)
    System.out.printf(" ");
  System.out.printf("%s%s%s", labelPreString(kind), label, labelPostString(kind));
  System.out.printf("%s\n", giftString(giftOp));

  if (child != null) child.printTree(indent + 1);
  if (sibling != null) sibling.printTree(indent);
};

public void printTerm(int indent) {
  System.out.printf("%s%s%s", labelPreString(kind), label, labelPostString(kind));

 if (child != null) { System.out.print("("); child.printTerm(++indent);
 }

 if (sibling != null) { System.out.print(","); sibling.printTerm(indent);
  } else
    System.out.print(")");
};
```

Tree outputs

We shall demonstrate the tree outputs with this example:

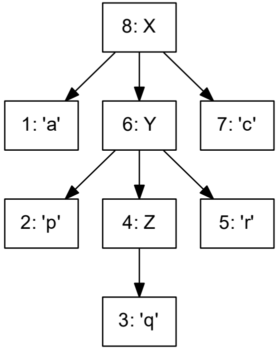
$$X ::= aYc$$

$$Y ::= pZr$$

$$Z ::= q$$

The language has only one string apqrc with this derivation:

$$X \Rightarrow aYc \Rightarrow apZrc \Rightarrow apqrc$$



```
> parse treeTest

Input: 'apqrc '
Accepted
Oracle: 1 1 1

Semantics phase

Tree construction phase

Derivation term
X('a',Y('p',Z('q'),'r'),'c'))

Derivation tree
8: X
1:   'a'
6:   Y
2:     'p'
4:     Z
3:       'q'
5:     'r'
7:   'c'
```

Generated tree functions

```
  TreeNode tree_X() {
    TreeNode leftNode = null, rightNode = null;
    switch(oracle[co++]) {
        case 1:
            leftNode = rightNode = new TreeNode("a", null, rightNode, TreeKind.TERMINAL, GIFTKind.NONE);
            match("a");
            rightNode = new TreeNode("Y", tree_Y(), rightNode, TreeKind.NONTERMINAL, GIFTKind.NONE);
            rightNode = new TreeNode("c", null, rightNode, TreeKind.TERMINAL, GIFTKind.NONE);
            match("c");
        break;
    }
  return leftNode;
}
  TreeNode tree_Z() {
    TreeNode leftNode = null, rightNode = null;
    switch(oracle[co++]) {
        case 1:
            leftNode = rightNode = new TreeNode("q", null, rightNode, TreeKind.TERMINAL, GIFTKind.NONE);
            match("q");
        break;
    }
  return leftNode;
}
```

Skipping tree nodes with promotion operators

Sometimes it is useful to omit tree nodes during construction. For instance, consider the Java **if-then-else** statement. In Java (and C) the **if** keyword is *always* followed by an opening parenthesis. From the point of view of the semantics, it is entirely redundant.

In fact, bracketing conventions of all kinds are redundant in the tree representation since we can use the parent-child structure of the tree to directly represent the nesting of all kinds of constructs, from expression operator priorities and associativities, through control flow statement nesting and local classes.

We can also *rename* language constructs by creating dummy nonterminal names and dropping keywords. So we might write a rule like

```
if ::= 'if' '(' predicate ')' statement ';'
```

but want the elements in red to be omitted from the tree so that we and up with the term `if(predicate,statement)`

The ^ and ^^ fold operators

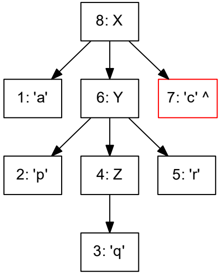
ART provides a set of *tree annotations* called GIFT operators (which stands for Gather-Insert-Fold-Tear). In this section we only consider two main the *Fold* annotations: **fold-under** `^` and **fold-over** `^^`.

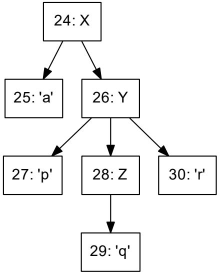
Conceptually, the fold operators move the annotated node up one level in the tree, folding on top of or underneath their parent. The annotated node's children are dragged up with it.

Fold under ˆ on a terminal

A ˆ annotation applied to a terminal simply deletes it (it is folded under its parent, and has no children)

```
X ::= 'a' Y 'c'^
Y ::= 'p' Z 'r'
Z ::= 'q'
```

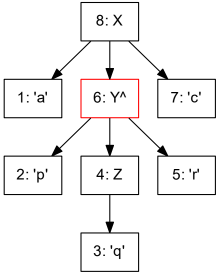


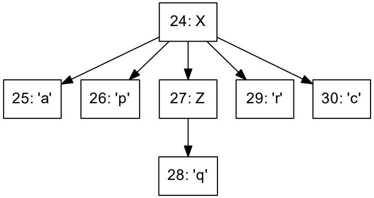


Fold under ^ on a nonterminal

A ^ annotation applied to a nonterminal pulls that node up under its parent. The children come with it, and are inserted into the children of the parent:

```
X ::= 'a' Y^ 'c'
Y ::= 'p' Z 'r'
Z ::= 'q'
```



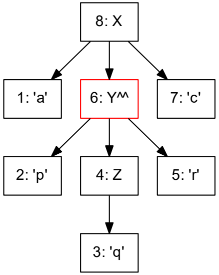


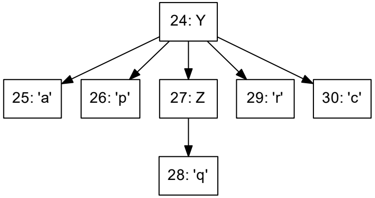
Fold over ^^ on a nonterminal

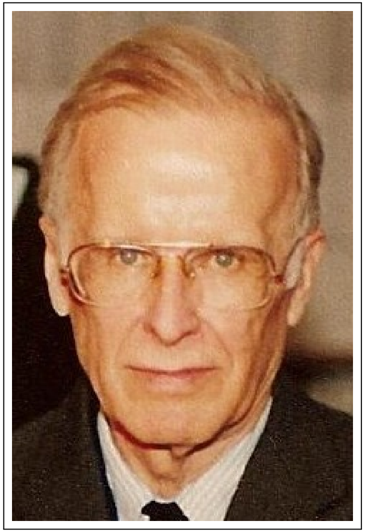
A ^^ annotation applied to a nonterminal pulls that node up *over* its parent, effectively

```
X ::= 'a' Y^^ 'c'
Y ::= 'p' Z 'r'
Z ::= 'q'
```

relabelling the parent node. As before, the children come with it:







John Backus 1924–2007
https://en.wikipedia.org/wiki/John_Backus

4C From strings to tokens - lexical analysis

Peter Naur 1928–2016

https://en.wikipedia.org/wiki/Peter_Naur

4D The space of parsing algorithms

Donald Knuth 1938–
https://en.wikipedia.org/wiki/Donald_Knuth

4E Generalised Context Free parsing

Masaru Tomita 1957 –
https://en.wikipedia.org/wiki/Masaru_Tomita

4F Generalised regular and context free lexical analysis

Adrian Johnstone and Elizabeth Scott

5 Grammars, trees and attributes

Now we have the formalisms, how do we work with them?

You now have a working knowledge of parsing, action execution, derivation tree rework for internal syntax generation and SOS interpretation. Just as with any form of programming, once you have understood the building blocks you still need to develop idioms and design patterns to generate the effects that you want.

We shall now look at a sequence of small languages in SLELabs/AttributeEvaluation and develop grammar styles to handle common programming language features.

Section	Topic
5A	Motivation
5B	Grammar idioms for expressions
5C	Delayed attributes and control flow
5D	Control flow abstraction
5E	Formal aspects of attribute grammars
5F	A synthesis of attribute grammars and term rewriting

5A Motivation

We have two main routes to language interpretation: the genertaion of prefix internal forms suitable for eSOS interpretation, and the direct implementation of interpreters using *syntax directed translation* in which actions are executed as a side effect of the parse.

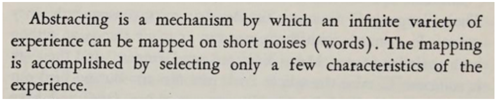
These two are not necessarilly in tension with each other: for instance we might use a syntax directed translation to construct the internal form (and in fact that is effectively what we shall do, but the actions are disguised as GIFT annotations).

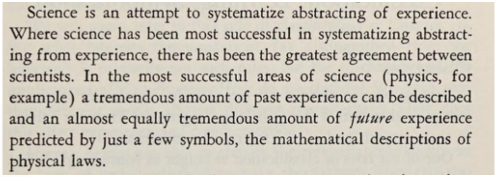
So both techniques are important. I think it is fair to say, though, that most little languages are implemented as syntax directed translations, possibly to some three-address code-like internal form that may be easily executed by a simple simulator.

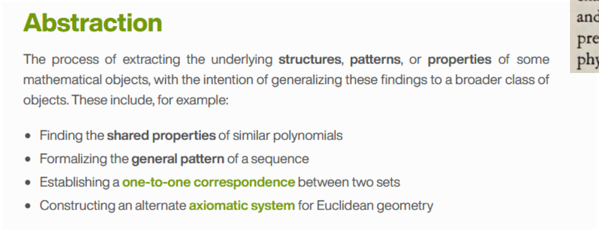
The common thread is *abstraction* or, loosely speaking, knowing what to throw away.

Abstraction in linguistics, science and mathematics

Abstraction is central to what we do in computing – essentially the process of finding clean, general models of things that we aid implementation and often describe real applications as a special case of some more general pattern or idea. The idea of abstraction as a powerful intellectual tool emerges from linguistics and mathematics:


Abstracting is a mechanism by which an infinite variety of experience can be mapped on short noises (words). The mapping is accomplished by selecting only a few characteristics of the experience.


Science is an attempt to systematize abstracting of experience. Where science has been most successful in systematizing abstracting from experience, there has been the greatest agreement between scientists. In the most successful areas of science (physics, for example) a tremendous amount of past experience can be described and an almost equally tremendous amount of *future* experience predicted by just a few symbols, the mathematical descriptions of physical laws.


Abstraction

The process of extracting the underlying **structures**, **patterns**, or **properties** of some mathematical objects, with the intention of generalizing these findings to a broader class of objects. These include, for example:

- Finding the **shared properties** of similar polynomials
- Formalizing the **general pattern** of a sequence
- Establishing a **one-to-one correspondence** between two sets
- Constructing an alternate **axiomatic system** for Euclidean geometry

Quotations from: Anatol Rappaport *Science and the goals of man* (1950) and from Maths Vault at `https://mathvault.ca/math-glossary`. See also `https://en.wikipedia.org/wiki/Abstraction`

Abstraction in programming

Early programming languages dealt very much at the level of individual machine operations and individual machine locations. The *subroutine* (first really identified by Maurice Wilkes) was initially thought of as a way of reducing teh size of prigrams.

Consider an integer only machine and a program that needs to perform floating point addition. We could insert a sheaf of instructions to perform the addition at every point where FP addition was called for (this is called a *macro*. The subroutine idea was to call a separate piece of library code which resided at one location in memory, and since the call and return sequence was small compared to the FP addition syntax, the program size was reduced.

What began as a technical trick was soon recognised as an excellent way of organising the programmer's thoughts. The structured programming debates of the 1970s, and the Design Patterns movement are examples of *procedural abstraction*. We recognise *data abstraction* in our use of (amongst other things) parameterised collection data types such as `HashMap<>` in Java. Continuations are *control flow abstractions*.

The idea of abstract syntax

Donald Knuth attributes the term *abstract syntax* to John McCarthy, the inventor of LISP. The LISP style as we know it today was originally intended to be just the internal syntax for another, more human-friendly language. So LISP's intended internal syntax ended up as its external syntax.

As an aside, the use of a single internal representation for both data and code in LISP is intensely liberating, and allows all kinds of idioms that are hard – and often impossible – to express in conventional languages. Learn Scheme if you want your mind expanded.

So LISP shows that separating out the notions of human-friendly concrete syntax and internal abstract syntax is rather fraught.

I don't use the terms myself because real systems are often multilayer, in which the internal syntax of an outer layer is the external syntax of the next layer down. And as one progresses towards code generation for a real architecture, the layers tend to get *more* concrete. Nevertheless, the term Abstract Syntax Tree is ubiquitous - when you see it just think *Internal Syntax Tree/Term*.

Abstraction in syntax

Now, just because I don't like the term *abstract syntax* please don't get the idea that I think we aren't abstracting...

In this context (internal syntactic forms) *abstraction* usually involves one of two transformations

1. *Discarding* syntactic elements that do not change the semantics of the internal syntax
2. *Normalising* syntactic forms so that external phrases with similar meanings are mapped to the same (possibly more general) internal form.

Occasionally, especially for normalisation, we want to reorder elements, or insert pieces of tree that do not directly arise from the concrete syntax.

Occasionally, especially for normalisation, we also want to *gather* together children of disparate nodes into a single new node.

Discarding nesting tokens

Humans like to write text linearly, but ever since Algol-60 introduced the notion of block structure, programming languages have been strongly *compositional* making extensive use of nesting to build complex behaviour from small building blocks.

In Java, the { and } braces delimit blocks of code that we think of as compisite statements. Each control structure takes a single statement (which may be a composite). In other languages, explicit bracketing is used instead if the composite statement, with structures such as **if-then-if** or **if-then-endif**

In Java, classes can nest. In Pascal like languages, procedures can nest. In nearly all languages, operator expressions can nest.

All of these nesting ideas need delimiter tokens in linear text, but can be represented directly by parent-child relationships in the tree. So we do not need the various kinds of brackets that we see in Java. However, getting the right *shape* of tree requires care.

Rear Admiral Grace Hopper
https://en.wikipedia.org/wiki/Grace_Hopper

5B Grammar idioms for infix expressions

Early pocket calculators executed arithmetic operations in strictly left to right order. This was straightforward from an implementor's point of view, but ignores the arithmetic conventions that we learn in school.

These conventions can be viewed as a way of *putting in* parentheses that a user has omitted. So, for instance, we typically use these *priority levels*

- parentheses (do-first) [High]
- exponentiate
- multiply, divide, remainder
- add, subtract [Low]

So the, by convention, the meaning of $x + y \times z$ is $x + (y \times z)$ **not** $(x + 7) \times z$

We could avoid this ambiguity by insisting that all operations are parenthesized, or by using Reverse Polish Notation. The difficulty arises from our love of infix notation.

Commutativity

Some operations are mathematically commutative, which means that we get the same result if we swap the operands around. So for instance,

$$3 + 4 = 7 = 4 + 3$$

but

$$3 - 4 = -1 \neq 4 - 3$$

Hence, in mathematics, addition of integers is commutative but subtraction is not.

In computing, even addition is not commutative, because of the finite representation of numbers. So, for instance, for eight bit unsigned addition, $254 + 1 = 255$ but $255 + 1 = 0$.

Forgetting to take this irregularity of computer arithmetic into account has led to some expensive failures (eg Ariane 501: `https://www.youtube.com/watch?v=PK_yguLapgA`

Associativity

In programming languages, we disambiguate compound expressions over operators of equal priority by specifying that operators are left-, right- or non associative. (Note that this is different to the mathematical notion of associativity in which an operator is associative if the order of parentheses does not matter.)

In most programming languages, relation operations are not associative. For instance $1 \leq x \leq 6$ is aperfectly reasonable thematically expression which constrains x to the interval 1..6. However, in Java `1 <= x` returns a boolean value which cannot be used in a further relational operation. We call such operators *non-associative*.

By convention $x - y - z$ means $(z - y) - z$. We sometimes say that subtraction $-$ *binds more tightly to the left* or is *left associative*.

Exponentiation in arithmetic is conventionally right associative: x^{y^z} means $x^{(y^z)}$

When treated as an operator, assignment must also be right associative since `x = y = z = 3` must be processed right to left.

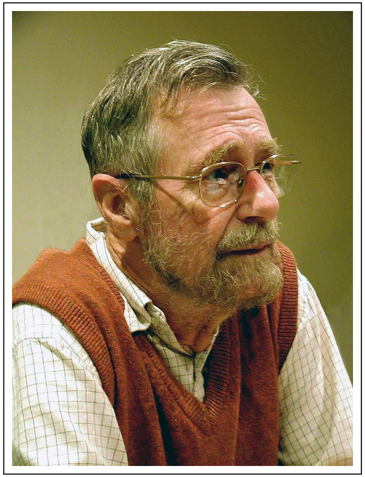
Syntax	Semantics	Precedence	Associativity
`[ ]`	Array element select	1	Left
`( )`	Method call	1	Left
`.`	Class member select	1	Left
`+`	Unary posite	2	Right
`-`	Unary negate	2	Right
`!`	Unary logical complement	2	Right
`~`	Unary bitwise complement	2	Right
`++`	Unary postfix or prefix complement	2	Right
`--`	Unary postfix or prefix complement	2	Right
`(type)`	Type cast	2	Right
`new`	Create new object	2	Right
`*`	Multiply	3	Left
`/`	Divide	3	Left
`%`	Remainder after division	3	Left
`+`	Add	4	Left
`-`	Subtract	4	Left
`+`	String concatenation	4	Left

Syntax	Semantics	Precedence	Associativity
`<<`	Signed left shift	5	Left
`>>`	Signed right shift	5	Left
`>>>`	Unsigned right shift	5	Left
`<`	Less than	6	Non
`<=`	Less than or equal	6	Non
`>`	Greater than	6	Non
`>=`	Greater than or equal	6	Non
`instanceof`	Reference is an instance of	6	Non
`==`	Equal to	7	Non
`!=`	Not equal to	7	Non
`&`	Bitwise AND	8	Left
`&`	Logical AND	8	Left
`^`	Bitwise XOR	9	Left
`^`	Logical XOR	9	Left
`|`	Bitwise OR	10	Left
`|`	Logical OR	10	Left
`&&`	Logical short circuit AND	11	Left
`||`	Logical short circuit OR	12	Left

? :	Conditional	13	Right
=	Assignment	14	Right
+*=	Operation and assign	14	Right
−=	Operation and assign	14	Right
*=	Operation and assign	14	Right
/=	Operation and assign	14	Right
%=	Operation and assign	14	Right
<<=	Operation and assign	14	Right
>>=	Operation and assign	14	Right
>>>=	Operation and assign	14	Right
&=	Operation and assign	14	Right
^=	Operation and assign	14	Right
|=	Operation and assign	14	Right

Niklaus Wirth
https://en.wikipedia.org/wiki/Niklaus_Wirth

5C Delayed attributes and control flow



Edsger W. Dijkstra

https://en.wikipedia.org/wiki/Edsger_W._Dijkstra

5D Control flow abstraction

Sir Maurice Wilkes
https://en.wikipedia.org/wiki/Maurice_Wilkes

5E Formal aspects of attribute grammars

Donald Knuth on writing
https://www.youtube.com/watch?v=vGOD-kKTF1g

5F A synthesis of attribute grammars and term rewriting

6 Pragmatics and case studies

Section	Topic
6A	The architecture of language processors
6B	The landscape of programming languages
6C	The Zoo of implementation technologies
6D	The tar pit of complexity
6E	Ambition, management and maintenance
6F	The future

Further reading on language design and implementation

These books are useful compendia of topics in language design and compilation:

Programming Language Pragmatics
Michael L Scott, fourth edition, Morgan Kaufman, 2016

Modern compiler design
Dick Grune et al, Second edition, Wiley, 2016

Programming language design concepts
David A. Watt, Wiley, 2010

Design concepts in programming languages
Franklyn Turback and David Gifford, MIT Press, 2008

A now very dated, but in my opinion enjoyable and motivating book is:

The programming language landscape
Henry Ledgard and Michael Marcotty, Science Research Associates, 1981

Further reading on machine level topics and software management

These two books are the standard texts on computer architecture and systems software:

Computer architecture: a quantitative approach
John Hennessy and David Patterson, sixth edition, Morgan Kaufmann,2017

Computer organisation and design
John Hennessy and David Patterson, sixth edition, Morgan Kaufmann,2020

Anybody who ever has to be a member of a software development should read this:

The mythical man-month
Fred Brooks, anniversary edition, Addison Wesley, 1995

6A The architecture of language processors

We can identify four main parts of a language system

1. The *front end* (F) translates from external syntax to internal syntax
2. The *back end* (B) translates from external syntax to target syntax
3. The *middle end* (M) translates from internal syntax to internal syntax, transforming the program in some way
4. The *execution end* (X) executes the program, consuming data and generating the programs effects

There are at least three representations of the user program in use here: external (E), internal (I) and target (T). In the case of `javac`, the Java compiler, syntax E is the Java language itself, syntax T is Java byte code and syntax I is a property of the particular java compiler implementation. Some compiler toolkits, for instance the gcc toolchain, have multiple internal syntaxes, each optimised for a particular task.

It’s languages all the way along

We can visualise these four parts as a *translation pipeline*, each stage of which translates the user’s initial program, perhaps producing some outputs as a side effect.

Syntax	Stage	Side outputs
External	*User program*	
	Front end	Syntax check
Internal	*Internal representation*	
	Middle end	Call graph, cross refeference, lint and verify
Internal	*Reworked internal representation*	
	Back end	Source-to-source transformations
Target	*Translated program*	
	Execution end	Profile, coverage, program effects

In detail, the middle end often involved multiple phases, each of which is an I-to-I translation

Why four stages?

The four stage process arises naturally from a modularisation based on dependencies.

The front end is dependent only on the external syntax and outputs the internal syntax
The middle end is independent of external and target syntax
The back end is dependent only on the target syntax and the internal syntax
The execution end is dependent only on the target syntax

This separation of concerns allows external and target independent middle ends to be developed, which can then be connected to multiple front and back ends. So the GNU compiler collection includes front ends for C, C++, Go, Pascal, Modula-2, FORTRAN, Ada and others. GNU has supported over 50 different machine architectures over its development history.

Detailed code analysis and rearrangement so as to improve speed or compactness of the generated code resides in the middle end, so having a machine and programming language independent middle end allows reuse of significant intellectual property.

Do we always need four stages? The single pass compiler

In some use cases we do not need all four stages

It is perfectly possible (for certain restricted classes of programming language) to emit machine code for a real computer architecture as a side effect of the parsing process, with no discernible middle- or back-end. In fact nearly all languages designed before the mid-1990s support this so-called *single pass compiler* architecture.

The defining feature of 'single pass friendly' languages is that variables and procedures must be declared before use. ANSI-C and Pascal are good examples of this phenomenon. It cause particular problems with recursive functions, whose definitions cannot be completed before they are used. To handle such cases, both languages allow a signature to be declared without a corresponding function body.

Such compilers usually generate rather inefficient executable code, but they can be run on very small memory machines with only a few thousand memory locations. They can also be very fast - the 1980s Turbo Pascal compiler wasa very popular development environment for IBM PCs.

Do we always need four stages? The assembler

Until highly portable languages such as Java appeared in the mid 1990s, most working software engineers would have had the skills to drop down into machine language if they wanted to speed up a critical piece of code. In our terms, this means writing programs directly in syntax T.

To a hardware engineer, syntax T comprises bit strings, and humans are not very good at distinguishing bit strings by eye, so very early on in the development of computers (around 1947 and 1948 in Birkbeck College London and at the Mathematical Laboratory in Cambridge) a programming style developed in which short mnemonics such as ADD and JMP were defined for each discrete machine instruction, along with a facility to associate an alphanumeric name with a machine address. This allowed programmers to write, for instance, `ADD X,Y,Z` which could easily be turned into a bit string by looking up the bit-string values of A, X, Y and Z, and then concatenating them together into a binary machine word. The style of programming language is called an *assembler*; and the reverse lookup provides a *dissassembly* when examining machine instructions.

Nearly all execution environments provide some sort of assembly and dissassembly. The `javap` command dissasembles JVM byte code from .class files.

Do we always need four stages? The interpreter

Programs in languages other than assembly language first began to emerge in the mid-1950s. So-called *autocodes* were developed for the Manchester Mk 1 and later Ferranti machines also associated with Manchester University and in Cambridge for EDSAC. These systems typically allowed individual expressions to be compiled into subroutines, and were the beginnings of portable programs that could run on more than one machine architecture.

Language development then took two paths for two user communities. The FORTRAN language (1957) aimed to produce code which was fast enough to persuade assembly language programmers to learn the new technique.

Universities were interested in teaching programming to engineers and scientists, and needed simple languages that could run on low performance systems with instant feedback to the user. BASIC (Beginners All Purpose Symbolic Instruction Code) (1964) was designed to be executed line-by-line in an interactive environment. No syntax T program is produced: instead the middle end has a routine for each language feature which is activated as each line is parsed: this style is called an *interpreter*.

Do we need more than four stages? The Just-In-Time compiler

One way to achieve high portability but still allow significant code improvements to be applied to the user's program is to compile to a *virtual machine*. The UCSD P-system (1977) was a response to the plethora of new microprocessor architectures being developed at that time. By compiling to *P-code* and then having an efficient interpreter which could be easily implemented in, say, assembly language one could quickly get Pascal running on a new architecture.

The same approach was taken by the designers of Java (1995) which was originally intended for embedded systems in which code would be compiled on a general purpose machine, but run on processors with very restricted resources.

Early Java systems were very slow compared to C code compiled to native instructions. Modern systems are often within a small integer factor of the performance of C due to the development of *Just-In-Time* techniques in which the interpreter monitors *hot spots* that are frequently executed, and can stop and compile those to native instructions. Note that this is a T to T' translation, where T is JVM byte code and T' is the native instruction set of the host architecture.

Tooling use cases

1. Syntax checker (F)
2. Cross referencer, call graph generator, linter, ‘bad smell’ detector (FM)
3. Code verifier against some high level specification (FM)
4. Pretty printer, minifier, normaliser (FMB, but with syntax T=E)
5. Test case generator (FMB, but with syntax T=E)
6. Live editor with immediate feedback to the programmer (all of the above, running in the background of an editor)
7. Interpreter (FMX)
8. Single pass compiler (FX)
9. Optimising compiler (FMBX)
10. Just-in-time compiler (FMBX(B’X)*)
11. Reverse compiler (FMB, but with syntax E being low level (eg Java byte code) and syntax T being high level (eg Java))
12. Profiler (FMBX)

6B The landscape of programming languages

Most languages evolve from earlier attempts, but occasionally radical new ideas emerge. FORTRAN was the first complete programming language that could run on more than one architecture. COBOL was the first attempt to converge programming language syntax with non-scientific prose (SUBTRACT A FROM B GIVING C). LISP (1958) was the first functional language (although LISP as we know it was intended only to be the internal syntax for a more baroque external syntax!)

Algol was the first widely used language to allow user defined types, recursion and block structured control flow. Simula-67 introduced object orientation. Smalltalk took the object idea to the limit - everything in Smalltalk-80 is an object including control flow structures.

SETL (1969) introduced *comprehensions* which now appear in many functional languages, and more recently Python. ML (1973)and its precursors introduced the notions of *type inference*, *type variables*, algebraic types and pattern matching. Rust (2015) provides a type system that guarantees certain safety properties.

How not to design a successful language

Notions are more important than notations. Notation matters though!

When programmers start thinking about a new language there is a strong tendency to focus on external syntax by finding neat ways to capture common idioms. Now, it turns out that a clean and unified design is more likely to emerge from a consideration of abstract language features which are then fitted into an external syntax.

This is why on this course we have focused on internal syntax first – in our prefix style we only have to decide on set of features, their names and their arities and then we have the internal syntax defined. With it we can start to write semantic descriptions and discover unpleasant inconsistencies and interactions using a very simple regular syntax which supports reasoning. Once the design has settled down, we can 'decorate' the internal syntax by developing an external syntax that admits straightforward translation to our internal semantics-friendly notation.

The perfect language may not exist

It is conceivable that at some point in the future, all programs will be written in a single language. After all, we are still quite early on in the evolution of computing systems, and are only just emerging from the frenetic era of exponential hardware improvement during which Moore's 'law' has delivered a doubling of speed and memory density every 18 months for many decades.

However, at the moment new ideas and ways of implementing them continue to surface in experimental languages.

Programming languages and their ecosystems have lifecycles, just like any other kind of software. Many modern features such as generics, lambdas and limited type inference have been retro-fitted into Java's initial core, just as object oriented programming was retro-fitted to ANSI-C. It is likely that in the end these languages will be superceded by more elegant, integrated designs

Programming language facets

1. Names
2. Atomic values and types
3. Binding time and scope regions
4. Expressions
5. Sequencing, selection, iteration and call/return
6. Statements
7. Data abstraction
8. Functional and procedural abstraction
9. Packaging
10. Concurrency and non-determinism
11. Exceptions
12. Meta-programming and reflection

Selected ideas in programming languages

1. Throw away the syntax – Scheme, Forth
2. Orthogonality vs one-way-to-say – Algol-68, Lua, Pascal
3. When to bind – Python (dynamic) vs Occam (very static)
4. Organising arguments – keyword arguments vs signature ordering
5. Algebraic datatypes and pattern matching
6. Collection comprehensions – set builder notation
7. Making functions first order – lambdas
8. Making types first order – generic vs type variables
9. Pragmatic types – lengths do not add to areas; ranging
10. Type inference
11. Throw away the control flow - continuations
12. Concurrent friendly notions – map/reduce, array operations

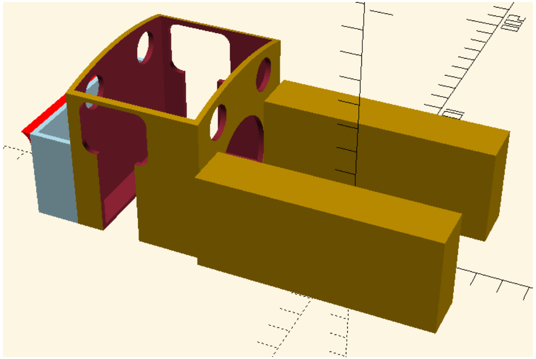
6C A Zoo of specification and implementation technologies

1. Deterministic lexing parsing and parsing
2. General lexing and parsing
3. Intermediate form languages
4. Structural Operational Semantics
5. Other approaches to formal semantics
6. Control flow analysis
7. Dataflow analysis
8. Traditional optimisations
9. Static scheduling
10. Dynamic scheduling

L1 Laboratory: Domain specific languages for 3D printing

Our first laboratory session is designed to familiarise you with one example of a Domain Specific Language.

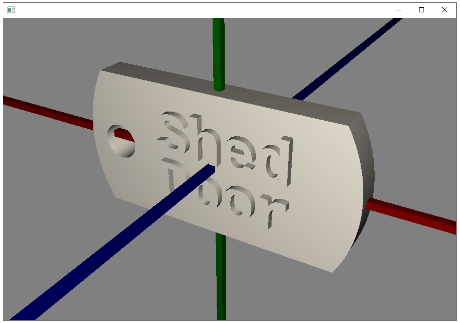
Please turn to Appendix A of the book, and work with Section A.1.

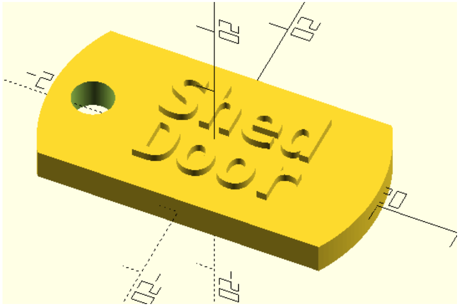


P1 Project: Java FX 3D as an internal DSL

Our first project session introduces you to some aspects of the JavaFX 3D modelling API, as one of thre epotential bases for your individual project.

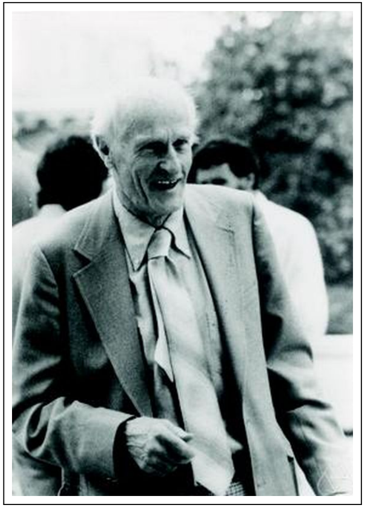
Please turn to Appendix B of the book, and work with Section B.1.





Revision topics

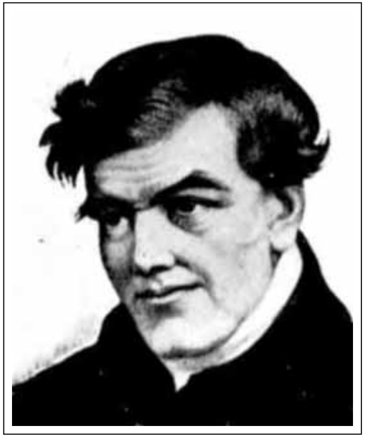
Term rewriting and the ARTValue system
Context Free grammars :idioms for context free grammars - priority, associativity, modularity [Cava exprssion grammar] Ordered Singleton Backtrack Recursive Descent parsing: Parse functions Semantic functions Tree builder functions GIFT opertions Producing rewritten derivation trees for direct input as ESOS terms
Attribute grammars: L-attribution [minicalc] Delayed attributes [minicall] Flow control Variables Accessing native code
SOS Inference rules specifiying relations The ESOS system The interpretation function Fsos Fsos traces Flow control Variables Dynamic type checking via the artvalue package
Domain specific languages Solid, image processing and music domains, physical unit types



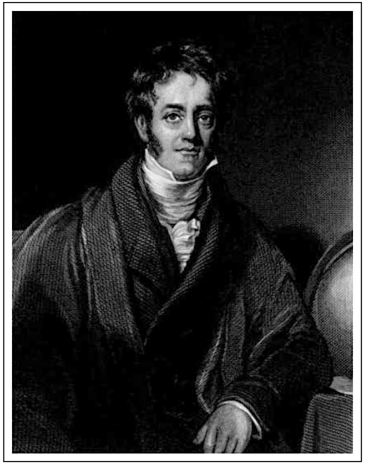
Stephen Kleene 1909–1994

https://en.wikipedia.org/wiki/Stephen_Cole_Kleene

Emil Post 1897–1954
https://en.wikipedia.org/wiki/Emil_Leon_Post



George Peacock 1791–1858
https://en.wikipedia.org/wiki/George_Peacock



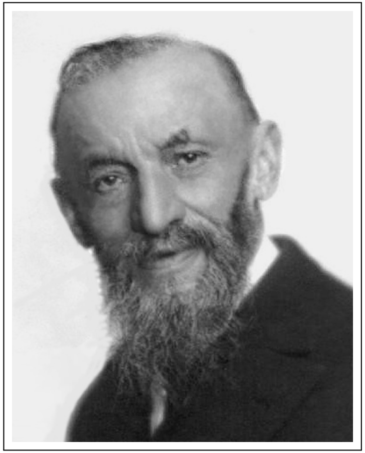
John Herschel 1792–1871

https://en.wikipedia.org/wiki/John_Herschel

Kurt Gödel 1906–1978

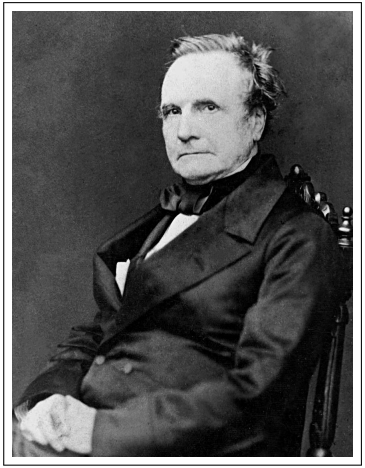
https://en.wikipedia.org/wiki/Kurt_Godel

Ludwig Wittgenstein 1889–1951
https://en.wikipedia.org/wiki/Ludwig_Wittgenstein



Giusepe Peano 1858–1932

https://en.wikipedia.org/wiki/Giuseppe_Peano



Charles Babbage 1791–1871
https://en.wikipedia.org/wiki/Charles_Babbage

Adrian Johnstone and Elizabeth Scott