ART laboratory and project manual

Adrian Johnstone
a.johnstone@rhul.ac.uk

September 29, 2025

Department of Computer Science
Egham, Surrey TW20 0EX, England

Contents

List of Figures

List of Tables

Chapter 0

Introduction

This document gives a series of lab and project exercises that underpin the curriculum for module *Software Language Engineering* at Royal Holloway, University of London.

Chapter 1

Solid modelling with OpenSCAD

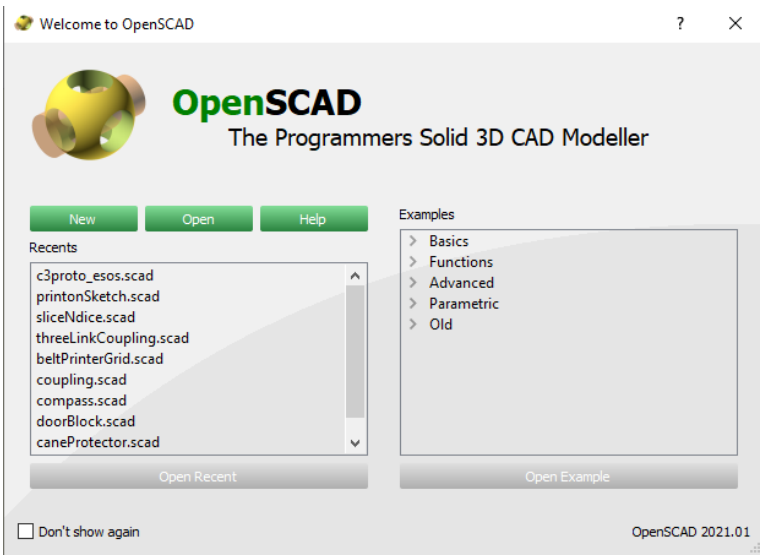
This laboratory session aims to help you understand the users' view of Domain Specific Languages. We shall look at the domain of solid modelling languages which may be used, amongst other things, to create objects suitable for 3D printing.

Whilst you are working, keep in mind the following questions: (i) is OpenSCAD easy to get started with? (ii) Is the syntax easy to use? (iii) Are there places where the syntax could be simpler? (iv) Can you pass 3D objects as arguments?

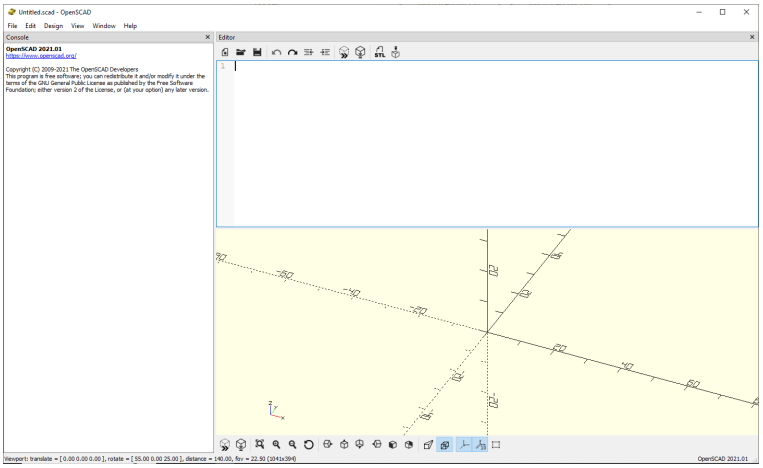
Install OpenSCAD for your system by accessing the download page at

https://openscad.org/downloads.html

When you first run OpenSCAD you will see something like this box:



Click on the New button and the OpenSCAD environment will open:



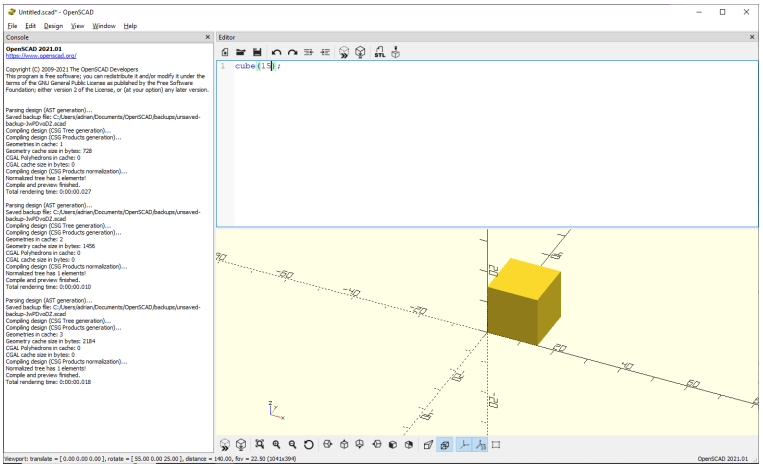
You see three windows: to the left a console and to the right a text editor above a graphics display window.

1.1 A first object

We begin by creating a cube. Use the text editor window to enter this command:

```
1 cube(15);
```

You *view* the code by pressing the F5 function key. This specification asks for a cube with 15mm edges. You should see this:



1.2 Changing the view

Left-click on the displayed object and drag the mouse pointer around. You will find that you can rotate the object around the origin. You can pan the view by

right-dragging, and zoom in and out by pinching, using the mouse wheel, or by clicking on the magnifying glass buttons.

There is a Help entry on the menu bar which will open a browser window on some online documentation and tutorials.

1.3 Changing size and colour

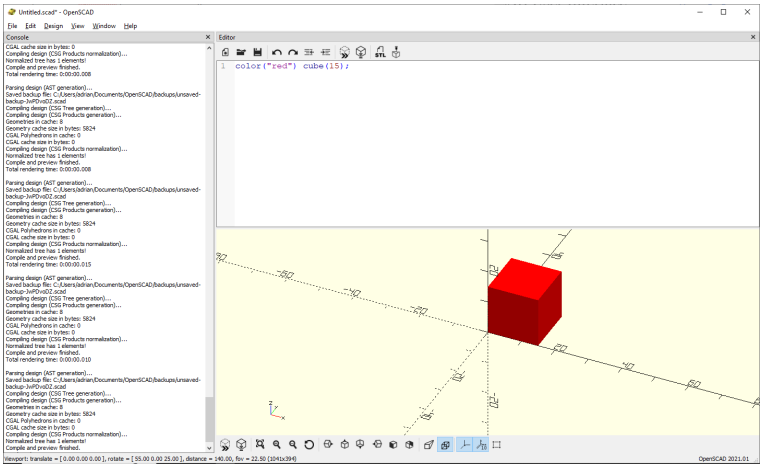
The numeric argument sets the length of the cube's edge. Change it to 25 and re-view with F5. The cube will become larger.

An interesting feature of the text editor is the ability to interactively change the value of a quantity, and immediately see its effect. Position the cursor between the 5 and the) in the text editor, and then press ALT-UPARROW. The argument will increment to 16, and the cube will be redrawn. ALT-UPARROW decrements, and you may use ALT-RIGHTARROW to add decimal places to a quantity.

You can set the colour of objects by preceding them with a color() modifier (note the American spelling!). Try:

```
1 color("red") cube(15);
```

The output should look like:

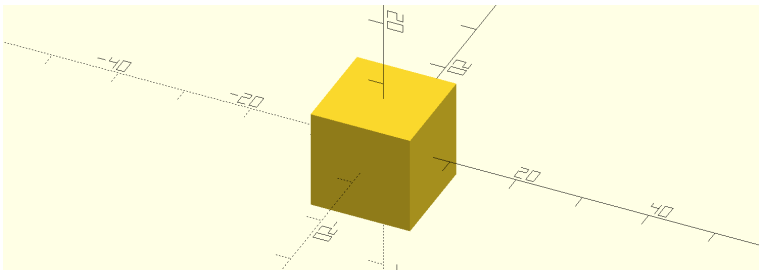


1.4 Where is the centre?

Rotations are specified around the origin, and as a result it is useful to ensure that all objects are created at the origin so that they can be re-oriented before being moved to their final position. Some objects like spheres are centred by default, but cubes are not. However, we can add an argument to force centring (again, note American spelling):

```
cube(15, center=true);
```

Notice how the cube is centred in all three axes: the coordinate origin is at the centre of the cube.

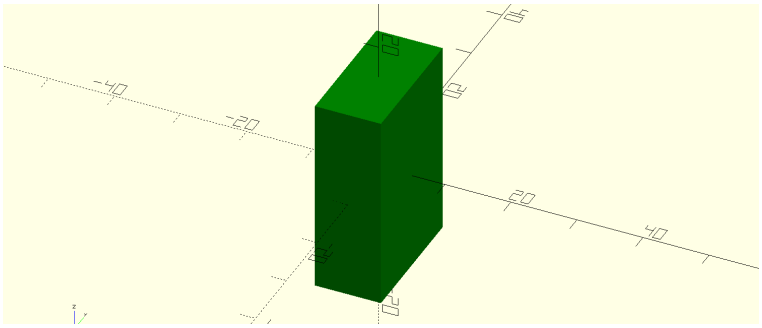


1.5 Cubes are really cuboids

The `cube` primitive can be used to specify cuboids, that is objects with varying x, y and z edge lengths.

```
color("green") cube([10,20,30], center=true);
```

The sequence of three numbers within square brackets is a *vector* and may be used to specify the three coordinates. Actually, just using a single integer, say `13`, in this context is taken to be shorthand for the vector `[13, 13, 13]`.

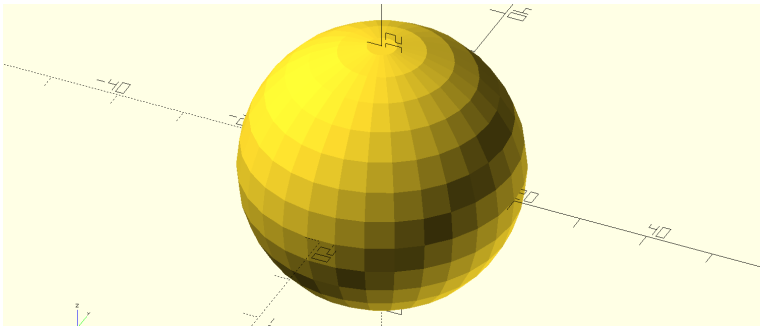


1.6 Spheres

OpenSCAD uses the *triangle mesh* method of representing objects: in reality all of the objects are represented as collections of flat triangles.

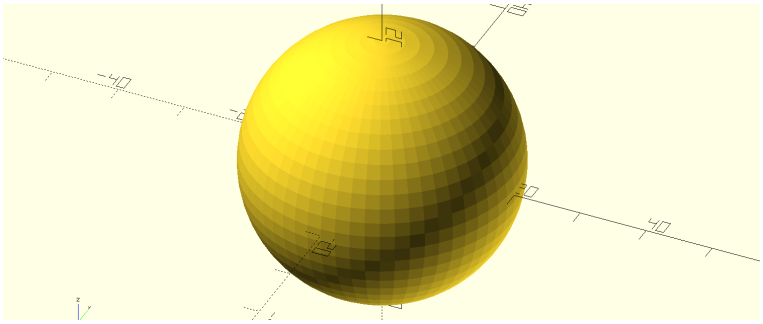
We can model spheres as polyhedra with sufficient faces to make the surface look smooth. The default for a sphere is only 30, which looks blocky if the sphere is large. The `sphere()` primitive constructs a spherical mesh:

```
sphere(20);
```



We can increase the number of segments in a circle (and by extension facets around a sphere) with the special argument `$fn`

```
sphere(20, $fn=100);
```



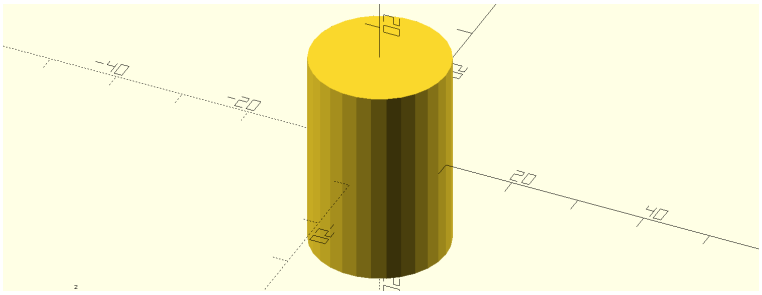
Try raising the value of `$fn` to 120, and then to 360. You will see that the sphere gets much smoother.

This example also shows the use of a language feature called *named arguments* which may be omitted (in which case a default value applies) or they can be explicitly specified. Contrast this with, say, Java method arguments which are strictly positional.

1.7 Cylinders and polyhedral bars

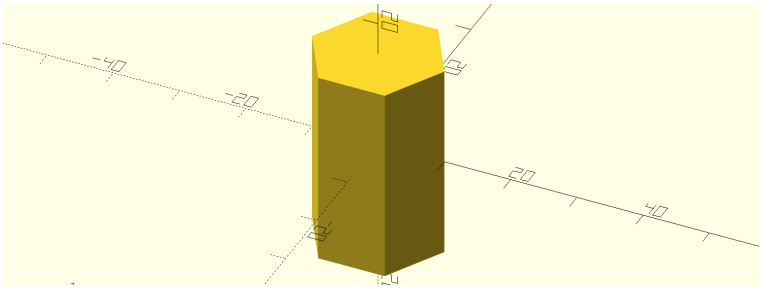
The `cylinder()` primitive takes a height `h` and a radius `r`:

```
cylinder(h=30, r = 10, center=true);
```



Now really, the cylinder primitive is just extruding a polygon. As before, we can use the `$fn` argument to make a cylinder smoother. We can also go the other way to make simpler objects. For instance, if we want to make an hexagonal bar, we can set it to six:

```
cylinder(h=30, r = 10, center=true, $fn=6);
```



We could in fact make many kinds of box this way too. Is there a `cylinder()` equivalent for every `cube()`?

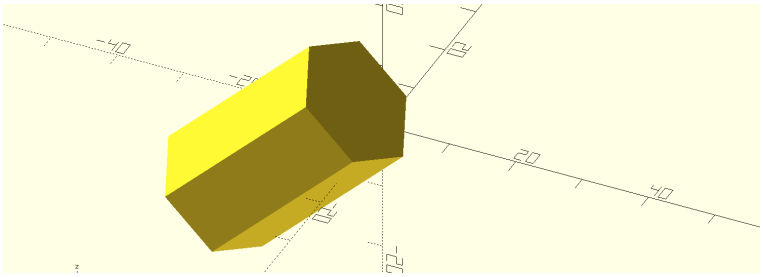
1.8 Translation and rotation

So far, we have made objects at the coordinate origin. We can move an object in space using the `translate()` operation which takes as an argument a vector: that is, the usual three values within square brackets corresponding to the x, y and z displacements.

For instance, changing the previous example to

```
translate([−10,−10,0]) rotate([0,45,0]) cylinder(h=30, r=10, center=true, $fn=6);
```

moves the hexagonal rod so that its centre is at $(x, y, z) = (-10, -10, 0)$ and so that is tilted 45 degrees around the y axis.

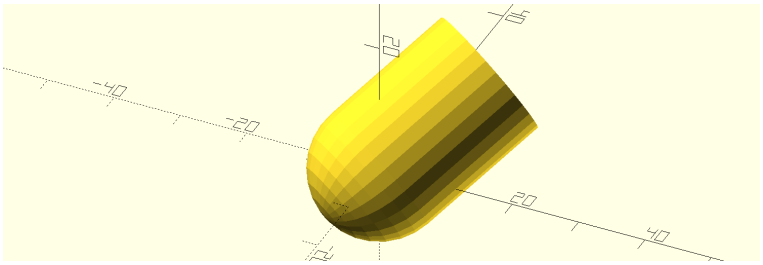


Rotations always occur around axes, so you will get very different effects if you rotate and then translate (as here) as opposed to translating and then rotating. The translate and rotate operations work like prefix operators, so they are effectively executed right-to-left. Experiment with changing the order of operations, and using the ALT-ARROW text editor mechanism to rotate and move things interactively.

1.9 Grouping objects

Typically we want to rotate and translate groups of objects together, which we can do by *uniting* then into a single mesh

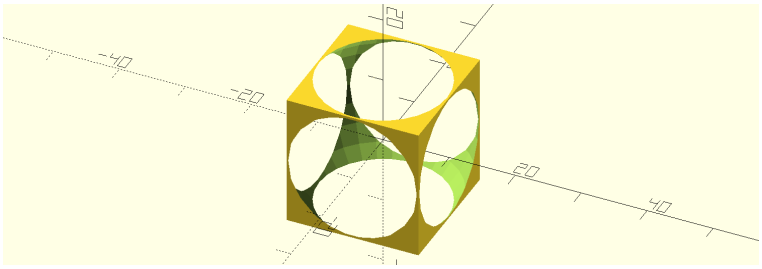
```
rotate([−45,45,0])
union() {
   cylinder(h=20, r = 10);
   sphere(10);
}
```



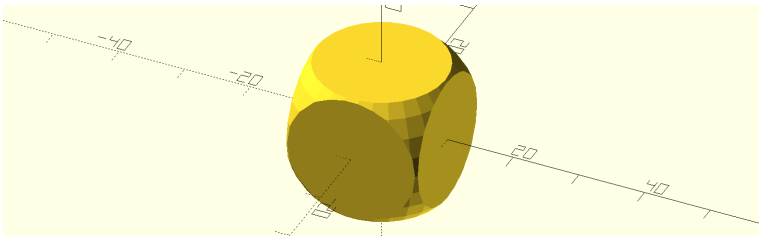
1.10 Computational solid geometry

Computational Solid Geometry (CSG) is a technique for making new objects from old via the operations of `union`, `difference` and `intersection`. We met `union` in the previous example. Here the equivalent examples for difference and intersection.

```
difference() {
   cube(20, center=true);
   sphere(14);
}
```



```
intersection() {
    cube(20, center=true);
    sphere(14);
}
```



The `difference()` operation is widely used to make holes in objects by subtracting a cylinder from them.

1.11 Using modules to structure a design

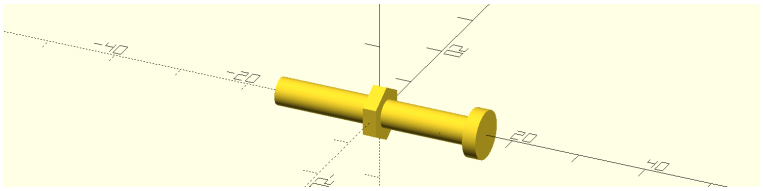
When programming, we use functions, procedures and methods to break our task down into manageably small pieces. In OpenSCAD, the equivalent construct is the *module* which wraps some 3D object descriptions up together with an implicit union, and allows arguments to *parameterise* the resulting objects.

In this example, we make a rough model of an hexagonal nut and a cylindrical bolt which are combined together in the `main()` function. The `bolt()` function takes an argument `length` which specifies how long the bolt should be.

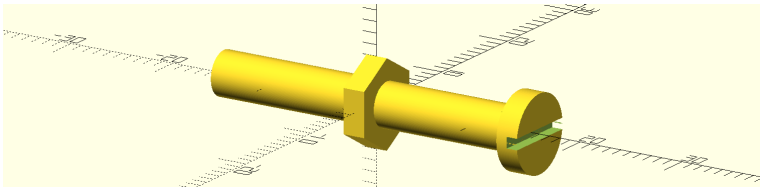
```
module nut() { cylinder(h=2, r = 4, $fn=6, center=true); }

module bolt(length) {
    cylinder(h=length, r=2, $fn=200, center=true);
    translate([0,0,length/2]) cylinder(r=3.5, h=2, $fn=200, center=true);
}

rotate([0,90,0]) union() {
    nut();
    bolt(30);
}
```



See if you can add a screw cut to the head of the bolt so that it looks like this:

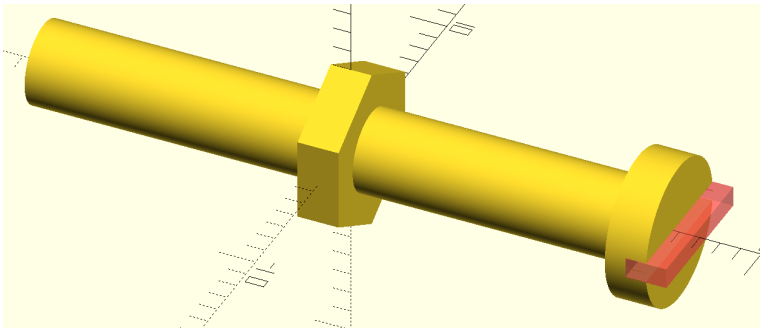


Hint: you need to subtract a cuboid from the larger of the two cylinder above.

1.12 The # 'ghost' modifier

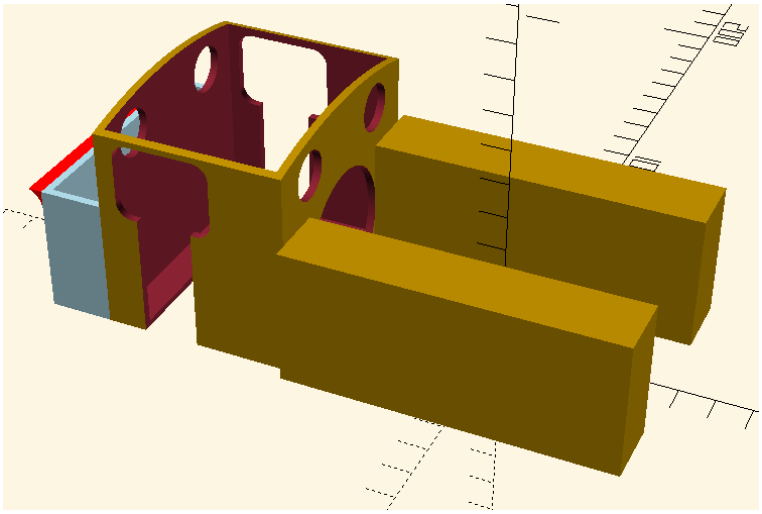
When trying to use the difference and intersection commands, it can be hard to visualise what is going on as some of the elements will be invisible.

If you place a # character in front of an invisible object it will be displayed in a ghostly form. So, for instance, when I was adding the screw slot above, I put a # character in front of the cuboid I was using to make the slot so that I could see its relative position to the screw head:



1.13 Your exercise: the J69

This is the body section from a simple steam engine, suitable for 3D printing:



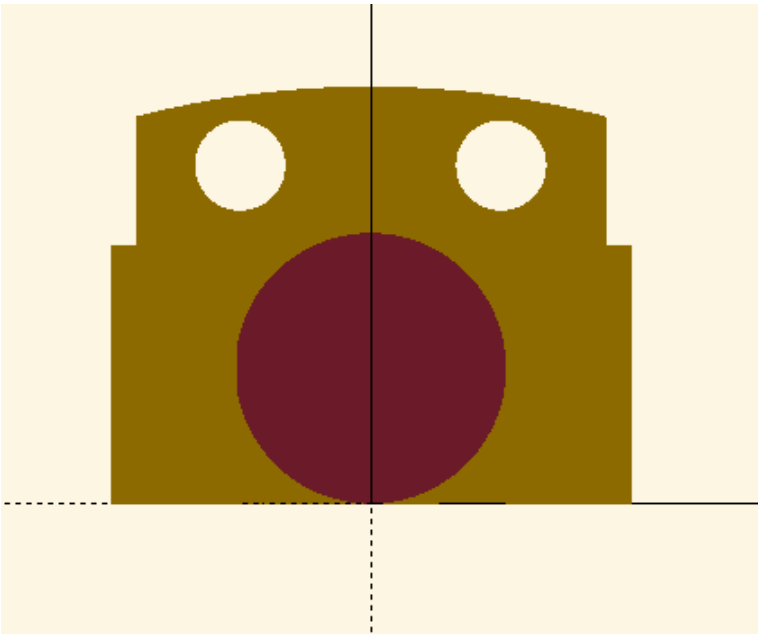
and here is a painted model:

It is based on a prototype, real world engine called the J69 class. You can read more about them at https://www.lner.info/locos/J/j67j69.php. There is just one real example left, shown here at Bressingham:

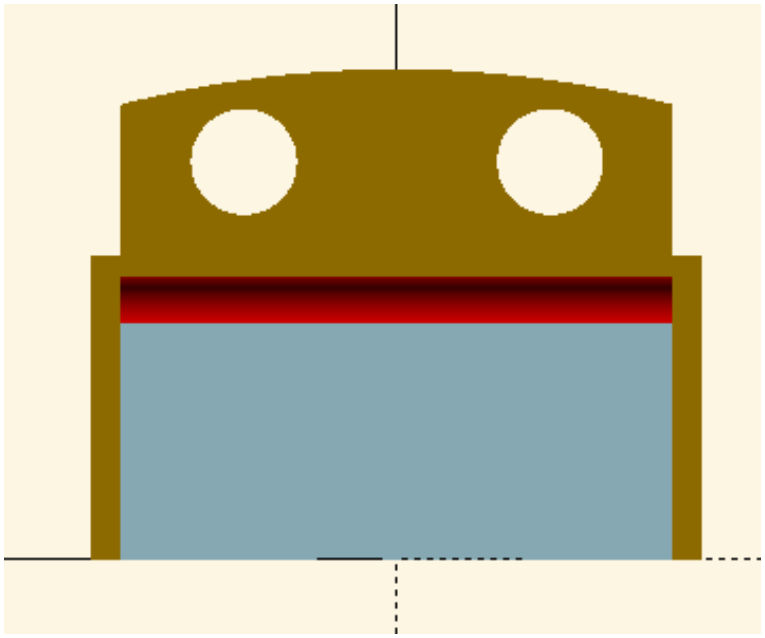
Your task now is to build your own version of the simple steam engine body, and if you are so inclined to embellish it so that it captures more of the prototype. Begin by defining the tanks, which are simple cuboids, and then make the cab as a cuboid which has another cuboid subtracted from it to make a hollow box. You can then subtract cylinders from it to make the round windows.

Here are reference views of the model from various angles.

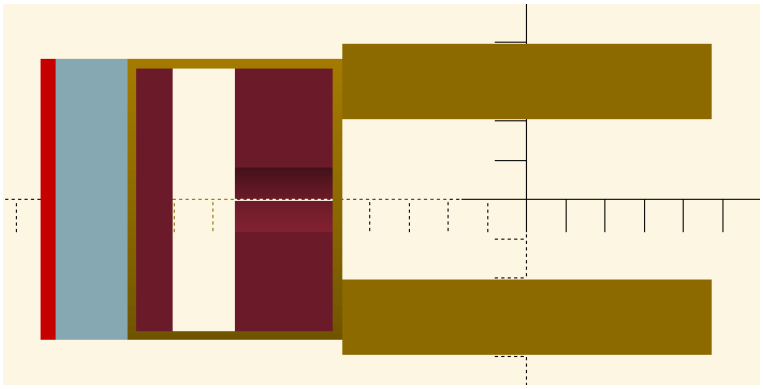
Front:



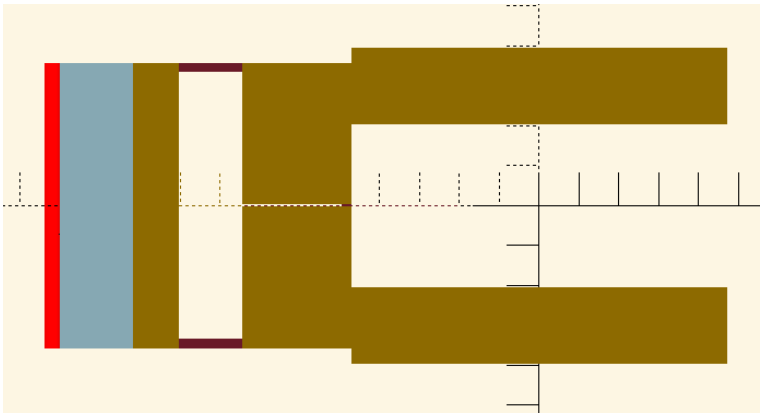
Rear:

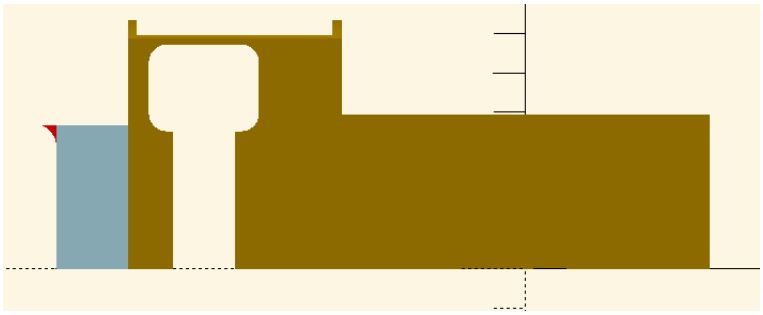


Above:



Below:



Side: 

When you have something you are pleased with, take a screenshot and then submit your work via Moodle. For inspiration, here are some examples from previous years.

Chapter 2

Simple rewrites

This laboratory session introduces you to the ART rewrite engine through *simple* rewrites — that is, rewrite rules that have no premises (i.e. nothing 'above the line'). In logic, these kinds of rules are called *axioms* because they exist as independent statements of fact that are not derived from any other facts. In our programming-style view of rewrite rules, these axioms can be viewed as rewrites that *must* be applied whenever the left hand side of their conclusion matches the top of the current term, Θ.

2.1 Download and test art.jar

Turn to section **??** and follow the instructions there. After completion, you will have a new directory (whose name and location is your choice, but which we shall refer to as ***artwork***). The directory will be empty except for a copy of `art.jar`.

2.2 A quick test

Open a command line window and change directory to ***artwork***. At the command line, type

```
java -jar art.jar !print version
```

You should receive a response from ART similar to this:

```
Attached to ARTDefaultPlugin
Interpreter set to eSOS
ART 5_0_385 2025-01-29 06:20:22
```

Of course, the version number, date and time will probably be later than shown here.

> **If you are running under Unix, Linux or MacOS then your command line shell may intercept ! characters and treat them specially... If you do not get the response above, try typing the command with an escape character before the ! like this:**
>
> ```
> java -jar art.jar \!print version
> ```

2.3 Make a work file

Use your preferred text editor to make a new working file called `reduction.art`. Eventually this file will contain the full specification of the reduction semantics for your language although in this lab we are just going to use it as a sandbox for experiments. When you have completed these exercises you *must* submit your final version of `reduction.art` via the Lab 2 Moodle page submission box so that we can track your progress, and give you support if you need it.

Add one line to the start of your new file containing

```
!print version
```

Save your file, and then at the command line type

```
java -jar art.jar reduction.art
```

You should see the same response as before.

```
Attached to ARTDefaultPlugin
Interpreter set to eSOS
ART 5_0_385 2025-01-29 06:20:22
```

> **In what follows, the idea is for you to copy each example into your `reduction.art` file, and then run it with the command**
>
> `java -jar art.jar reduction.art`
>
> **As you start each new example, please just comment out the parts you are no longer working on so that the final `reduction.art` that you submit shows all of the work that you have done. ART block comments are delimited (*like this*).**

2.4 A first rewrite

In ART, rewrite rules are written as an optional sequence of conditions, followed by three minus signs `---` followed by a *transition* comprising two terms separated by a *relation* symbol. In this lab we are using axioms which have no conditions, so each rule will just start with `---`. We shall also limit ourselves to the *small step transition* `->`.

Our simplest example, then, looks like this

```
1 --- a->b
2 !try a
```

The `!try` directive initialises the current term Θ to be the term you specify (just `a` in this case), and then passes it to the rewrite engine for processing.

The rewrite engine repeatedly attempts to apply its known rules to Θ until either it reaches a *terminal* or it gets stuck, because it has a Θ which is not terminal but for which no rules match. Each rewrite attempt is called a *step*.

When we run the example above, we get this trace output.

```
*** Warning: in relation -> constructor b has no rule definitions
Step 1
  Rewrite attempt 1: a ->
  -R1  --- a->b
  -R1 rewrites to b
Step 2
  Rewrite attempt 2: b ->
  No rules in relation -> for constructor b
Stuck on b after 2 steps and 2 rewrite attempts
```

Now this has worked as we can see that the a has been rewritten to b but something is not right because we are getting a warning message about missing rule definitions, and the rewriter says it is stuck. The reason for this unpleasantness is that the rewriter does not know that it is supposed to stop when it gets to b

We fix this by declaring that b is a *terminal* for which there are no rules, and at which the rewriter can terminate.

2.5 Tidying things up

The !terminal directive takes a transition arrow and a comma-delimited list of terminal terms. We need to specify a particular transition because sometimes we want different transition systems to have different terminals.

Change your example by adding the appropriate terminal declaration:

```
1 !terminal -> b
2 --- a->b
3 !try a
```

Try running it, and you should see this trace:

```
Step 1
  Rewrite attempt 1: a ->
  -R1  --- a->b
  -R1 rewrites to b
Normal termination on b after 1 step and 1 rewrite attempt
```

That is much more satisfactory. Once the `b` appears, the rewriter knows that it has reached a terminal and so it stops trying to step and terminates. We also get rid of the warning message about there being no rules for `b`.

2.6 Adding test outcomes to !try

We use !try directives to test our rules, and like any good testing system we would like to be able to supply an expected outcome for the test, and have the system give us a summary of how many tests have passed and failed. In ART, you can simply add a test term as part of each !try like this:

```
1 !terminal –> b
2 ––– a–>b
3 !try a = b
4 !try b = b
5 !try a = a
```

Here we have three !trys which will be performed in order. The third one is deliberately incorrect; we get this trace:

```
Step 1
  Rewrite attempt 1: a ->
  -R1  --- a->b
  -R1 rewrites to b
Normal termination on b after 1 step and 1 rewrite attempt
*** Successful test
Step 1
  Rewrite attempt 1: b ->
    Terminal b
Normal termination on b after 1 step and 1 rewrite attempt
*** Successful test
Step 1
  Rewrite attempt 1: a ->
  -R1  --- a->b
  -R1 rewrites to b
Normal termination on b after 1 step and 1 rewrite attempt
*** Failed test: expected a
Successful tests: 2; failed tests 1
```

This is how we do unit testing and regression testing in ART.

2.7 Changing the trace level

The rewriter tells you what it is doing, which is helpful for debugging but can be a bit overwhelming for long runs. ART has a global *trace level* which controls how verbose these messages are. The default is trace level 3, which tells you every rule that is being tested. Trace level 2 restricts that to just listing the successful rewrite rule. Trace level 1 further restricts the output to just giving the terminating term, and trace level 0 suppresses all of the rewriter messages.

There are higher trace levels too which we shall use in the next lab to look in detail at the processing of conditions.

You set the trace level by writing (for instance)

```
!trace 2
```

This will remain in force until the next !trace directive, or until the end of the script.

2.8 Comments and console messages

ART scripts can contain comments in two forms: line comments that begin with // and continue to the next line break, and block comments delimited by (*... *). So we can write things like

```
(* Lab 2 sandbox *)
!trace 2
!terminal -> b // set b as a terminal under -> transitions
--- a->b
!try a = b // first unit test
```

The !print ''a string'' directive allows you to output a message whilst a script is running.

2.9 Using variables: swapping

ART rewrite rules may contain *variables* which are bound to subterms as part of the matching process. Variable names all begin with a *single* underscore character.

A simple example is to take an arity-2 term `swap` and rewrite it so that the arguments are swapped over:

```
!terminal -> swapped
---swap(_X,_Y) -> swapped(_Y,_X)
!try swap(a,b) = swapped(b,a)
```

```
Step 1
  Rewrite attempt 1: swap(a, b) ->
  -R1  --- swap(_X, _Y)->swapped(_Y, _X)
  -R1 rewrites to swapped(b, a)
Normal termination on swapped(b, a) after 1 step and 1 rewrite attempt
*** Successful test
Successful tests: 1; failed tests 0
```

It you increase the trace level to 5, then the rewriter will show you the variable bindings as it goes along.

```
Trace level set to 5
Step 1
  Rewrite attempt 1: swap(a, b) ->
  -R1  --- swap(_X, _Y)->swapped(_Y, _X)
  -R1 bindings after Theta match { _X=a, _Y=b }
  -R1 rewrites to swapped(b, a)
Normal termination on swapped(b, a) after 1 step and 1 rewrite attempt
*** Successful test
Successful tests: 1; failed tests 0
```

2.10 A logic inverter

We shall now develop some rewrite rules that simulate basic logic gates. The idea is to encode the truth table of the logic function in a set of rewrite rules.

The simplest logic gate is the *inverter* or NOT-gate. It has one input and one output, if the input is true then the output is false, otherwise the output is true.

We shall use `T` and `F` to denote true and false respectively. Add these lines to your file

```
!terminal -> F,T
--- invert(T) -> F
--- invert(F) -> T

!try invert(T) = F
!try invert(F) = T
```

You should get the following trace:

```
Step 1
  Rewrite attempt 1: invert(T) ->
  -R1  --- invert(T)->F
  -R1 rewrites to F
Normal termination on F after 1 step and 1 rewrite attempt
*** Successful test
Step 1
  Rewrite attempt 1: invert(F) ->
  -R1  --- invert(T)->F
  -R1  Theta match failed: seek another rule
  -R2  --- invert(F)->T
  -R2 rewrites to T
```

```
Normal termination on T after 1 step and 1 rewrite attempt
*** Successful test
Successful tests: 2; failed tests 0
```

This is what we want,but notice how inverting T to F needs less work than inverting F to T because we have to search through the rules until we find one that matches.

2.11 An AND gate

We can now extend the approach to two-input gates. These rules embody the truth table for the AND function:

```
1 !terminal -> F,T
2 ---and(F,F) -> F
3 ---and(F,T) -> F
4 ---and(T,F) -> F
5 ---and(T,T) -> T
6
7 !try and(T,F) = F
8 !try and(T,T) = T
```

Try these out. Note again how the order of the rules affects the length of the trace.

2.12 Rewrite rules are looked at in order

We can quite often use a dirty trick to reduce the amount of searching that the rewrite engine has to do, and to reduce the number of rules that we have to write. ART has a special 'wildcard' variable written as _ (a single underscore) which will match anything. We can rewrite our `and` rules like this:

```
1 !terminal -> F,T
2 ---and(T,T) -> T
3 ---and(_,_) -> F
4 !try and(T,F) = F
5 !try and(T,T) = T
```

How does this work? Well the first rule handle the only row of the truth table that yields T (when both inputs are also T). All of the other truth table rows yield false. As long as we look for and(T,T) first, we can use the wildcards to handle all the other cases.

Of course, if we get the order of the rules wrong, then bad things happen. If we swap the rules

```
!terminal -> F,T
---and(_,_) -> F
---and(T,T) -> T
!try and(T,F) = F
!try and(T,T) = T
```

then we get this trace:

```
Step 1
  Rewrite attempt 1: and(T, F) ->
  -R1  --- and(_, _)->F
  -R1 rewrites to F
Normal termination on F after 1 step and 1 rewrite attempt
*** Successful test
Step 1
  Rewrite attempt 1: and(T, T) ->
  -R1  --- and(_, _)->F
  -R1 rewrites to F
Normal termination on F after 1 step and 1 rewrite attempt
*** Failed test: expected T
Successful tests: 1; failed tests 1\
```

The rule is that **we must proceed from the particular to the general.**

Now, this technique is useful for improving interpreter efficiency, but exploiting the ordering goes against the declarative styles that theorem provers and code verifiers demand. So if you are working with a formal verification process, youmay need to write out rules that are order-independent (and sometimes that is hard).

2.13 Your first exercise: OR, NOR, NAND and XOR

Using the style developed in the previous two sections, implement and write tests for the two input functions OR, NOR , NAND and XOR. In case you can't remember the truth tables for these functions, see the Wikipedia page at

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

2.14 Addition using 'beads'

Now that we have basic logic gates, we could in principle extend to full adders and indeed any binary digital hardware system. However we can use an alternative approach to implement arithmetic more directly. You'll recall that the Roman numbering system does not have a denotation for zero, and that their non-positional notation makes arithmetic directly in Roman numerals difficult. For everyday calculations, the Romans used an abacus: essentially a set of

wires strung with beads: by moving beads in and out of the viewing windows we can represent different numbers. We shall use terms and rewrites to model an abacus.

We shall make use of two terminals `bead` and `zero`. `bead` has arity one and `zero` has arity zero. The idea is that the number n is represented by a nest of n `bead` terms wrapped around a `zero` term:

```
0  zero
1  bead(zero)
2  bead(bead(zero))
3  bead(bead(bead(zero)))
```

and so on. We can increment (add one to) a number by adding an extra outer bead, and we can decrement (subtract one) by stripping the outer bead off.

Now, the addition operator takes two numbers represented in this way. We write rules that increment the first operand and decrement the second operand until the second operand is zero. Effectively the second operand counts down to zero, and the first counts up in lockstep.

```
!terminal -> bead, zero
--- add(_X, zero) -> _X
--- add(_X, bead(_Y)) ->add(bead(_X),_Y)
```

Use these !try commands like these to exercise these rules, and understand how they work.

```
!try add(zero,zero) = zero
!try add(bead(bead(bead(bead(zero)))), zero) = bead(bead(bead(bead(zero))))
!try add(bead(bead(bead(bead(zero)))), bead(bead(zero))) = bead(bead(bead(bead(bead(bead(zero))))))
```

What can you say about the number of steps that will be executed when adding p to q? Can you add rules to reduce the number of steps?

2.15 Subtraction using beads

For addition, we incremented the left operand whilst decrementing the right operand to zero. For subtraction, we can decerement the left operand whilst counting the right operand down. We do this by stripping the outer bead fromboth operands.

```
--- sub(_X, zero) -> _X
--- sub(bead(_X), bead(_Y)) ->sub(_X,_Y)
```

Here are some !trys that exercise subtraction.

```
!try sub(bead(bead(bead(bead(zero)))), bead(bead(zero))) = bead(bead(zero))
!try sub(bead(bead(zero)), bead(bead(zero))) = zero
!try sub(bead(bead(zero)), zero) = bead(bead(zero))
```

Now, there is a problem. Our subtraction operation only works with natural numbers: we have no way of representing and manipulating negative numbers. If we try a subtraction that yields a negative integer, the rules get stuck:

```
!try sub(bead(bead(zero)), bead(bead(bead(bead(zero)))))
```

gives us this trace:

```
Step 1
  Rewrite attempt 1: sub(bead(bead(zero)), bead(bead(bead(bead(zero))))) ->
  -R1  --- sub(bead(_X), bead(_Y))->sub(_X, _Y)
  -R1 rewrites to sub(bead(zero), bead(bead(bead(zero))))
Step 2
  Rewrite attempt 2: sub(bead(zero), bead(bead(bead(zero)))) ->
  -R1  --- sub(bead(_X), bead(_Y))->sub(_X, _Y)
  -R1 rewrites to sub(zero, bead(bead(zero)))
Step 3
  Rewrite attempt 3: sub(zero, bead(bead(zero))) ->
  -R1  --- sub(bead(_X), bead(_Y))->sub(_X, _Y)
  -R1  Theta match failed: seek another rule
  -R2  --- sub(_X, zero)->_X
  -R2  Theta match failed: seek another rule
Stuck on sub(zero, bead(bead(zero))) after 3 steps and 3 rewrite attempts
```

As soon as the left hand operand reaches zero, we get stuck.

There is a kind of arithmetic called *saturation arithmetic* which sets lower and upper bounds (l, u) on numbers, typically zero and some integer such as 255 or 65,536. You can read more at https://en.wikipedia.org/wiki/Saturation_arithmetic

In saturation arithmetic, $l - 1 = l$ and $u + 1 = u$. We can model this by adding bounds rules. Here is subtraction again, but with a rule that sets a lower bound of zero.

```
--- sub(bead(_X), bead(_Y)) ->sub(_X,_Y)
--- sub(_X, zero) -> _X
--- sub(zero, _Y) -> zero
```

You will now get normal termination, but of course any negative results are jammed to zero. This kind of arithmetic is useful in signal processing (both

image processing and audio processing) and some specialised signal processing devices such as the Analog Devices ADSP 2105 processor offer machine code instruction for saturated arithmetic.

What rule would you need to put a upper bound of ten on the addition operation?

2.16 Your second exercise: arithmetic with negative numbers

As we have seen, the subtraction rules above really only work with the naturals, and once the left hand operand gets down to zero we get stuck. Your second (quite hard) exercise is to find a number representation and some rules that allow subtraction involving negative quantities, i.e. to extend our `sub` operation from the Natural numbers to the Integers.

Here's a hint: represent each Integer n as a pair of Natural numbers (p, q) where $n = p - q$. You may (and actually you may not) find this CS Stack exchange page useful: https://cs.stackexchange.com/questions/2272

Try and write rules for both addition and subtraction.

Can you do multiplication and division?

(Health warning: I have not worked out answers for these myself yet, and these puzzles can keep you awake at night... Have a play and learn, but don't be concerned if you can't find a perfect solution.)

2.17 The ART value system - a preview

It is pretty clear that doing arithmetic in element rewrites is hard work, both in design terms but perhaps more importantly in terms of the quantity of rewriting that is required: we do not really want to have to represent the number 1,000,000 with a million-and-one nested terms.

In practice, we use these ideas to convince ourselves that rewrite systems *could* handle all of the operations we see on a modern processor,and then we cheat a bit and provide a set of special terms which when substituted into other terms magically perform complicated calculations. In reality these special operations are escape hatches from the rewriting world into conventional computation, and they act both to help structure our rules around well-understood operations, and to massively improve efficiency.

In ART, these special functions all have names starting with two underscores, like `__add` and `__sub`. We also provide a set of types, and some shorthands when writing terms. So, for instance, here is a much easier way to do addition:

```
1 ---- add(_X, _Y) -> __add(_X, _Y)
2 !try add(3,4) = 7
```

Note carefully that the `add` constructor is *not the same thing* as the `__add` built in function. It is easy to get confused.

When you run this example, you should see:

```
Step 1
  Rewrite attempt 1: add(3, 4) ->
  -R1  --- add(_X, _Y)->__add(_X, _Y)
  -R1 rewrites to 7
Normal termination on 7 after 1 step and 1 rewrite attempt
*** Successful test
```

The addition gets done in one magical step, rather then the countdown sequence that we saw earlier.

We shall look at ART's *value system* in much more detail later. For now, you might like to glance at Section **??** to see a list of these builtin types and operations.

Please do not forget to submit your `reduction.art` file to Moodle when you have finished, and by the deadline.

Chapter 3

Starting a language

This laboratory session introduces you to the discipline of language design.

The work is is in two parts: firstly we shall develop the *internal syntax* of your project languagewhich comprises a set of primitive types, a set of *signatures* and a *configuration*. We will then look at the efficient implementation of arithmetic and logic operations using ART's builtin types and operations (rather than the bead-based rules we looked at in Lab 2) and how to compose them into compound expressions.

You have a blank sheet of paper, so where do you begin?

Often, folk try to replicate the 'look' of their favourite language but with some personal tweaks. Of course, studying existing languages can be a fertile source of inspiration but extending such a language can be very challenging, and many projects founder in what Fred Brooks famously called *the tar pit of complexity*. We do not recommend this approach because fitting new capabilities into an existing language is hard, especially if you have no experience of building a language from the ground up.

Instead, we recommend a 'semantics first' approach in which the designer begins by simply listing the features of the language, independent of the syntax. We begin by thinking about primitive (atomic) types, the basic arithmetic and logical operations and then turn to control flow, data type constructors and special domain-specific operations.

For each operation, we write a *signature* comprising a name for the operation and then a sequence of operands with optional type constraints.

So, for instance, subtraction over 32-bit integers might have this signature:

```
1 sub(_L: __int32, _R: __int32):__int32
```

This says that there is an operation called sub which takes two operand called __L and __R (for left and right) each of which must be of type __int32, and the operation returns a value which will also be of type __int32. The type constraints are just that — things that must be checked — and are optional. If they are omitted then that element of the signature will accept anything.

Similarly, the 'not equal to' operation might have this signature:

```
neq(_L, _R):__bool
```

taking two unconstrained operands and returning a boolean.

The arguments to a signature can be code as well as data. For instance we might represent an if—then—else feature as

```
if(_P:__bool, _S1, _S2):__bool
```

Here _P is a *predicate* return a boolean result, and _S1 and _S2 are unconstrained terms that might represent simple numbers, expressions or statements. We can use this to represent these three Java-like features:

1. A selection expression a > b ? 3 : 4; represented as if(gt(a,b),3,4)
2. An if—then statement if (a > b) return 3; as if(gt(a,b),3,__done)
3. An if—then—else statement if (a > b) return 3; else return 4; represented as if(gt(a,b),3,4)

Notice how cases 1 and 3 have the same internal syntax but different external syntax. (Actually we have cheated a little as the Java return may need some special handling.)

Once we have a plausible set of signatures, we start to write the rules that will 'explain' their semantics using rewrite rules. Often one finds that a few new 'helper' signatures will be needed, especially for features such as a switch (or case) statement that have a sequence of sub-operations.

We can test individual features and features in combination by trying different test terms. Once we have a complete and consistent set of rules that process our test cases to give the expected results (and do not get stuck!) then we can write some context free rules to create an external syntax parser which outputs expressions built from our semantic signatures.

Like all software engineering processes, we end up iterating the design as issues arise during development and testing, but this separation of concerns between the semantics and the design of syntax (the 'look') of the language avoids a problem that ofte arises otherwise: syntactic constraints which create special cases in the semantics that complicate the rules. We seek uniformity and orthogonality in the semantic rules; then we use the syntax and static typing rules to outlaw dubious combinations.

As an example, in the original C language the array indexing operation is rather general; one can write something like a[3] (which is conventional) but also 101[3], that is the base of the array may be an explicit number. This eases the design of a translator, because the indexing operation may have a signature

like index(_base:__int32, _offset__int32) where any expression that yields a 32-bit integer may appear as the base. Now, in some contexts this is a feature; in others a bug. Allowing a numeric base means that we can directly address any specific location in memory and are not just limited to areas that have been allocated by the translator and given an identifying name. For some embedded applications, accessing specific memory locations is a necessary feature. On the other hand, unfettered access to memory on multi-process systems allows malignant code access to information which should be secured, and that is at the root of much malware. Therefore we would expect a compiler for such a system to use syntax and static semantic checks to filter out such idioms before the dynamic semantic rules are activated.

Similarly, in the selection examples above we would expect the parser to enforce the requirement that selection *expressions* must always have both a THEN and an ELSE clause so that the expression always returns some value, whereas a selection *statement* is valid without the else clause, as that simply means 'do-nothing'.

3.1 Exercise: primitive type selection

Review the type structure of the Java language by reading sections 4.1 and 4.2 of the Java Language Specification at https://docs.oracle.com/javase/specs/jls/se23/html/jls-4.html#jls-4.1.

Pocket calculators do not distinguish between floating point and integer numbers, but Java and other languages provide a plethora of different integer and floating point data types. Why?

Review the Java BigDecimal and BigInteger classes, documented at

https://docs.oracle.com/javase/8/docs/api/java/math/BigDecimal.html
and
https://docs.oracle.com/javase/8/docs/api/java/math/BigInteger.html

How do they work, and what are their advantages and disadvantages?

ANSI C-89 does not have a Boolean type. How are the results of predicates represented?

Examine the Java GCD program on page **??** and the 'compact' version that follows. What data types are in use?

For the three domains of music, image processing and 3D CAD, what data types do we need, and which might be desirable?

What will you use in your language, and why?

3.2 Exercise: enumerate the operations and types in the GCD language

Refer back to the GCD program on page **??**. Do not think about syntax at the moment, but just try to list the primitive types and operations in use in that example, and write a signature for each of them. Do not overlook control flow, assignment to variables and dereferencing of variables.

3.3 Deciding on the configuration

We model programming languages as transitions between states of a *configuration* which contains at the very least a program term. For anything other than a truly pure functional langauge (and these are rare beasts) the configuration will also contain some other *semantic entities* whose function is to capture side effects such as assignment and output as we rewrite the program term.

Once you have a clear idea of the operations that your language will provide, you will be able to decide which semantic entities they need, and hence what your language's configuration will be. We represent configurations as *tuples* of semantic entities, written, for instance, as $\langle \theta, \sigma, \alpha \rangle$. Every language needs a at least a program term: the configuration for a pure functional language would be $\langle \theta \rangle$. We shall now consider other kinds of semantic entity.

The store and environment - σ and ρ

A language with assignment needs at least a store.

In more detail, a language where all variables are global can only have one binding to each variable name. In languages with multiple scope regions, the same variable name can refer to different variables (and thus different bindings) in each scope region. In either case we need a store (usually called Sigma, or σ) which represents main memory. If we also want multiple scope regions, then we need a *symbol table* or *environment* (usually called Rho or ρ) to be associated with each scope region.

When just using σ we implement it as a map from identifiers to values. Wen using environments, we have single map σ from location identifiers (often natural numbers, as they are in the hardware) to values, and then each scope region has environment which is a map from identifiers to location identifiers. This two-level representation allows the same identifier to appear in different scopes bound to different locations in the store, and indeed for multiple identifiers to bind to the same store location, as is required for reference based languages such as Java.

If your language is to have a procedure-like construct (e.g. methods in Java, functins in C and modules in OpenSCAD) then multiple scopes are almost mandatory otherwise you cannot model local variables.

Output and input - α and β

Many languages need output α and and input β. Whilst prototyping our language, we usually just represent these semantic entities as linked lists. In reality, input and output are rather fraught aspects of programming as illustrated by the scale of the java.io and java.nio packages: see

https://docs.oracle.com/javase/8/docs/api/java/io/package-summary.html and
https://docs.oracle.com/javase/8/docs/api/java/nio/package-summary.html

And then, of course, there is input and output associated with Graphical User Interface systems such as Java Swing and Java FX. All of these libraries are complex to the point of being overwhelming,but fortunately we do not need to consider them as they are all written *in* the host programming language and are not features *of* the language. However there are languages such as Pascal and BASIC that do have simple I/O built into them. Some languages, such as the OpenSCAD language from Lab 1 do not really have input at all, and only very limited output facilities.

Exceptions and signals - ν

An *exception* breaks the normal flow of control. Typically, we *raise* and exception which then causes nested functions to return until we encounter an *exception handler* that is given control. Operating system *signals* and hardware *interrupts* are similar events which are generated outside of the running user program.

It is perfectly possible to model exceptions in our rewrite rules, but when starting out it is probably best to delay thinking about exceptions until the core of the language is stable, so we shall not discuss exceptions in this lab.

Domain specific entities

It *may* (and may not) be useful to have semantic entities representing state for the domain specific entities that your programming language is manipulating. A common approach is keep all such entities on the Java side of the system, and use anonymous numeric handles to refer to them on the rewrite side. An alternative is to use the builtin type __blob to hold references to arbitrary binary objects. We shall consider this more fully when we look a the ART plugin conventions in a later lab.

3.4 Exercise: decide on your configuration

For a general purpose language that will act as the base for your project, what configuration should you use?

At this point in the design process, you have decided on the ***internal syntax*** of your language, the notation that defines the state space and operation repertoire of your language.

You may need to revisit and extend your design, but now we are ready to begin implementing a prototype rewrite-based interpreter.

3.5 Values and their types in ART

ART's rewriter is very simple minded: it looks for matches between closed terms and patterns which may contain variables, and creates an environment of bindings for those variables that may be substituted into another pattern to make a new closed term. It is the very simplicity of this matching operation that makes it a suitable basis for defining language semantics since it acts a sort of lowest-common-denominator description of computation.

The matching process only consider the shape of the terms (trees) and the labels on the nodes which are treated as simple text strings without interpretation. So the label `123` is just a string of characters. At the level of the programming language we are modelling we might want that label to represent an integer, or a real number or the a programming language string. How do we differentiate these cases?

The idea is that a value should be a term in the form type(payload). The type label can be anything you like, but ART has built-in support for a set of type names that are recognised during substitution, all of which start with two underscores. So, for instance, a 32-bit integer 123 is represented as __int32(123), a Java double as __real64(123) and a string as __string(123). You can also have __intAP(123) which represents an arbitrary precision integer (like a BigInteger in Java) and __realAP(123) which represents an arbitrary precision real number.

Shorthands in the ART front end

It becomes quite tedious to have to write __int32(123) and __real64(123.0) so the ART scripting front end *automatically* converts certain forms to the built in type equivalents.

You can see this process in action by asking ART print out the raw form of terms:

```
1 !try 123.0
2 !print derivation
3 !printraw derivation
4
5 !try 321
```

```
!print derivation
!printraw derivation
```

In detail the !try directive loads the current derivation term with its argument. The !print directive prints out the 'cooked' (abbreviated) version of this term; the !printraw directive shows explicitly what is going on under the hood.

```
current derivation term: [1383]
123.0
current raw derivation term: [1383]
trTopTuple(__real64(123.0))
current derivation term: [1392]
321
current raw derivation term: [1392]
trTopTuple(__int32(321))
```

The trTopTuple constructor is used to tie together the elements of a configuration tuple.

3.6 Implementing arithmetic operations with the ART Value system

The bead based rules from Lab 2 are an interesting parlour game, and show that pure rewriting can perform arithmetic, but they are not at all practical for our present purpose since the size of the term representing a number is proportional to that number, and for anything other than very simple examples, vast amounts of storage and rewriting are required.

The ART builtin types are accompanied by a fixed set of builtin operations, the names of which also start with two underscores. When the rewriter is making a substitution, if it sees a builtin operation it converts it's arguments to their equivalent Java partner types, then performs the operation as a Java expression, and then converts the result it back to a term.

The complete set of builtin types and operations is summarised in the table on page **??**. The blue entries are disallowed combinations: for instance it makes no sense to divide strings by one another. Allowed operations have a grey background, and the label tells you the expected type of the arguments.

We use these functions by creating our own signatures for operations such as subtraction which then rewrite to the builtin operation. As a side effect of substituting the bindings, the builtin operation is evaluated;

```
--- subtract(_L, _R) -> __sub(_L, _R)
!try subtract(12,7)
```

```
Step 1
   Rewrite attempt 1: subtract(12, 7) ->
   -R1 --- subtract(_L, _R)->__sub(_L, _R)
   -R1 rewrites to 5
Normal termination on 5 after 1 step and 1 rewrite attempt
```

Now, the example above is *fragile* in that there is no check that the arguments to __sub are valid, and that can cause ART to stop with a fatal error:

```
--- subtract(_L, _R) -> __sub(_L, _R)
!try subtract(12,7.0)
```

yields

```
Step 1
   Rewrite attempt 1: subtract(12, 7.0) ->
   -R1 --- subtract(_L, _R)->__sub(_L, _R)
*** Fatal: Term 7.0 failed type check type against __int32
```

Note that this is not the same thing as getting stuck: the rewriter has had to completely stop in an uncontrolled way because it is not able to evaluate the __sub operation. The problem is that we are trying to subtract the real number 7.0 from the integer 12. The builtin operations require the arguments to be of the same type, so you can add and subtract pairs of reals or integers, but cannot do mixed mode arithmetic.

How can we avoid the fatal error? We add conditions above the line. Recall that the match operation is represented by the ▷ symbol. In ART's script language we write that as |>, and use it to check the type of both of the operands:

```
!trace 5
_L |> __int32(_) _R |> __int32(_)
---
subtract(_L, _R) -> __sub(_L, _R)

!try subtract(12,7.0)
!try subtract(12,7)
```

The way to interpret this rule is to start at bottom left, with the left hand side of the conclusion. That must match the current term for this rule to be 'triggered'. As part of the match, _L and_R will be bound to terms. We then check the conditions above the line: both _L and_R must match __int32(_) that isthe root must be __int32 and there must be exactly one child. If the matches pass, then the rewrite occurs. If not, then the rule will be discarded and the rewriter will look for another. If it can't find one then it will announce that

it is stuck; which is what we want here because we are trying to avoid that irrecoverable fatal type error.

```
*** !try(subtract(12, 7.0))
Step 1
  Rewrite attempt 1: subtract(12, 7.0) ->
  -R1 _L |> _ _R |> _ --- subtract(_L, _R)->__sub(_L, _R)
  -R1 bindings after Theta match { _L=12, _R=7.0 }
  -R1 premise 1 _L |> _
  -R1 bindings after premise 1 { _L=12, _R=7.0 }
  -R1 premise 2 _R |> _
  -R1 premise 2 failed: seek another rule
Stuck on subtract(12, 7.0) after 1 step and 1 rewrite attempt
*** !try(subtract(12, 7))
Step 1
  Rewrite attempt 1: subtract(12, 7) ->
  -R1 _L |> _ _R |> _ --- subtract(_L, _R)->__sub(_L, _R)
  -R1 bindings after Theta match { _L=12, _R=7 }
  -R1 premise 1 _L |> _
  -R1 bindings after premise 1 { _L=12, _R=7 }
  -R1 premise 2 _R |> _
  -R1 bindings after premise 2 { _L=12, _R=7 }
  -R1 rewrites to 5
Normal termination on 5 after 1 step and 1 rewrite attempt
```

Note how the mixed mode !try subtract(12,7.0) sticks, but the well-formed !try subtract(12,7) passes both type checks and rewrites successfully.

We raised the !trace level to 5 so that you can see the bindings and the individual premises being evaluated. This is very whilst during debugging.

3.7 Nested expressions

We now have an efficient way of getting the rewriter to perform *single* arithmetic operations, but of course we really want to be able to evaluate nested terms: we might represent the Java expression 5−2−1 as sub(sub(5,2),1). The left hand operand of the outer sub is not an integer, so the rule above will not pass its type checks:

```
!trace 5
_L |> __int32(_) _R |> __int32(_)
---
sub(_L, _R) -> __sub(_L, _R)

!try sub(sub(5,2),1)
```

yields

```
Step 1
  Rewrite attempt 1: sub(sub(5, 2), 1) ->
  -R1 _L |> _ _R |> _ --- sub(_L, _R)->__sub(_L, _R)
  -R1 bindings after Theta match { _L=sub(5, 2), _R=1 }
  -R1 premise 1 _L |> _
  -R1 premise 1 failed: seek another rule
Stuck on sub(sub(5, 2), 1) after 1 step and 1 rewrite attempt
```

(Notice by the way that we changed the constructor from subtract to sub - it is just a text string and we can use whatever we like as long as the !try argument has a matching rule.)

The trick is to somehow get the rewriter to process the _L argument separately so as to reduce it to an integer. We achieve this by making a second rule that has a *transition* above the line. This has the effect of recursively calling the rewriter on the argument until it is reduced to a value. We call this kind of rule a *resolution rule*. In this specification, we have named the first rule subBase and the second rule subResolve. These rule names make it easier to read the trace output, which as you can see gets quite long.

```
!trace 5
- subBase _L |> __int32(_) _R |> __int32(_)
---
sub(_L, _R) -> __sub(_L, _R)

-subResolve _L -> _LP
---
sub(_L, _R) -> sub(_LP, _R)

!try sub(sub(5,2),1)
```

The order of the rules is important. We need the rewriter to first check the base rule, and only if that fails can we go on to try and resolve sub-expressions.

```
Step 1
  Rewrite attempt 1: sub(sub(5, 2), 1) ->
  -subBase _L |> _ _R |> _ --- sub(_L, _R)->__sub(_L, _R)
  -subBase bindings after Theta match { _L=sub(5, 2), _R=1 }
  -subBase premise 1 _L |> _
  -subBase premise 1 failed: seek another rule
  -subResolve _L->_LP --- sub(_L, _R)->sub(_LP, _R)
  -subResolve bindings after Theta match { _L=sub(5, 2), _R=1 }
  -subResolve premise 1 _L->_LP
    Rewrite attempt 2: sub(5, 2) ->
    -subBase _L |> _ _R |> _ --- sub(_L, _R)->__sub(_L, _R)
```

```
      --subBase bindings after Theta match { _L=5, _R=2 }
      --subBase premise 1 _L |> _
      --subBase bindings after premise 1 { _L=5, _R=2 }
      --subBase premise 2 _R |> _
      --subBase bindings after premise 2 { _L=5, _R=2 }
      --subBase rewrites to 3
   --subResolve bindings after premise 1 { _L=sub(5, 2), _R=3, _3=1 }
   --subResolve rewrites to sub(3, 1)
Step 2
   Rewrite attempt 3: sub(3, 1) ->
   --subBase _L |> _ _R |> _ --- sub(_L, _R)->_sub(_L, _R)
   --subBase bindings after Theta match { _L=3, _R=1 }
   --subBase premise 1 _L |> _
   --subBase bindings after premise 1 { _L=3, _R=1 }
   --subBase premise 2 _R |> _
   --subBase bindings after premise 2 { _L=3, _R=1 }
   --subBase rewrites to 2
Normal termination on 2 after 2 steps and 3 rewrite attempts
```

What happens if the order of the rules is reversed, and why?

The beauty of this approach is that the resolution will recursively compute any depth of nested expression. Here is a trace of !try sub(sub(sub(5,2),1),1) with !trace 3 so as to reduce the volume of output, whilst still showing the individual rule activations.

```
Trace level set to 3
Step 1
   Rewrite attempt 1: sub(sub(sub(5, 2), 1), 1) ->
   --subBase _L |> _ _R |> _ --- sub(_L, _R)->_sub(_L, _R)
   --subResolve _L->_LP --- sub(_L, _R)->sub(_LP, _R)
      Rewrite attempt 2: sub(sub(5, 2), 1) ->
      --subBase _L |> _ _R |> _ --- sub(_L, _R)->_sub(_L, _R)
      --subResolve _L->_LP --- sub(_L, _R)->sub(_LP, _R)
         Rewrite attempt 3: sub(5, 2) ->
         --subBase _L |> _ _R |> _ --- sub(_L, _R)->_sub(_L, _R)
         --subBase rewrites to 3
      --subResolve rewrites to sub(3, 1)
   --subResolve rewrites to sub(sub(3, 1), 1)
Step 2
   Rewrite attempt 4: sub(sub(3, 1), 1) ->
   --subBase _L |> _ _R |> _ --- sub(_L, _R)->_sub(_L, _R)
   --subResolve _L->_LP --- sub(_L, _R)->sub(_LP, _R)
      Rewrite attempt 5: sub(3, 1) ->
      --subBase _L |> _ _R |> _ --- sub(_L, _R)->_sub(_L, _R)
      --subBase rewrites to 2
   --subResolve rewrites to sub(2, 1)
```

```
Step 3
  Rewrite attempt 6: sub(2, 1) ->
  -subBase _L |> _ _R |> _ --- sub(_L, _R)->__sub(_L, _R)
  -subBase rewrites to 1
Normal termination on 1 after 3 steps and 6 rewrite attempts
```

Now, we have not quite yet finished because our resolution rule only resolves left operands. We need to add a third rule which handles the right operand, and we need to ensure that the left operand will always be evaluated first.

```
!trace 3
- subBase _L |> __int32(_) _R |> __int32(_)
---
sub(_L, _R) -> __sub(_L, _R)

-subRight _L |> __int32(_) _R -> _RP
---
sub(_L, _R) -> sub(_L, _RP)

-subLeft _L -> _LP
---
sub(_L, _R) -> sub(_LP, _R)

!try sub(sub(7,1),sub(4,2))
```

When processing the term `sub(sub(7,1),sub(4,2))`, at the first step the left operand will be rewritten to an integer, then the right operand and finally the two resolved subexpressions will be handled by the base rule.

```
Step 1
  Rewrite attempt 1: sub(sub(7, 1), sub(4, 2)) ->
  -subBase _L |> _ _R |> _ --- sub(_L, _R)->__sub(_L, _R)
  -subRight _L |> _ _R->_RP --- sub(_L, _R)->sub(_L, _RP)
  -subLeft _L->_LP --- sub(_L, _R)->sub(_LP, _R)
    Rewrite attempt 2: sub(7, 1) ->
    -subBase _L |> _ _R |> _ --- sub(_L, _R)->__sub(_L, _R)
    -subBase rewrites to 6
  -subLeft rewrites to sub(6, sub(4, 2))
Step 2
  Rewrite attempt 3: sub(6, sub(4, 2)) ->
  -subBase _L |> _ _R |> _ --- sub(_L, _R)->__sub(_L, _R)
  -subRight _L |> _ _R->_RP --- sub(_L, _R)->sub(_L, _RP)
    Rewrite attempt 4: sub(4, 2) ->
    -subBase _L |> _ _R |> _ --- sub(_L, _R)->__sub(_L, _R)
    -subBase rewrites to 2
  -subRight rewrites to sub(6, 2)
```

```
Step 3
  Rewrite attempt 5: sub(6, 2) ->
  -subBase _L |> _ _R |> _ --- sub(_L, _R)->__sub(_L, _R)
  -subBase rewrites to 4
Normal termination on 4 after 3 steps and 5 rewrite attempts
```

Why are we so concerned about ordering? Well, some languages allow side effects in expressions and we need them to appear in some well-defined order (usually left to right). Examples of operators thathave side effects include ++ and −− in C and Java, and the stream output operator >> in C++.

3.8 Your first project task - fill in the operator gaps

The triad of rules (xxxBase, xxxRight and xxxLeft in that order) is the standard way of implementing arity operations in ART. Referring back to the Value table on page **??** you will find six predicate operators (such as __ne and __gt), six arithmetic operators (__add to __exp) and a variety of logic and shift operations. For each operation/type combination that you want in your langauge, you will need to create this triad of rules. That is a lot of rules - nearly 60 in fact if you want to use all of the basic operations. However each triad has the same pattern, and if you make your own constructor name the same as the builtin name without the underscores, then a little use of search and replace will enable you to build the rule set quickly.

For this week's submission, save your old `reduction.art` and make a new empty file. Then add in the necessary rules to support basic arithmetic and comparisons.

Using your rules, show a reduction trace for a term corresponding to

$$(7 - 3) * 2 > 3 + 4 + 5$$

Please do not forget to submit this week's `reduction.art` file to Moodle when you have finished, and by the deadline.

Chapter 4

Making a full language, and project demos

This laboratory session is an opportunity for you to exercise the examples that you saw in Chapter 5. You will then hybridise the two examples to create a version of the GCD language that also supports an output list. Finally, we shall look at the use of a *plugin* to give our languages access to general Java code.

4.1 Go through chapter 5 examples

Begin by looking through the contents of the ART examples directory at

https://github.com/AJohnstone2007/ART/tree/main/doc/examples

and fetch a copy of `slewaChapter5Examples.art` from https://github.com/AJohnstone2007/ART/blob/main/doc/examples/slewaChapter5Examples.art which has many of the rules from Chapter 5 already setup for you.

Your task now is simply to *play* with the rules. Run different !trys with the rules, and change the rules around. Find out what happens if you change the order of the rules in a triad and try and understand what is happening. Work methodically through the chapter, learning by doing.

4.2 GCD with output

Now take a copy of `gcdReduction.art` from https://github.com/AJohnstone2007/ART/blob/main/doc/examples/gcdReduction.art and rename it `reduction.art`; this is the file that you will submit for this week's lab work.

Your task is to extend this example so that it has an output command, and change the test program so that it writes the final GCD result on to the output list.

You will need to change the configuration, and that means every transition will have to be changed... Search and replace is your friend.

Please do not forget to submit this week's `reduction.art` file with your extended GCD language to Moodle when you have finished, and by the deadline.

Chapter 5

Plugins and external syntax

This laboratory session is in two parts: firstly we find out how to write and compile a plugin for ART that will allow your reduction interpeter to trigger actions written in Java, and then we start to develop a suitable external syntax for your project language.

The `Reduction.art` and `ARTValuePlugin.java` files that you work on today will form the nucleus of the Part A project submission you will make at the end of week 7.

5.1 Getting set up

This week we shall be compiling Java code, so you will need to have installed the Java compiler. At the command line, type

```
java --version
```

On my 2025 system, I see

```
java 17.0.12 2024-07-16 LTS
Java(TM) SE Runtime Environment (build 17.0.12+8-LTS-286)
Java HotSpot(TM) 64-Bit Server VM (build 17.0.12+8-LTS-286, mixed mode, sharing)
```

If you get an error message, or a see a version number below 17 you must install an up to date version of Java: see page **??** for instructions.

If you are intending to build a language for the image processing or solid modelling domains, you must also install an up to date version of JavaFX: see page **??** for instructions.

5.2 The plugin mechanism

When ART starts up, it searches the class path to find a class file named `ARTValuePlugin.class`. If it finds one it will load that class. If it does not find one, then it will use a plugin that is built into the `art.jar` file called the *System Default Plugin*. Either way, the plugin code will receive calls from ART when the __plugin() function is activated.

This is the same basic mechanism as with functions such as __add() and __get(): the arguments to the function are converted to their Java partner values, a Java

method is called and the return value is converted back into ART's internal form. The difference is that whilst __add() has a fixed definition and will only work with some of the ART types, __plugin() will accept any number of ART values and just convert and pass them over to your code.

5.3 Using the __plugin() function

The system default plugin just prints out the arguments that are passed to it, and returns __done, so we can use it to check that whatever protocol we choose to use for passing information across from your language to your Java backend is operating correctly. Try this example.

```
1 –plugin ––– plugin(_O) –> __plugin(_O)
2 –plugin ––– plugin(_O, _X) –> __plugin(_O, _X)
3 –plugin ––– plugin(_O, _X, _Y) –> __plugin(_O, _X, _Y)
4
5 //!try plugin("OperationName")
6 //!try plugin("SomeOtherOperation", 23.4)
7 !try plugin("OperationWithList", [1,2,3], 666)
```

Here we have defined three rules and three corresponding !trys. Each rule simply rewrites the constructor plugin to the function call __plugin just as we do for, say addition. A useful convention is to pass an *opcode* as the first argument, and any arguments needed by the opcode in the following arguments. Since ART reduction rules are fixed arity, we need a rule for each arity we want to use. However, we can use collection types such as __list to build up collections of arguments.

If we run this example, we get this:

```
1  Step 1
2    Rewrite attempt 1: plugin("OperationWithList", [1, 2, 3], 666) ->
3    -plugin --- plugin(_O)->, __plugin(_O)
4    -plugin Theta match failed: seek another rule
5    -plugin --- plugin(_O, _X)->, __plugin(_O, _X)
6    -plugin Theta match failed: seek another rule
7    -plugin --- plugin(_O, _X, _Y)->, __plugin(_O, _X, _Y)
8  Plugin called with 3 arguments
9     class java.lang.String OperationWithList
10    class java.util.LinkedList [1, 2, 3]
11    class java.lang.Integer 666
12   -plugin rewrites to __done
13 Normal termination on __done after 1 step and 1 rewrite attempt
```

Lines 2–7 show the rewriter working through the rules until it finds one that matches the !try term. Lines 8–11 show the output from the system default plugin: each argument is listed with its Java type and its value.

5.4 Using the generic example plugin

Go to your ***artwork*** directory. Rename any previous version of your `Reduction.art` file that you want to keep - we are going to download a new one. Make sure that you have a copy of `art.jar` in the directory.

Use a web browser to access the ART `project` directory at https://github.com/AJohnstone2007/ART/tree/main/doc/project. You will see four subdirectories: `generic, music, imageProcessing` and `solidModelling`.

Go into the `generic` directory and down load copies of `ARTValuePlugin.java` and `Reduction.art` from there.

If you are running on Windows, download a copy of `runReduction.bat`

If you are running on Unix, download a copy of `runReduction.sh`. **Important: make it executable by typing** chmod +x runReduction.sh

Running examples

On Windows, at the command line type

```
1 runReduction
```

On Unix-style operating systems, at the shell prompt type

```
1 ./runReduction.sh
```

If you see this message

./runReduction.sh: Permission denied

then you have omitted to make the Un*x shell script executable: type chmod +x runReduction.sh

If all is well, you will see the following. If not, read the instructions again carefully, and if that doesn't work then ask for help.

```
1  *** Value system attached to Adrian's generic example plugin
2 Step 1
3   Rewrite attempt 1: <plugin("init"), {=}> ->
4   -plugin --- <plugin(_0), _sig>->, <__plugin(_0), _sig>
5 Plugin initialised
6   -plugin rewrites to <__done, {=}>
7 Normal termination on <__done, {=}> after 1 step and 1 rewrite attempt
```

At line 1, the message tells us that ART has found the plugin in this directory and connected to it rather than to the system default plugin; the script calls the initialisation function in the plugin.

5.5 Inside the generic plugin example

ART Value plugin classes must be called `ARTValuePlugin` and they must extend `AbstractValuePlugin` which requires them to have two methods:

public String description() and public Object plugin(Object... args)

The description() method simply returns an identification string which is printed out by ART when it connects to a plugin. It can get quite confusing ifyou have different plugins in different work areas, and this description string is a way of keeping track.

The plugin() method receives the arguments from a call to the ART function __plugin after they have been converted to Java values. Inside the plugin() method, you can do whatever you like. A common convention is to pass a string in the first argument specifying the operation to be performed, and then use a switch statement to select the right action. Here is the complete source code for the generic `ARTValuePlugin` example.

```
1  import uk.ac.rhul.cs.csle.art.term.AbstractValuePlugin;
2  import uk.ac.rhul.cs.csle.art.util.Util;
3
4  public class ARTValuePlugin extends AbstractValuePlugin {
5
6    @Override
7    public String description() {
8      return "Adrian's generic example plugin";
9    }
10
11   @Override
12   public Object plugin(Object... args) {
13     switch ((String) args[0]) {
14     case "init":
15       System.out.println("Plugin initialised");
16       return __done;
17
18     case "type":
19       for (var i : args)
20         System.out.println(i);
21       return __done; // Command returns __done
22
23     case "invert":
24       return -(Integer) args[1]; // Expression operator returns value
25
26     default:
27       Util.fatal("Unknown plugin operation: " + args[0]);
28       return __bottom;
29     }
```

```
30      }
31  }
```

This particular plugin offers three operations: init, type and invert. Any other operation code generated a fatal error via the default: clause.

Since the arguments to plugin() are of typObject, we need to cast them when we use them. The invert operation illustrates this: args[1] is cast to an Integer, negated and returned.

5.6 Inside the generic Reduction.art

The `Reduction.art` file that you have just downloaded is the same GCD language we have seen before, but extended with some reduction rules at lines 32–38 that call the __plugin function, and with some CFG rules at lines 45-46 (whichare used on line 41) to provide some external syntax that activates the rewrite rules.

```
1  (* GCD language with extra rules for plugin *)
2
3  !configuration -> _sig:__map
4
5  (* Term rewrite rules *)
6  -assign _n |> __int32(_) --- assign(_X,_n),_sig -> __done,__put(_sig,_X,_n)
7  -assignR _E,_sig -> _I,_sigP --- assign(_X,_E),_sig -> assign(_X,_I),_sigP
8
9  -deref --- deref(_R),_sig -> __get(_sig,_R),_sig
10
11 -sequenceDone --- seq(__done, _C),_sig -> _C,_sig
12 -sequence _C1,_sig -> _C1P,_sigP --- seq(_C1,_C2),_sig -> seq(_C1P,_C2),_sigP
13
14 -ifTrue --- if(true,_C1,_C2),_sig -> _C1,_sig
15 -ifFalse --- if(false,_C1,_C2),_sig -> _C2,_sig
16 -ifResolve _E,_sig ->_EP,_sigP --- if(_E,_C1,_C2),_sig -> if(_EP,_C1,_C2),_sigP
17
18 -while --- while(_E,_C),_sig -> if(_E,seq(_C,while(_E,_C)),__done),_sig
19
20 -gt _n1 |> __int32(_) _n2 |> __int32(_) --- gt(_n1,_n2),_sig -> __gt(_n1,_n2),_sig
21 -gtR _n |> __int32(_) _E2,_sig -> _I2,_sigP --- gt(_n,_E2),_sig -> gt(_n,_I2),_sigP
22 -gtL _E1,_sig -> _I1,_sigP --- gt(_E1,_E2),_sig -> gt(_I1,_E2),_sigP
23
24 -ne _n1 |> __int32(_) _n2 |> __int32(_) --- ne(_n1,_n2),_sig -> __ne(_n1,_n2),_sig
25 -neR _n |> __int32(_) _E2,_sig -> _I2,_sigP --- ne(_n,_E2) ,_sig -> ne(_n,_I2),_sigP
26 -neL _E1,_sig -> _I1,_sigP --- ne(_E1,_E2),_sig -> ne(_I1,_E2),_sigP
27
28 -sub _n1 |> __int32(_) _n2 |> __int32(_) --- sub(_n1,_n2),_sig -> __sub(_n1,_n2),_sig
29 -subR _n |> __int32(_) _E2,_sig -> _I2,_sigP --- sub(_n,_E2),_sig -> sub(_n,_I2),_sigP
```

```
−subL _E1,_sig −> _I1,_sigP −−− sub(_E1,_E2),_sig −> sub(_I1,_E2),_sigP

−plugin −−− plugin(_O),_sig −> __plugin(_O),_sig
−plugin −−− plugin(_O, _X),_sig −> __plugin(_O, _X),_sig
−plugin −−− plugin(_O, _X, _Y),_sig −> __plugin(_O, _X, _Y),_sig
−plugin −−− plugin(_O, _X, _Y, _Z),_sig −> __plugin(_O, _X, _Y, _Z),_sig

−type −−− type(_A, _B, _C),_sig −> __plugin("type", _A, _B, _C),_sig

(* CFG rules *)
seq ::= statement^^ | statement seq
statement ::= assign^^ | while^^ | if^^ | plugin^^ ';'^ | type^^ ';'^
assign ::= &ID ':='^ expression ';'^
while ::= 'while'^ expression 'do'^ statement
if ::= 'if'^ expression 'then'^ statement | 'if'^ expression 'then'^ statement 'else'^ statement
plugin ::= 'plugin'^ '('^ expressions^ ')'^
type ::= 'type'^ __string ','^ __string ','^ __string
expressions ::= expression | expression ','^ expressions^
expression ::= rels^^
rels ::= adds^^ | gt^^ | ne^^
   gt ::= adds '>'^ adds
   ne ::= adds '!='^ adds
adds ::= operand^^ | sub^^ | add^^
   add ::= adds '+'^ operand
   sub ::= adds '−'^ operand
operand ::= __int32^^ | deref^^ | __string^^ | plugin^^
__int32 ::= &INTEGER
deref ::= &ID
__string::= &STRING_SQ

(* GCD examples *)
//!try seq(assign(a,6), seq(assign(b,9), while(ne(deref(a), deref(b)), if(gt(deref(a), deref(b)),
// assign(a, sub(deref(a), deref(b))), assign(b, sub(deref(b), deref(a)))))), __map
//!try "a := 6; b := 9; while a != b do if a > b then a := a − b; else b := b − a;"

(* Plugin test *)
!try "plugin('init');"
//!try "plugin('init'); plugin('type', 'B', 'A', 'D');"
//!try "plugin('init'); type 'B', 'A', 'D';"
//!try "plugin('init'); plugin('invert', 31);"

//!try "plugin('weird', 666); "
```

5.7 Your exercise

Add a plugin operation (and suitable external syntax) that takes two __int32 numbers and returns the largest integer that is one less than both of them. Write two !trys that use this operation, one using terms and one using external syntax.

Please do not forget to submit this week's `reduction.art` file to Moodle when you have finished, and by the deadline.

5.8 Project selection and demos

By now you should have selected a domain to work in and made a list of types and operation signatures that you need to implement your ideas.

Along side the `project/generic` directory you will find three directories, one for each project domain. Each contains a plugin and a `Reduction.art` to get you started which I will demonstrate to you; in particular I will explain how to create external syntax for your operations.

Your *summative* assessment task, due at the end of week 7, is to extend these demonstration samples. Read Section **??** to understand the requirements, and to get ideas for extensions that you can add for the final, part B submission.

Next week's lab session (week 6) is a support session for the project: there will not be a lab worksheet, and no submission. I will be available to help you work on your Part A submission.

Chapter 6

SOBRD parsing and attribute evaluation

This laboratory session introduces you to the internals of the RDSOB parsers. RDSOB is a very poor parsing algorithm, but it useful as a teaching aid because it is very simple and can still do useful work. In particular, we shall see how to use synthesized and inherited attributes with actions to make specify semantics, and understand how they work.

It is important that for each example, you examine the generated parser code which ART writes into `ARTGeneratedRDSOBOracle.java`

6.1 Checking your Java installation

This week we shall be compiling Java code, so you will need to have installed the Java compiler. At the command line, type

```
java ––version
```

On my 2025 system, I see

```
java 17.0.12 2024-07-16 LTS
Java(TM) SE Runtime Environment (build 17.0.12+8-LTS-286)
Java HotSpot(TM) 64-Bit Server VM (build 17.0.12+8-LTS-286, mixed mode, sharing)
```

If you get an error message, or a see a version number below 17 you must install an up to date version of Java: see page **??** for instructions.

6.2 Getting started

Download an up-to-date copy of art.jar from
https://github.com/AJohnstone2007/ART/blob/main/art.jar

Go to
https://github.com/AJohnstone2007/ART/tree/main/doc/examples/lab6
and download `slewaChapter6RDSOBExamples.art`, `test.str`.

If you are running on Windows, download `runRDSOB.bat` and type

```
runRDSOB
```

If you are running on UN*x or MaxOS, download `runRDSOB.sh` and type

```
chmod +x runRDSOB.sh
dos2unix runRDSOB.sh
./runRDSOB.sh
```

You should see the following output

```
*** Value system attached to System default plugin
Writing new ARTGeneratedRDSOBOracle: 2025-03-06 06:34:55

C:\adrian\eclipse\art\student>javac -cp .;art.jar ARTGeneratedRDSOBOracle.java

C:\adrian\eclipse\art\student>java -cp .;art.jar ARTGeneratedRDSOBOracle test.str
Input: a@
Accepted
Oracle: 2 2
Semantics:
```

If not, first try the instructions again and then ask for help.

6.3 First example

Our first example is the small grammar from the lectures. Open a text editor on file `slewaChapter6RDSOBExamples.art`. You will see:

```
///////// Example 1 – a first RDSOB example: bx*@

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

// test.str examples: b, a@, ax@, axx@

(* Omitted examples ...*)

!generate rdsobOracle
```

The two CFG rules are from the lectures. The directive `!generate rdsobOracle` tells ART to construct an RDSOB parser which is written out to the file `ARTGeneratedRDSOBOracle.java`.

Now look at the run script in `runRDSOB.bat` or `runRDSOB.sh`. It contains:

```
java –jar art.jar slewaChapter6RDSOBExamples.art
javac –cp .;art.jar ARTGeneratedRDSOBOracle.java
java –cp .;art.jar ARTGeneratedRDSOBOracle test.str
```

Line 1 runs ART on `slewaChapter6RDSOBExamples.art` to create a new parser source file `ARTGeneratedRDSOBOracle.java`.

Line 2 compiles that Java file with a class path `-cp` that includes `art.jar` so that the compiler can find the superclass.

Line 3 runs the resulting compiled code, with the parser taking its input from `test.str`.

Now open a text editor on file `test.str`. Edit the contents of the file, and then parse it by typing `runRDSOB` (Windows) or `./runRDSOB.sh` (Un*x).

Try each ofthese `test.str` examples in turn:

```
b
```

```
a@
```

```
ax@
```

```
axx@
```

```
bx
```

The first four are accepted; the last one is rejected.

Now work through the examples in `slewaChapter6RDSOBExamples.art` trying different strings, and looking at the generated parser source code.

In particular, pay attention to the subtraction tests in Example 8, and try and write some grammar rules that show how RDSOB fails to correctly process some inputs.

Please do not forget to submit this week's `reduction.art` file to Moodle when you have finished, and by the deadline.

Chapter 7

Attribute-Action interpreters with flow control

This final laboratory session introduces you to using attributes in the full version of the ART parser, and gets you started with `Attribute.art`, the base template for part B of your project. In particular, you will see how *delayed attributes* are used to implement flow control in Attribute Action intepreters.

7.1 Checking your Java installation

This week we shall be compiling Java code, so you will need to have installed the Java compiler. At the command line, type

```
java ––version
```

On my 2025 system, I see

```
java 17.0.12 2024-07-16 LTS
Java(TM) SE Runtime Environment (build 17.0.12+8-LTS-286)
Java HotSpot(TM) 64-Bit Server VM (build 17.0.12+8-LTS-286, mixed mode, sharing)
```

If you get an error message, or a see a version number below 17 you must install an up to date version of Java: see page **??** for instructions.

7.2 Getting started

Important: make a new, empty working directory for this lab so that we do not accidentally overwrite your plugin.

Download an up-to-date copy of art.jar from
https://github.com/AJohnstone2007/ART/blob/main/art.jar

Go to
https://github.com/AJohnstone2007/ART/tree/main/doc/project/generic
and download `Attribute.art` and `ARTValuePlugin.java`.

If you are running on Windows, download `runAttribute.bat` and type

```
runAttribute.bat
```

If you are running on UN*x or MaxOS, download `runAttribute.sh` and type

```
chmod +x runAttribute.sh
dos2unix runAttribute.sh
./runAttribute.sh
```

You should see the following output

```
*** Value system attached to Adrian's generic example plugin
Accept
Writing new ARTGeneratedActions: 2025-03-12 22:11:15

> javac -cp .;art.jar ARTGeneratedActions.java

> java -cp .;art.jar uk.ac.rhul.cs.csle.art.ART !interpreter attributeAction Attribute.art
*** Value system attached to Adrian's generic example plugin
*** Attached to ARTGeneratedActions 2025-03-12 22:11:15
Accept
Plugin initialised
Final variable map {a=3, b=3, x=-31, gcd=3}
```

If not, first try the instructions again and then ask for help.

7.3 The project Part B template

The file `Attribute.art` is the starting point for part B of the project. It is an implementation of the GCD language using Attribute Action interpretation.

```
(* gcdAttribute.art GCD in attribute−actions with native Java actions *)
(* Use ofplugin in expressions requires return result that can be cast to int *)
//!interpreter attributeAction
!support
!! import java.util.Map; import java.util.HashMap; !!
!! Map<String, Integer> variables = new HashMap<>();
    Map<String, AbstractAttributeBlock> procedures = new HashMap<>(); !!

statements ::= statement statements
| statement !! System.out.println("Final variable map " + variables); !!

statement ::=
   ID ':=' e0 ';' !! variables.put(ID1.v, e01.v); !! // assignment
| 'if' e0 'then' statement!< 'else' statement!< // if statement
   !! if (e01.v != 0) interpret(statement1); else interpret(statement2); !!
| 'while' e0!< 'do' statement!< // while statement
   !! interpret(e01); while (e01.v != 0) { interpret(statement1); interpret(e01); } !!
| 'plugin' '(' STRING_SQ ')' ';' !! plugin(STRING_SQ1.v); !!
| 'plugin' '(' STRING_SQ ',' e0 ')' ';'!! plugin(STRING_SQ1.v, e01.v); !!
| 'plugin' '(' STRING_SQ ',' e0 ',' e0 ')' ';' !! plugin(STRING_SQ1.v, e01.v, e02.v); !!
```

```
| 'plugin' '(' STRING_SQ ',' e0 ',' e0 ',' e0 ')' ';' !! plugin(STRING_SQ1.v, e01.v, e02.v, e03.v); !!

e0 ::= e1 !! e0.v = e11.v; !!
| e1 '>' e1 !! e0.v = e11.v > e12.v ? 1 : 0; !! // Greater than
| e1 '!=' e1 !! e0.v = e11.v != e12.v ? 1 : 0; !! // Not equal to

e1 ::= e2 !! e1.v = e21.v; !!
| e1 '-' e2 !! e1.v = e11.v - e21.v; !! // Subtract

e2 ::= INTEGER !!e2.v = INTEGER1.v; !! // Integer literal
| ID !! e2.v = variables.get(ID1.v); !! // Variable access
| '(' e1 !! e2.v = e11.v; !! ')' // Parenthesised expression
| 'plugin' '(' STRING_SQ ',' e0 ')' !! e2.v = (int) plugin(STRING_SQ1.v, e01.v); !!

// Lexical rules
ID <v:String> ::= &ID !! ID.v = lexeme(); !!
STRING_SQ <v:String> ::= &STRING_SQ !! STRING_SQ.v = lexemeCore().translateEscapes(); !!
INTEGER ::= &INTEGER !! INTEGER.v = Integer.parseInt(lexeme()); !!

//!generate actions
!try "a := 6; b := 9; while a != b do if a > b then a := a - b; else b := b - a; gcd := a;
            plugin('init'); x:= plugin('invert', 31);"
```

We shall discuss this, and then you should start to incorporate the syntax from your Reduction.art language and connect to yourARTValuePlugin.java

7.4 Mini

Mini is a tiny language with some useful features that will help you understand how to implement programming language elements in an Attribute Action interpreter. Yous hould study the specifications of the various Mini sublanguages to understand how they work.

Go to `https://github.com/AJohnstone2007/ART/tree/main/doc/examples/mini` and download the files there. Y

If you are using Windows, you run the first example by typing

```
runMini minisyntax.art
```

If you are using a Un*x-like operating system, you run the first example by typing

```
./runMini.sh minisyntax.art
```

Don't forget to make the shell script executable before you run it the first time using chmod +x runMini.sh

minisyntax

```
(*****************************************************************************
*
* miniSyntax.art for ART V5 – Adrian Johnstone 31 December 2024
*
*****************************************************************************)
statement ::= 'print' '(' printElements ')' ';'

printElements ::=
    STRING_SQ
  | STRING_SQ ',' printElements
  | e0
  | e0 ',' printElements

e0 ::=
    e1
  | e1 '>' e1
  | e1 '<' e1
  | e1 '>=' e1
  | e1 '<=' e1
  | e1 '==' e1
  | e1 '!=' e1

e1 ::=
    e2
  | e1 '+' e2
  | e1 '-' e2

e2 ::=
    e3
  | e2 '*' e3
  | e2 '/' e3
  | e2 '%' e3

e3 ::=
    e4
  | '+' e3
  | '-' e3

e4 ::=
    e5
  | e5 '**' e4

e5 ::=
    INTEGER
```

```
45 | '(' e1 ')'
46
47 (* Lexical productions *)
48 STRING_SQ ::= &STRING_SQ
49 INTEGER ::= &INTEGER
50
51 !try "print('Result: ', 3+4, '\n');"
```

minicalc

```
1  (*****************************************************************************
2  *
3  * miniCalc.art for ART V5 – Adrian Johnstone 31 December 2024
4  *
5  *****************************************************************************)
6  statement ::= 'print' '(' printElements ')' ';' // print statement
7
8  printElements ::=
9     STRING_SQ !! System.out.print(STRING_SQ1.v); !!
10 | STRING_SQ !! System.out.print(STRING_SQ1.v); !! ',' printElements
11 | e0 !! System.out.print(e01.v); !!
12 | e0 !! System.out.print(e01.v); !! ',' printElements
13
14 e0 <v:int> ::=
15    e1 !! e0.v = e11.v; !!
16 | e1 '>' e1 !! e0.v = e11.v > e12.v ? 1 : 0; !! // Greater than
17 | e1 '<' e1 !! e0.v = e11.v < e12.v ? 1 : 0; !! // Less than
18 | e1 '>=' e1 !! e0.v = e11.v >= e12.v ? 1 : 0; !! // Greater than or equals
19 | e1 '<=' e1 !! e0.v = e11.v <= e12.v ? 1 : 0; !! // Less than or equals
20 | e1 '==' e1 !! e0.v = e11.v == e12.v ? 1 : 0; !! // Equal to
21 | e1 '!=' e1 !! e0.v = e11.v != e12.v ? 1 : 0; !! // Not equal to
22
23 e1 <v:int> ::=
24    e2 !! e1.v = e21.v; !!
25 | e1 '+' e2 !! e1.v = e11.v + e21.v; !! // Add
26 | e1 '-' e2 !! e1.v = e11.v - e21.v; !! // Subtract
27
28 e2 <v:int> ::=
29    e3 !! e2.v= e31.v; !!
30 | e2 '*' e3 !! e2.v = e21.v * e31.v; !! // Multiply
31 | e2 '/' e3 !! e2.v = e21.v / e31.v; !! // Divide
32 | e2 '%' e3 !! e2.v = e21.v % e31.v; !! // Mod
33
34 e3 <v:int> ::=
35    e4 !!e3.v = e41.v; !!
```

```
| '+' e3 !!e3.v = e31.v; !! // Posite
| '−' e3 !!e3.v = −e31.v; !! // Negate

e4 <v:int> ::=
   e5 !! e4.v = e51.v; !!
| e5 '**' e4 !! e4.v = (int) Math.pow(e51.v, e41.v); !! // Exponentiate

e5 <v:int> ::=
   INTEGER !!e5.v = INTEGER1.v; !! // Integer literal
| '(' e1 !! e5.v = e11.v; !! ')' // Parenthesised expression

(* Lexical productions *)
STRING_SQ <v:String> ::= &STRING_SQ !! STRING_SQ.v = lexemeCore().translateEscapes(); !!
INTEGER <v:int> ::= &INTEGER !! INTEGER.v = Integer.parseInt(lexeme()); !!

!try "print('Result: ', 3+4, '\n');"
```

minicall

```
(*****************************************************************************
*
* minicall.art for ART V5 − Adrian Johnstone 16 January 2025
*
*****************************************************************************)
!support
!! import java.util.Map; import java.util.HashMap; !!
!! Map<String, Integer> variables = new HashMap<>();
    Map<String, AbstractAttributeBlock> procedures = new HashMap<>(); !!

text ::= statements !! System.out.println("Final variable map " + variables); !!
statements ::= statement | statement statements

statement ::=
   ID '=' e0 ';' !! variables.put(ID1.v, e01.v); !! (* assignment *)

| 'if' e0 'then' statement!< elseOpt!< (* if statement *)
   !! if (e01.v != 0) interpret(statement1);
       else interpret(elseOpt1); !!

| 'while' e0!< 'do' statement!< (* while statement *)
   !! interpret(e01);
       while (e01.v != 0) {
           interpret(statement1);
           interpret(e01);
        } !!
```

```
27
28 | 'print' '(' printElements ')' ';' (* print statement *)
29
30 | 'procedure' ID statement!< !! procedures.put(ID1.v, statement1); !!
31 | 'call' ID ';' !! interpret(procedures.get(ID1.v)); !!
32
33 | '{' statements '}' (* compound statement *)
34
35 elseOpt ::= 'else' statement | #
36
37 printElements ::=
38    STRING_SQ !! System.out.print(STRING_SQ1.v); !!
39 | STRING_SQ !! System.out.print(STRING_SQ1.v); !! ',' printElements
40 | e0 !! System.out.print(e01.v); !!
41 | e0 !! System.out.print(e01.v); !! ',' printElements
42
43 e0 ::=
44    e1 !! e0.v = e11.v; !!
45 | e1 '>' e1 !! e0.v = e11.v > e12.v ? 1 : 0; !! (* Greater than *)
46 | e1 '<' e1 !! e0.v = e11.v < e12.v ? 1 : 0; !! (* Less than *)
47 | e1 '>=' e1 !! e0.v = e11.v >= e12.v ? 1 : 0; !! (* Greater than or equals*)
48 | e1 '<=' e1 !! e0.v = e11.v <= e12.v ? 1 : 0; !! (* Less than or equals *)
49 | e1 '==' e1 !! e0.v = e11.v == e12.v ? 1 : 0; !! (* Equal to *)
50 | e1 '!=' e1 !! e0.v = e11.v != e12.v ? 1 : 0; !! (* Not equal to *)
51
52 e1 ::=
53    e2 !! e1.v = e21.v; !!
54 | e1 '+' e2 !! e1.v = e11.v + e21.v; !! (* Add *)
55 | e1 '-' e2 !! e1.v = e11.v - e21.v; !! (* Subtract *)
56
57 e2 ::=
58    e3 !! e2.v= e31.v; !!
59 | e2 '*' e3 !! e2.v = e21.v * e31.v; !! (* Multiply *)
60 | e2 '/' e3 !! e2.v = e21.v / e31.v; !! (* Divide *)
61 | e2 '%' e3 !! e2.v = e21.v % e31.v; !! (* Mod *)
62
63 e3 ::=
64    e4 !!e3.v = e41.v; !!
65 | '+' e3 !!e3.v = e31.v; !! (* Posite *)
66 | '-' e3 !!e3.v = -e31.v; !! (* Negate *)
67
68 e4 ::=
69    e5 !! e4.v = e51.v; !!
70 | e5 '**' e4 !!e4.v = (int) Math.pow(e51.v, e41.v); !! (* exponentiate *)
71
72 e5 ::=
73    INTEGER !!e5.v = INTEGER1.v; !! (* Integer literal *)
```

```
74 | ID !! e5.v = variables.get(ID1.v); !! (* Variable access *)
75 | '(' e1 !! e5.v = e11.v; !! ')' (* Parenthesised expression *)
76
77 (* Lexical productions *)
78 ID <v:String> ::= &ID !! ID.v = lexeme(); !!
79 STRING_SQ <v:String> ::= &STRING_SQ !! STRING_SQ.v = lexemeCore().translateEscapes(); !!
80 INTEGER ::= &INTEGER !! INTEGER.v = Integer.parseInt(lexeme()); !!
81
82 //!try "print('Result: ', (3+4)*2, '\n');"
83 //!try "x = 3;"
84
85
86 !try "
87 {
88 procedure sub { print('Hello from a procedure\n'); }
89
90
91 x = 3;
92 while x > 0 do { print('x is ', x, '\n'); x = x -1; }
93
94 call sub;
95 }
96 "
```

Please do not forget to submit this week's `Attribute.art` file to Moodle when you have finished, and by the deadline.