Software Language Engineering with ART

Adrian Johnstone
a.johnstone@rhul.ac.uk

January 20, 2025

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

Contents

Chapter 1

Introduction

Software Language Engineering concerns the design and implementation of compilers, interpreters, source-to-source translators and other kinds of programming language processors.

All forms of engineering are a mixture of creative insight and disciplined implementation. For instance, the architect of a bridge relies on structural engineers who can take a high level design and perform detailed calculations on that structure to test whether it will withstand daily use.

Ideally, this would be true for software too: our creativity would be expressed only through sound and principled techniques; that is techniques that have been found to be safe and efficient using mathematical and other forms of analysis.

In practice we mostly write software in a *hopeful* way, and then use testing to try and find the gaps in our understanding. Unfortunately, programming languages are inherently difficult to test since they are designed to be flexible notations with very many combinations of interacting features and in any case, as Edsger Dikstra famously noted, *Program testing can be used to show the presence of bugs, but never to show their absence.* [**?**, p6].

Our goal is the construction of *maintainable* tools which have concise specifications that are amenable to automated checking, and which allow automatic generation of the tool itself. By working at a high level, we hope to reduce implementation errors, just as high level programming languages with static type checking can catch many of the errors that arise when programming at machine level.

We emphasise maintainability because language processors are typically complex with many internal dependencies, and are thus fragile in the face of attempts to extend or modify them. High level programming language specifications in widely understood notations would make it much easier to extend and modify those specifications so that other engineers could both maintain and reuse our work into the future.

1.1 The classical approach to translation

Most programming language processors are built around the notion of these five classical *phases*: Lex – Parse – Analyse – Rework – Perform which together check the input source text, and then perform the actions specified therein.

phases

The source program text to be translated into actions is simply a string of characters. In the Lex phase, this string is partitioned into a sequence of substrings called *lexemes*, and the resulting sequence of lexemes is passed to the parser. Typically these lexemes comprise a single identifier, keyword or constant. In free-format languages, whitespace and comments are usually discarded by the lexer and do not appear in the lexeme sequence.

lexemes

The purpose of the Parse phase is to (a) check that the lexemes appear in an order that is allowed by the rules of the of the language, and (b) to build a *derivation* which shows how the lexeme string can be constructed from the language rules. For programming languages these rules are usually specified by string rewrite rules which together form a *grammar*. Together, the Lex and Parse phases are often referred to as the *front end*.

derivation

grammar

front end

The Analyse phase constructs an internal representation of the source program in a form that supports the later phases, and checks certain long range properties, such as whether the type declaration of a variable matches its subsequent usages. Typically a *symbol table* is also constructed which lists identifiers and their role in the source program.

symbol table

The resulting representation is often called an *Abstract Syntax Tree* (AST) but there is no general agreement on what should be in such a tree and what supporting structures should be provided: indeed complex compilers such as the GNU suite often have a variety of different internal representations which are used to support different facets of the compilation process, and not all of these representations are tree-like. As a result, in our work we avoid the term AST and instead refer to internal representations as *internal syntax* as opposed to the original form of the program which we call the *external syntax*

internal syntax

external syntax

The Rework phase iteratively modifies the internal representation, usually in an attempt to improve performance. For instance constant expressions inside loops may be moved so that they are evaluated once before the loop is entered, rather then being recomputed on every loop iteration. To be correct, the rework must not change the meaning of a program; establishing that correctness is hard. This phase is traditionally called an *optimiser* but we avoid that term because in general the resulting code is rarely optimal: indeed different kinds of rework can cancel each other out and in extreme cases actually reduce performance.

The Perform phase traverses the final form of the internal representation and performs the specified actions. In a *compiler* the actions output a machine level program which can be subsequently executed on some processor architecture. In an *interpreter* the Perform phase directly executes the actions itself. Compiled code is usually faster than interpreted code, but a good compiler requires much more engineering effort than an interpreter. Some systems blur the line between the two by, perhaps, starting execution in an interpreted mode but then pausing to compile frequently-executed code to native machine language which can then execute at full speed. The Perform phase is often called the *back end* and we may refer to the combined Analyse and Rework phases as the *middle end*.

compiler

interpreter

back end

middle end

We should note that although these five independent phases are a very useful way to *think* about the various tasks a production-quality compiler must perform, in a real translator they may be intertwined, or even absent. Simple translators may not have a Rework phase, and some parsing techniques are effective when the lexemes are just the individual characters in the source program: such *character level parsers* do not have a Lex phase.

Classical front end techniques

The years 1960–75 represent a golden age of research into front end techniques: Donald Knuth noted that *'compiler research was certainly intensive, representing roughly one third of all computer science in the 1960s'*[**?**, p.46].

The goal of that research was to balance utility with efficiency: Alfred Aho characterised part of the work as *Searching for a class of grammars that was big enough to describe the syntactic constructs that you were interested in, yet restricted enough that you could construct efficient parsers from it.'*[**?**, p.42].

This work coalesced around two classical approaches: (i) limited bottom-up parsing, in particular the YACC parser generator and its descendants such as Bison, and (ii) limited top-down parsing implemented using recursive descent. Typical bottom up parser generators are implementations of the theory of shift-reduce parsing based on LR tables; top-down parsers embody the theory of predictive LL parsing. There are many alternative and hybrid approaches, and a vast research literature.

We call these 'limited' parsing techniques to highlight the constraints that they demand of the language designer, who must massage their language specification into forms that are acceptable to these non-general algorithms. Once that is achieved, both approaches offer linear processing times (that is the processing time is simply proportional to the length of the input) and both approaches are sufficiently frugal in their use of memory that they were practical on 1970s computers with storage limited to a few tens of thousands of bytes.

Our approach emphasises *general* parsing which allows the language designer much more freedom. We shall return to this topic in section 1.6.

Classical approaches to describing languages

The convergence of theoretical analysis and engineering practice in the classical Lex and Parse phases represents a major (possibly *the* major) achievement of the first thirty years of Computer Science. Fifty years later, the lack of an agreed way to concisely and precisely specify the actions of a programming language in the Analyse, Rework and Effect phases is a continuing concern.

How are programming languages defined in current practice? Most widely used

programming languages have a 'standard': a document that describes the effects that should be induced by the phrases of the programming language. For instance, here is an extract from the Pascal report

9.2.2.1. If statements.

The if statement specifies that the statement following the symbol then be executed only if the Boolean expression yields true.

IfStatement = "if" BooleanExpression "then" Statement

Here is an extract from one of the *Java Language Specification* documents:

14.9. The if Statement

The if statement allows conditional execution of a statement.

IfThenStatement:
if (*Expression*) *Statement*

The Expression must have type boolean or Boolean, or a compile-time error occurs.

14.9.1. The if-then Statement

An if-then statement is executed by first evaluating the Expression.

If evaluation of the Expression completes abruptly for some reason, the if-then statement completes abruptly for the same reason.

Otherwise, execution continues by making a choice based on the resulting value:

(a) If the value is true, then the contained Statement is executed; the if-then statement completes normally if and only if execution of the Statement completes normally.

(b) If the value is false, no further action is taken and the if-then statement completes normally.

And here is a corresponding extract from one of the draft ANSI-C standards:

6.8.4 Selection statements

Syntax

selection-statement:
if (*expression*) *statement*

Semantics

A selection statement selects among a set of statements depending on the value of a controlling expression.

A selection statement is a block whose scope is a strict subset of the scope of its enclosing block. Each associated substatement is also a block whose scope is a strict subset of the scope of the selection statement.

6.8.4.1 The if statement

Constraints

The controlling expression of an if statement shall have scalar type.

Semantics

The substatement is executed if the expression compares unequal to 0.

All three extracts describe the same language feature: the simple conditional

statement.

If we write in C or Java the program fragment

```
z = 0; if (x == y) z = 3;
```

then we expect that after execution the variable `z` will hold the value 3 only if the variables `x` and `y` hold the same value. We get the same effect in Pascal by writing

```
z := 0; if x = y then z := 3;
```

From these examples we can deduce the following.

- ◇ All three languages have conditional statements that 'do the same thing' in some sense;
- ◇ The textual form of the program fragments for Java and C are the same, but the Pascal fragment has a different form, even though the effect is the same for all three languages.

In programming languages, the meaning of a fragment is best thought of as its effect on the *state* of the computer, where the state is the set of values maintained by a program, which could simply be the contents of the computer's memory along with any changes to the computer's input and output devices.

state

The written form of a programming language fragment is called its *syntax* and the effect on the system's state its *semantics*. The meaning of a program is the accumulated semantics of its phrases; the semantics of a programming language is the accumulated semantics of every phrase that could ever be written in that language.

syntax

semantics

Informal and formal semantics

The extracts above from Pascal, C and Java language standards are all trying to explain the syntax and semantics of a conditional statement. In each case the semantics is described in careful English prose (what we might call 'legalistic' English) but with differing levels of detail. Nearly all programming language standards adopt this approach, but that is problematic because it is hard to check that a prose specification is complete (that is, there aren't any special cases that have been left undefined) and consistent (that is, that there aren't any conflicting statements). It is even harder to check that a programming language processor, such as the Java compiler, correctly implements all aspects of the standard. Since so much of modern life is mediated by software written in programming languages, this vagueness in the underlying specification of languages and their implementations is worrying.

In an ideal world, we would have a commonly understood concise notation for describing the semantics of a language fragment with which we could construct arguments for the completeness and consistency of a programming language standard, and which we could use to check the correctness of compilers, interpreters and other language processors perhaps using a computer itself to do the checking. A semantics described this way would be called a *formal semantics*.

formal semantics

In fact such notations do exist, but they have not been widely adopted by programming language designers, for perhaps two reasons: firstly they are conventionally presented in a mathematical style that deters many software practitioners; and secondly complete language descriptions are very dense and can be quite long. Now, the second reason is not a very good one, because legalistic English prose descriptions of semantics are also very dense and long: the Java Language Specification for Java 22 runs to 876 pages.

The defining quality of a formal semantics is that it should in some sense be *mechanical*, that is: it should be amenable to implementation as a manual procedure that could be followed without insight or thought, or as a computer program. The legalistic English in the extracts above does not meet that criterion because English is itself open to interpretation. We do not want to read the semantics for a language feature and then have to argue about what the semantics itself means – that is simply to move the problem on one level!

1.2 Early compilers and modern challenges

For many languages, classical parser generators can automatically produce front ends, Most of these tools incorporate some facility for triggering *ad hoc* side effects during parsing which allows simple translators to be constructed in a style called *Syntax Directed Translation*. For instance, many early compilers for the Pascal programming language were little more than a Parse phase that extended a symbol table in response to each declaration, and emitted machine-level code in response to expressions and control flow constructs. A particular advantage of this approach is that that the source program is read only once, and all of the translation work is performed during that read so we do not need to hold the whole program in memory and can work line-by-line. However, this *single-pass compiler* approach constrains the kind of languages that can be translated because at the time that expressions are processed, we must know the type of the operands so as to decide whether to emit, say, a floating point or an integer addition. As a result, languages such as Pascal and C originally required the types of all identifiers to be declared before use. In addition, in a single pass compiler we cannot perform any long range analyses, or realistically re-order the code to allow more efficient execution.

single-pass compiler

In multi-pass compilers the details of the Analyse, Rework and Effect phases vary widely between systems, and are often poorly documented, so it can be hard to establish the completeness, consistency and correctness of real compilers. Compiler errors are not rare, at least in the early stages of a language's development: as the user base for a language grows, confidence in the asso-

ciated translators naturally increases because the users are effectively testers. Language processors with few users should perhaps be treated with caution.

Just as software applications have a life-cycle, typically starting small and acquiring extra features over time, languages themselves display a life cycle, and modern versions of languages such as C and Java provide features that present significant challenges to all phases. The C language began as a small systems-oriented language but has had significant new functionality grafted on to it over the years, especially the extension to object orientation with C++.

As new features were added with necessary extensions to the syntax, the task of producing an appropriate grammar that was admissible by classical parsing algorithms became so difficult that around the turn of the twentieth century, the GNU C++ compiler abandoned parser generators in favour of manually crafted parsers. Without a generator, we must carefully analyse the hand written parser code and reassure ourselves that the front end is complete, consistent and correct.

The challenges multiply as we move through the phases. Modern versions of Java support limited *type inference* (that is the ability to deduce the type of an identifier at compile time without the programmer needing to explicitly specify it) and *pattern matching* (the use of patterns involving variables and constants in selection statements). Designing these sorts of features in a way that provides both backwards compatibility with earlier versions of the language *and* supports the sometimes quirky special cases that arise is very difficult indeed, and can lead to curious and unexpected behaviours.

1.3 Specification styles for syntax

Although formal semantics has not achieved much traction with language designers, there is almost complete agreement that syntax should be specified using a particular style of rewrite rule called a context free string-rewrite rule and that is evident in the extracts above; they all give a context free rewrite rule for the syntax, although the notation used for the rule varies slightly.

For Pascal we have

IfStatement = "if" BooleanExpression "then" Statement

The doubly-quoted symbols are Pascal keywords. The other symbols are placeholders for language fragments. For instance, a BooleanExpression could simply be the constant `true` or an expression like `x > y`.

In the C and Java standards, the placeholder symbols are written in an italic font with the literal keywords in an upright font. Clearly the placeholders could represent arbitrarily long pieces of program but when focussing on the syntax and semantics of the conditional statement, we don't want to have to specify their exact form. This is an example of *abstraction*, that is the hiding of unnecessary detail so that we can focus on the matter in hand.

abstraction

The purpose of a rule is to give a template for one feature of the language: the Pascal rule tells us that a conditional statement starts with the keyword `if` which must be followed by an expression yielding a boolean, then the keyword `then` followed by an arbitrary statement. Somewhere else in the grammar we expect to see definitions for the placeholders BooleanExpression and Statement. The C and Java versions tell us that a conditional statement startes with the keyword `if` which must be followed by an expression yielding boolean that *must* be enclosed in parentheses – in C and Java the keyword `if` is always followed by an open parenthesis.

Context Free Grammar

A complete set of context free rules with no missing definitions and a nominated *start rule* is called a *Context Free Grammar* (CFG).

1.4 Specification styles for semantics

We do not use English to write programs because the meaning of English phrases is too fuzzy. Ambiguity, allusion, and occasional precision are exactly what poets need, but our topic is not poetry: we are trying to build reliable computer systems. Hence, our programs are written in *formal languages* which we hope have a well defined syntax and semantics resulting in the same behaviour on all implementations. Nevertheless, current practice is to describe the semantics of the programming language itself in English.

Clearly we *ought* to be writing the specifications for programming languages formally too so that we can use software tools to help us show that our syntax and semantics are complete, consistent and correct, and to generate the phases of our translators, just as compilers check our programs and generate executable programs. As we've seen with GNU C++, even in the front end implementers have retreated from that ideal over time due to (in that case) the weakness of classical parser generator tools.

It seems that non-trivial languages always have 'dark corners' where the interaction of useful features can cause surprising effects. Java's design benefited greatly from previous generations of programming languages, but the *Java Puzzlers* book [**?**] show a wide variety of small programs with surprising effects – and that book was published in 2005 at the time of Java 5. The language has been significantly extended since then and further quirks will have resulted. Of course, these are confusions that arise at the level of the *user* of a language: the understanding of the language by *implementers* must (if their implementations are to be correct) subsume all of these details and more. A specification written in English is unlikely to answer all implementers' questions.

The most significant attempt to define a general purpose language using formal rules (rather then English-language descriptions) is Standard ML. SML began as a scripting language for a theorem prover. A key design goal was to provide an external syntax that was comfortable for mathematicians via the use of type inference which (almost) eliminates the need for type declarations and pattern matching on *Algebraic Data Types* to directly support the kinds of case

Algebraic Data Types

analysis that naturally arise when writing down proofs. The formal definition of the language was published in 1997 [**?**] and *in principle* allows automatic construction of an interpreter from the rules given in the book. We are going to use a related specification style, coupled to new algorithms which remove the weaknesses of classical front end tools, and intepreters which can directly execute formal semantic specifications.

1.5 Ambiguity

An ambiguous statement is one that can be interpreted in more than one way.

Ambiguity is a fertile source of jokes:

How do you make a sausage roll? Release the sausage at the top of a ramp.

How do you make a professor fast? Take their lunch away.

And so on.

Allegedly a medicine was once advertised with this slogan: *Try our headache cure: you won't get better.* That is probably too obvious to be true, but from the mid-1950s an aspirin-based analgesic really was promoted with the line *Nothing works faster than Anadin*; one interpretation being that taking nothing would offer faster relief than using the product.

We do not want ambiguity in our programming languages since that would suggest that different implementations might make different interpretations, and so a program might behave differently on different machines (or even different runs on the same machine). However, Java and other languages are littered with phrases that have multiple *possible* meanings. Here are three example questions: answers below.

1. In C and Java, does the entirely valid phrase $\mathsf{z = x---y}$ mean the same as $\mathsf{z = (x--) - y}$ or $\mathsf{z = x - (--y)}$ or $\mathsf{z = x - (- (-y))}$?
2. In everyday arithmetic (never mind programming languages) does $5-4-3$ evaluate to -2 or to 4, that is, should we interpret the expression as $((5-4)-3)$ or $(5-(4-3))$?
3. In this Java program fragment $\mathsf{y = 6;}$ **if** $\mathsf{(x > 3)}$ **if** $\mathsf{(x > 5)\ y = 1;}$ **else** $\mathsf{y = 0;}$ what should the final value of y be when x is 4? If we think that the fragment has the same meaning as

```
1 y = 6;
2 if (x > 3) {
3    if (x > 5)
4       y = 1;
5 }
6 else
```

```
7   y = 0;
```

then the the final value of y is 0.

If we use this interpretation

```
1 y = 6;
2 if (x > 3) {
3   if (x > 5)
4     y = 1;
5   else
6     y = 0;
7 }
```

then the final value of y is 6. The difference between the two interpretations is essentially whether the **else** clause belongs to the outer or the inner **if** statement.

In Section G.1 you will find a Java listing that illustrates all of the different interpretations which you can compile and run to see the differing outcomes.

disambiguation rule

For these three rather simple examples, it turns out that there is an agreed *disambiguation rule* (but beware: deciding how to resolve ambiguities in general can be very challenging).

1. The lex phase partitions the input using a so-called *longest match* strategy so the input is broken down into z = (x−−) − y and the resulting values are x=3 y=6 z=-2.

2. We are taught in elementary school that, by convention, subtraction binds more tightly to the left so the expression is interpreted as ((5 − 4) − 3) resulting in -2. Programming languages implement this convention.

3. The rule is that the **else** clause binds to the most recent **if** statement, so the phrase is interpreted as y = 6; **if** (x > 3) { **if** (x > 5) y = 1; **else** y = 0; } and the final value of y is unchanged by the **if** statements.

There are well established techniques that may be used to ensure that classical lexers and parsers always pick the agreed interpretation when handling these simple cases, but some ambiguities are harder to manage.

Consider the Java declaration Set<Set<Integer>> setOfSets; The intrinsic arguments are bracketed using <...> and so the nesting finishes with >>. But those two characters together also represent the right-shift operator in Java, so how does the lex phase know whether to partition >> as two closing bracket lexemes >, > or as a single operator >>? Once we have a phrase level analysis from the parser we can resolve this ambiguity, but in the classical pipeline the

lexer runs first, and classical lexers can only return a single sequence of lexemes. Users of classical tools have to adopt a variety of complex mechanisms to overcome these difficulties, and that makes the resulting translators hard to understand and hard to completeness, consistency and correctness. We shall show how to use *general lexer* and a *general multiparser* which allows all

We should stress that this kind of ambiguity does not result from using English to express the semantics: we have to be able to manage ambiguity in all translation stages even if they are fully formal.

1.6 Our approach – Ambiguity Retained Translation

A fundamental flaw in the classical front end pipeline is that each phase must commit to a *single* interpretation before moving on the next, but as we have seen with the >> ambiguity, the information needed to make that decision may not become available until later phases have done their work. Similar problems arise in the Parse phase when we may need type information for identifiers before it has been analysed.

What we need is a pipeline in which we can keep our options open, resolving ambiguities only when the necessary information becomes available. We call this strategy *Ambiguity Retained Translation* (ART) which is also the name of the translation tool we designed to illustrate the approach. You will find detailed documentation on ART in Appendix A along with installation instructions in Section A.1.

The ART front end In ART, the classical LR and LL style deterministic parsers are replaced by a *multiparser* and a *multilexer*. As the name suggests, multiparsers are capable of returning multiple derivations (interpretations) of a string, and in fact our MGLL algorithm will return *all* derivations, even when there are infinitely many of them [**?**]. MGLL achieves this in worst case cubic time and cubic space; in practice the worst case bound is elusive and the algorithms only require linear time for typical current programming language phrases, degrading gracefully towards the cubic bound when processing difficult parts of the grammar.

multiparser

multilexer

This capability does not come for free though: the data structures required during parsing and lexing grow rapidly and require many megabytes for realistic applications, and the amount of processing required for each input character is also significantly greater than for classical algorithms. At the time that the classical approach was being developed, MGLL would have been entirely impractical as it required far more memory than would have been available and would have run very slowly. However, modern machines typically run around a thousand times faster and have a thousand times more memory than those 1970s computers and so we can use these more advanced algorithms to free the language designer from the constraints of the classical algorithms.

Semantics in ART There are many formalisms that have been developed for

capturing the semantics of programming languages. We could, for instance, establish a relationship between phrases of a language and well-understood mathematical objects such as sets and functions, and then use traditional mathematical proof techniques to investigate program properties. That approach requires solid mathematical training to be fruitful.

term rewriting

SOS

In ART we do something much simpler: take the derivation tree that emerges from the front end and progressively transform it under the control of *term rewriting* rules until the program has all been rewritten away, accumulating the program's side effects as we go. The particular form of rewrite rules that we use is called a *Structural Operational Semantics* (*SOS*); this approach was introduced by Gordon Plotkin in 1980 [**?**].

attribute grammar

ART also offers a more concise specification style called an *attribute grammar* from which can automatically generate rewrite rules. That allows us to 'explain' attribute grammars as a constrained form of our rewrite rules. Importantly, attribute grammars may be interpreted rather efficiently, and so a rewrite system limited to attribute grammar style rules will run faster with an attribute-evaluator style interpreter than with a full rewrite interpreter. It turns out that placing extra constraints in the style of attribute grammar will allow even faster interpetation.

This refinement process rewrite rules to from illustrates a general principle: we want to proceed from formal specifications to implementations using, as automation as far as possible, because whenever we intervene manually we risk introducing errors.

The ART pipeline An ART specification is a collection of three different kinds of rules: **Context Free Grammar** rules specifying string rewrites which define the syntax; **Chooser** rules used to selectively discard interpretations; and **Conditional Term Rewrite** rules specifying tree rewrites which define the semantics

As well as these declarative rules, there may be *directives* which specify, for instance, which parts of the grammar is to be processed by the lex phase and which give test cases.

Conceptually ART follows the classical pipeline phases Lex – Parse – Analyse – Rework – Perform except that ambiguity management phases are inserted after Lex and Parse, and the other phases are merged together into sets of rewrite rules which may work on the internal form of the program in an intertwined way. Figure 1.1 shows this internal structure.

The ART front end (comprising the Lex – Lex choose – Parse – Parse choose phases) delivers a derivation term, unless lexical or parser syntax errors are detected.

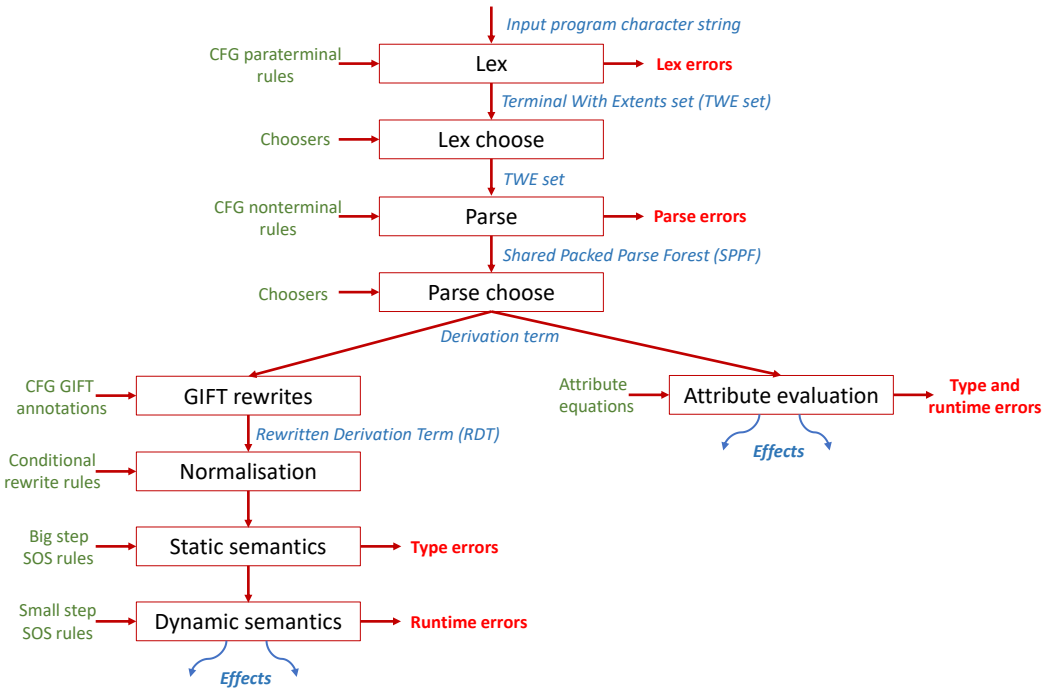


Figure 1.1 The ART pipeline

Chapter 2

Models of program execution

We shall use as a running example a tiny language which illustrates the core procedural concepts of variables, assignment, arithmetic and control flow in the form of conditionals and loops.

The inspiration for our language is Euclid's integer Greatest Common Divisor algorithm, described in the second proposition of *Elements VII* some 2,300 years ago. It is worth looking up the original description which is written in quite verbose prose. Here is a version written in Java.

```
public class GCD {
  public static void main(String[] args) {
    int a = 6;
    int b = 9;

    while (a != b) {
      if (a > b)
        a = a - b;
      else
        b = b - a;
    }
    int gcd = a;
  }
}
```

Java programs need quite a lot of setting up and anyway this is a course on language design, so let us construct our own, more compact programming notation to express the same program.

```
a := 6;
b := 9;

while a != b {
  if a > b
    a := a - b;
  else
    b := b - a;
}
```

```
10 gcd := a;
```

We shall call this notation the MiniGCD language. Note that assignment to a variable is denoted by `:=`, not by `=` asit would be in FORTRAN, C and Java. As in C and Java, statements are terminated with (not separated by) a semi-colon and can be grouped within braces. Variable names are not pre-declared and are assumed to be of type integer. MiniGCD only contains the features used here: it does not even provide addition (though we will extend it later).

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

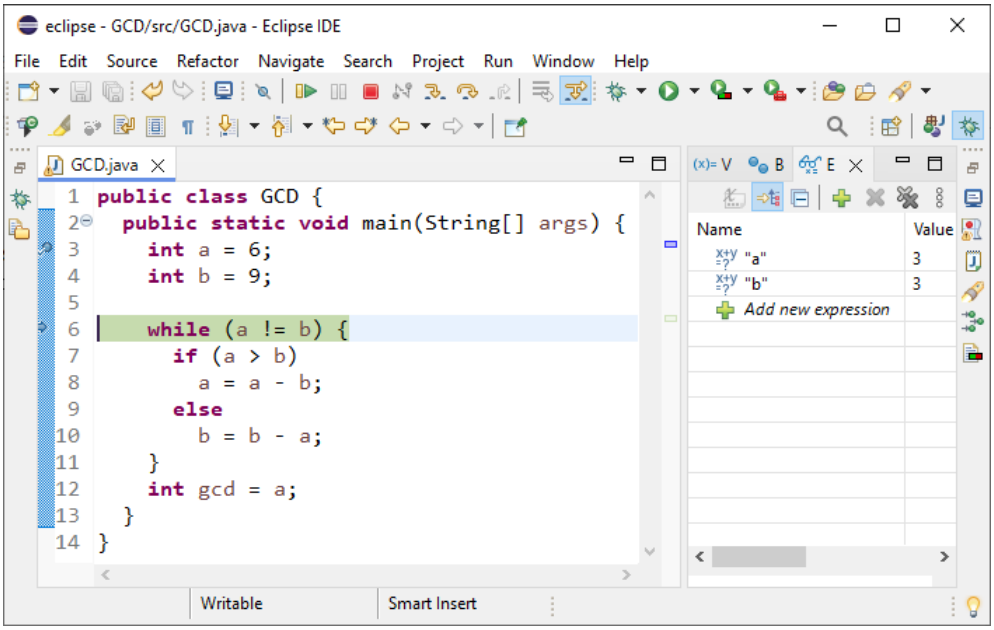
store

Program Counter

In almost all modern computing devices most programs are lists of instructions that reside in the memory or *store*. The instructions for a particular program do not change as it is being executed. A special register called the *Program Counter* (PC) which points to the next piece of code to be executed, and is usually simply incremented as each instruction is executed which induces sequential execution of the instructions. At a branch point we may test a condition and update the PC with a new values depending on that outcome; that causes the processor to start execution at some new location.

control flow

The sequence of values displayed by the program counter during a program's execution records the *control flow* for this particular input. The easiest way to visualise the control flow for a program is to load it into a development environment such as Eclipse and then run it under the controller of the debugger, which can execute it one line at a time. Here is a screen shot from Eclipse showing our Java GCD which is currently at line 6 on the final iteration.



We can see the program's Java code on the left and the state of the store with variables `a` and `b`, both presently mapped to the value 3. The program counter

value is represented by the small arrow in the margin at line 6.

2.2 The problem with assignment

substitution model

assignment model

The *substitution model* for variables that is used when thinking mathematically is much simpler to reason about than the *assignment model* for variables used in procedural programming languages. In mathematics, if I say $x = 3$ then, whilst that version of x remains in scope, I mean that x and 3 are synonyms and so anywhere that x appears subsequently in that scope I could cross it out and write 3. In procedural programming languages like Java, if I write `x = 3` I may subsequently write `x = 4` in the same scope region, and so the relationship between `x` and its value depends on the most recent assignment to `x` according to the execution history for a particular input. Hence assignment in languages like Java is fundamentally different to mathematical equality (and that is why programming languages use different symbols to denote assignment and equality).

bindings

A physical store is a fixed set of cells, each with a fixed address but containing a value which may be changed. A useful mathematical model of a store is as a set of *bindings* where each binding associates an identifier with a value.

Evaluating a *declaration* in the program term has the effect of creating a new store S' from S which has all of the bindings in S *and* the new binding required by the declaration. Assigning a new value to a variable has the effect of changing the mapping of one variable in the store, and using a variable in an expression requires us to look up the value mapped to the variable's identifier.

2.3 The problem with the program counter

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

The substitution model is simple and easy to reason about but the assignment model has the great advantage of being efficient in that identifiers may be re-used within a scope rather than having to have fixed content throughout the runtime of a program and that saves both memory and allocation time. The use of assignment to variables presents a challenge to formal analyses of program semantics though it is managable as we shall see. However it is *particularly* problematic that the program counter itself works by assignment because that obscures the control flow within a program, and that makes it difficult to decide whether two programs states are the same.

2.4 The reduction interpretation

There is an alternative way of thinking about program execution that does not require the use of a program counter. The trick is to think of the program code itself as something that can be progressively rewritten until all that we have left is a result, coupled to a set of bindings that record changes to variables.

Consider this program

```
1 x := 3;
2 y := 10+2+4;
```

The first thing the program does is assign 3 to variable x, that is create the store binding $x \mapsto 3$.

Now, there is a sense in which the program fragment x := 3; y := 10+2+4; coupled to the empty store {} is equivalent to the program fragment y := 10+2+4; with the store $\{x \mapsto 3\}$ because they both lead to the same final result. We could start with either and get the same result.

It is helpful to think of the store as representing the computer's state, and the program fragment as representing 'that which is left to do', so we can represent the execution of a program as a sequence of pairs comprising a program that represents only what remains to be done and a store:

1. x := 3; y := 10+2+4;, {}

2. y := 10+2+4;, $\{x \mapsto 3\}$

Next we need to evaluate expression 10+2+4 before we can assign the result to y. In detail, the computer can only execute one arithmetic operator at a time so we must pick a sub-expression to evaluate first; let us choose to execute 10+2 and rewrite it to the result 12.

3. y := 12+4;, $\{x \mapsto 3\}$

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

4. y := 16;, $\{x \mapsto 3\}$

Finally we can assign to y and set the program fragment to __done which is a special value indicating that there is no more computation required.

5. __done, $\{x \mapsto 3, y \mapsto 16\}$

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

reduction trace

reduction step

We call this kind of display of machine states the *reduction trace* for our program, and each line represents a *reduction step*–so called because usually the program fragment reduces in size at each step (though not always, as we shall

see in the next section). The steps match up rather well with the individual machine level instructions that would be executed by a real computer, and at every point we have a complete record of the state of the machine as well as being able to see what else we have to do.

2.5 The problem with loops

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

while booleanExpression **do** statement;

then we can *always* transform it into

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 can then be rewritten away. This device, then, allows us to treat **while** loops using only **if** statements.

2.6 A reduction evaluation of GCD in MiniGCD

Figure 2.1 on page 21 presents a reduction semantics trace for the GCD algorithm written in MiniGCD program shown on page 16.

There are a large number of steps in this trace, which make for intimidating reading, but bear in mind that each step (very roughly) corresponds to a machine operation such as fetching an operand or adding two numbers. Useful programs entail the execution of a *lot* of operations: some of the programs we run on modern processors take an appreciable amount of time to execute even though a 4GHz processor will, in just two seconds, execute one instruction for every person on the planet — a number well beyond our abilities to directly comprehend. This is just a roundabout way of saying that machine operations are fine grained, and we need an awful lot of them to do useful work. Any attempt to list all of the steps that are gone through by a non-trivial running program is going to generate a long list.

We shall use a slightly more compact form to display the steps. First, we write the entire program term on a single line: rather than the nicely laid out version shown on page 16, we say

a=6; b=9; **while** a!=b **if** a>b a:=a−b; **else** b:=b−a; gcd=a;

As before, our starting point is the whole program term coupled to an empty store:

a=6; b=9; **while** a!=b **if** a>b a:=a−b; **else** b:=b−a; gcd=a;, {}

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

redex

When reading the reduction trace below, the bold headings should simply be treated as comments: they are there to break up the reductions into related blocks as an aid to comprehension and have no part in the formal description of program execution. At each step look for the highlighted redex: the folowing step should contain a term which has all of the blue parts from its predecessor, and a replacement for the redex. The black part of each reduction step will record any changes arising from side effects of the reduction, which for this program are limited to creating or updating bindings but in general might also include changes to the input and output, the raising of exceptions, and other so-called *semantic entities*

semantic entities

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

normal forms

Upon termination, the variable gcd is bound to 3 which is indeed the greatest common divisor of 6 and 9.

2.7 Executable semantics and automation

Manually constructing the execution trace shown in Figure 2.1 would be time consuming, although it does have the benefit of showing very clearly and concisely what each step of the computation does (as opposed to the legalise English commentaries that we showed at the start of this chapter).

The whole point of this way of thinking about programming languages is allow *automation*. We need descriptions of programming languages that facilitate the mechanical construction of language processors. Ideally, we should like to be able to specify both the syntax and the semantics of a language like GCD in a few pages, and then have the computer run GCD programs for us so that we can test the specification and satisfy ourselves that it works the way we want it to. We call this prototyping style of language execution an *Executable*

Start of trace
a=6; b=9; **while** a!=b **if** a>b a:=a−b; **else** b:=b−a; gcd=a;, $\{\}$
b=9; **while** a!=b **if** a>b a:=a−b; **else** b:=b−a; gcd=a;, $\{a \mapsto 6\}$
Rewrite using `while p s → if p { s ; while p s }`
while a!=b **if** a>b a:=a−b; **else** b:=b−a;gcd=a;, $\{a \mapsto 6, b \mapsto 9\}$
Evaluate $a \neq b$ **with store** $\{a \mapsto 6, b \mapsto 9\}$
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; } gcd=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; } gcd=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; } gcd=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; }gcd=a;, $\{a \mapsto 6, b \mapsto 9\}$
Evaluate $a > b$ **with store** $\{a \mapsto 6, b \mapsto 9\}$
if a>b a:=a−b; **else** b:=b−a; **while** a!=b **if** a>b a:=a−b; **else** b:=b−a; gcd=a;, $\{a \mapsto 6, b \mapsto 9\}$
if 6>b a:=a−b; **else** b:=b−a; **while** a!=b **if** a>b a:=a−b; **else** b:=b−a; gcd=a;, $\{a \mapsto 6, b \mapsto 9\}$
if 6>9 a:=a−b; **else** b:=b−a; **while** a!=b **if** a>b a:=a−b; **else** b:=b−a; gcd=a;, $\{a \mapsto 6, b \mapsto 9\}$
if false a:=a−b; **else** b:=b−a; **while** a!=b **if** a>b a:=a−b; **else** b:=b−a; gcd=a;, $\{a \mapsto 6, b \mapsto 9\}$
Evaluate $b - a$ **with store** $\{a \mapsto 6, b \mapsto 9\}$
b:=b−a; **while** a!=b **if** a>b a:=a−b; **else** b:=b−a; gcd=a;, $\{a \mapsto 6, b \mapsto 9\}$
b:=b−6; **while** a!=b **if** a>b a:=a−b; **else** b:=b−a; gcd=a;, $\{a \mapsto 6, b \mapsto 9\}$
b:=9−6; **while** a!=b **if** a>b a:=a−b; **else** b:=b−a; gcd=a;, $\{a \mapsto 6, b \mapsto 9\}$
b:=3; **while** a!=b **if** a>b a:=a−b; **else** b:=b−a; gcd=a;, $\{a \mapsto 6, b \mapsto 9\}$
Rewrite using `while p s → if p { s ; while p s }`
while a!=b **if** a>b a:=a−b; **else** b:=b−a;gcd=a;, $\{a \mapsto 6, b \mapsto 3\}$
Evaluate $a \neq b$ **with store** $\{a \mapsto 6, b \mapsto 3\}$
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; } gcd=a;, $\{a \mapsto 6, b \mapsto 3\}$
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; } gcd=a;, $\{a \mapsto 6, b \mapsto 3\}$
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; } gcd=a;, $\{a \mapsto 6, b \mapsto 3\}$
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; } gcd=a;, $\{a \mapsto 6, b \mapsto 3\}$
Evaluate $a > b$ **with store** $\{a \mapsto 6, b \mapsto 3\}$
if a>b a:=a−b; **else** b:=b−a; **while** a!=b **if** a>b a:=a−b; **else** b:=b−a; gcd=a;, $\{a \mapsto 6, b \mapsto 3\}$
if 6>b a:=a−b; **else** b:=b−a; **while** a!=b **if** a>b a:=a−b; **else** b:=b−a; gcd=a;, $\{a \mapsto 6, b \mapsto 3\}$
if 6>3 a:=a−b; **else** b:=b−a; **while** a!=b **if** a>b a:=a−b; **else** b:=b−a; gcd=a;, $\{a \mapsto 6, b \mapsto 3\}$
if true a:=a−b; **else** b:=b−a; **while** a!=b **if** a>b a:=a−b; **else** b:=b−a; gcd=a;, $\{a \mapsto 6, b \mapsto 3\}$
Evaluate $a - b$ **with store** $\{a \mapsto 6, b \mapsto 3\}$
a:=a−b; **while** a!=b **if** a>b a:=a−b; **else** b:=b−a; gcd=a;, $\{a \mapsto 6, b \mapsto 3\}$
a:=a−3; **while** a!=b **if** a>b a:=a−b; **else** b:=b−a; gcd=a;, $\{a \mapsto 6, b \mapsto 3\}$
a:=6−3; **while** a!=b **if** a>b a:=a−b; **else** b:=b−a; gcd=a;, $\{a \mapsto 6, b \mapsto 3\}$
a:=3; **while** a!=b **if** a>b a:=a−b; **else** b:=b−a; gcd=a;, $\{a \mapsto 6, b \mapsto 3\}$
Rewrite using `while p s → if p { s ; while p s }`
while a!=b **if** a>b a:=a−b; **else** b:=b−a;gcd=a;, $\{a \mapsto 3, b \mapsto 3\}$
Evaluate $a \neq b$ **with store** $\{a \mapsto 3, b \mapsto 3\}$
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; } gcd=a;, $\{a \mapsto 3, b \mapsto 3\}$
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; } gcd=a;, $\{a \mapsto 3, b \mapsto 3\}$
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; } gcd=a;, $\{a \mapsto 3, b \mapsto 3\}$
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; } gcd=a;, $\{a \mapsto 3, b \mapsto 3\}$
Assign result
gcd=a;, $\{a \mapsto 3, b \mapsto 3\}$
, $\{a \mapsto 3, b \mapsto 3\, gcd \mapsto 3\}$

Figure 2.1 Reduction trace for the GCD algorithm with inputs 6,9

Executable semantics

semantics to distinguish it from am efficient production-quality compiler.

In this testing phase, we would be prepared to accept quite poor performance because our test programs would be small. If we were building a general purpose language we would probably need to then move on to a more efficient style of implementation, though still guided by the specification which would be the ultimate arbiter of correctness. However, it turns out that on modern hardware, for many applications needing a small language, an executable semantics might be fast enough on its own. We shall revisit this topic in Chapter 7.

Chapter 3

Rewriting

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

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

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

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

$a \wedge b$...

Now we know a few facts about Boolean algebra.

3.1 Equality of programs

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

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

generates the same output as

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

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

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

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

Code improvement

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

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

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

3.2 Mathematical objects, their denotations and software implementations

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

Mathematical objects When we are thinking mathematically, we are usually *imagining* abstract objects and operations regardless of whether we can make

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

Denotations When we are communicating about mathematics or programs we need conventions that enable us to write down what we mean. Consider the mathematical object that we get by adding unity to zero six times: we might denote that as 6, 06, *six* or *vi* (in Roman numerals). Which form we use is just a convention, and real programming languages usually support more than one convention: for instance Java allows us to write six as `6`, `06`, `0x06` or `0_6` and these are all denotations for the same mathematical object.

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

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

3.3 String rewriting

3.4 Term rewriting

Programs often contain *expressions* such as

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

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

This particular way of writing expressions follows the style that we learn in school which makes use of *infix operators* like + and / to represent the operations of addition and division; they are called infix because are written in between the things they operate on. Expressions can nest and we understand that evaluation of an expression proceeds from the innermost bracket: to compute

$17/(4+(x/2))$ we first need to divide the value of x by 2, then add 4, and then divide the result into 17.

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

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

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

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

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

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

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

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

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

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

.

If you are interested in the design of external language syntax then there are some alternatives to this approach that you might like to investigate. For instance Scheme and other LISP-like language use an exclusively prefix style; the printer control language PostScript uses Reverse Polish Notation; the Smalltalk language effectively uses an infix notation to activate all methods; the C++ language allows the dyadic operator symbols like `+` to have their meanings extended

to include new datatypes, and the Algol-68 language allowed completely new dyadic operator symbols to be defined. We shall return to these matters of syntactic style in Chapter **??**.

3.5 Internal syntax style

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

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

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

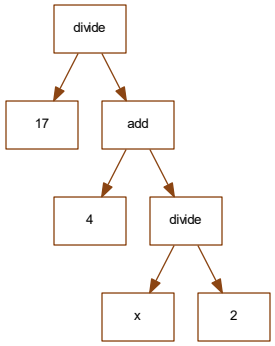
might be written

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

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

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

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



3.6 Terms

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

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

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

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

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

3.6.1 Denoting term symbols

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

Now, great care is needed when reasoning about and writing down terms. Rule 2 above make comma and parentheses special: how would we go about writ-

ing a symbol that contained parentheses or command? We call these special characters *metacharacters* because they are used in the denotation of terms.

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

3.7 The Value system and plugins

Chapter 4

Context Free Grammars

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

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

gcd := a
```

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

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

4.1 Outer and inner syntax

In practice, real tools usually maintain internal representations that are optimised for the task in hand, and they might not be tree shaped, or they might not need all of the elements of the derivation tree. In the context of language based software tools, we often call this internal form the *abstract* or *inner syntax* of a language. The terms *intermediate form*, *internal representation* and *model* also appear in different contexts. The term abstract syntax usually implies a formal (or at least semi-formal) relationship to the syntax of the user language, which is then called the *outer* or *concrete syntax*. Intermediate and internal forms are

often understood to be rather *ad hoc*, and it often not easy to show that all possible concrete syntax programs have a valid internal form: forgetting certain cases is a common implementation error. When the internal form is called a *model* we usually understand that the concrete syntax is primarily being used to load the members of a set of classes with data, where the interrelationships between the classes can be specified with a UML diagram. Tools exist that allow an existing UML diagram to be 'decorated' with concrete syntax so as to produce a language tuned to that model. If we need to add new classes, then the language can be regenerated with extended syntax. An alternative approach is to derive the model from the concrete syntax, by annotating the nonterminals and terminals which are 'significant' and must be loaded into the internal form.

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

4.1.1 Syntactic sugar, redundancy and syntactic 'noise'

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

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

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

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

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

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

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

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

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

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

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

4.2 The legacy of non-general parsing

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

The JLS example reinforces our expectation that even informal semantics are conventionally defined over the compositional syntax of the language. In practice, this means that we need to specify semantics with respect to particular productions, and would be very convenient to have productions (and thus nonterminals) which in some sense reflect the semantic concepts within the language in as simple a way as possible so as to reduce the number of formal cases that have to be specified. Abstract syntax for formal semantics systems emphasise this notion of *compressing* the concrete syntax into a concise form which matches *at least* all of the strings in the concrete language. In practice, inner grammars often match larger languages, and are typically highly ambiguous. When building tools for such systems, we use a concrete parser to produce an

individual derivation tree in the concrete syntax, and then give rules for, say, discarding redundant terminals and merging the children of nonterminals so as to produce a derivation in our inner syntax. The formal semantics interpreter can then work on the simplified tree.

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

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

4.3 Parsing by expanding the start symbol

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

4.4 Parsing by reducing to the start symbol

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

4.5 Multiparsing and the lexer-parser interface

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

4.6 OSBRD: Implementing a parser toolchain

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

A *parser generator* is a program which reads specifications for a grammar Γ written in BNF (or EBNF) and outputs the source code for a parser. When compiled the parser will test strings to see if they are in the language $L(\Gamma)$, and perhaps build a derivation tree.

Parser generators, then, can be thought of as processors for a DSL (BNF)

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

4.7 Ordered Singleton Backtrack Recursive Descent parsing

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

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

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

4.7.1 The OSBRD algorithm

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

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

This seemingly innocuous change has a big impact on the languages that can be successfully parsed by OSBRD compared to a truly general technique such

as GLL. We are highlighting this difference here because occasionally one encounters parsing tools which use algorithms based on ordered grammars, and in my experience the authors often do not adequately explain the limitations of the technique.

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

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

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

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

4.7.2 An OSBRD example in Java

Here is a small grammar.

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

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

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

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

```
1  boolean parse_S() {
2    int rc = cc, ro = co;
3
4    /* Nonterminal S, alternate 1 */
5    cc = rc; co = ro; oracleSet(1);
6    if (match("b")) { return true; }
7
8    /* Nonterminal S, alternate 2 */
9    cc = rc; co = ro; oracleSet(2);
10   if (match("a")) {
11   if (parse_X()) {
12   if (match("@")) { return true; }}}
13
14   return false;
15 }
16
17 boolean parse_X() {
18   int rc = cc, ro = co;
19
20   /* Nonterminal X, alternate 1 */
21   cc = rc; co = ro; oracleSet(1);
22   if (match("x")) {
23   if (parse_X()) { return true; }}
24
25   /* Nonterminal X, alternate 2 */
26   cc = rc; co = ro; oracleSet(2);
27   /* epsilon */ return true;
28 }
```

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

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

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

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

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

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

```
Input: 'a x x @'
S() at rc = 0, cc = 0 'a'
  S() alternate 1 rc = 0, cc = 0 'a'
    At 0 'a' match b - reject
  S() alternate 2 rc = 0, cc = 0 'a'
    At 0 'a' match a - accept
X() at rc = 2, cc = 2 'x'
  X() alternate 1 rc = 2, cc = 2 'x'
    At 2 'x' match x - accept
X() at rc = 4, cc = 4 'x'
  X() alternate 1 rc = 4, cc = 4 'x'
    At 4 'x' match x - accept
X() at rc = 6, cc = 6 '@'
  X() alternate 1 rc = 6, cc = 6 '@'
    At 6 '@' match x - reject
  X() alternate 2 rc = 6, cc = 6 '@'
    At 6 '@' match @ - accept
Accepted
Oracle: 2 1 1 2
```

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

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

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

4.8 Engineering a complete Java parser

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

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

1. `ARTOSBRDBase.java` which contains auxiliary methods.

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

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

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

```
40      cc++;
41    return true;
42  }
43  }
```

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

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

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

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

```
1  import java.io.FileNotFoundException;
2
3  class ARTGeneratedParser extends uk.ac.rhul.cs.csle.artosbrd.ARTOSBRDBase {
4  boolean parse_S() {
5    int rc = cc, ro = co;
6
7    /* Nonterminal S, alternate 1 */
8    cc = rc; co = ro; oracleSet(1);
9    if (match("b")) { return true; }
10
11   /* Nonterminal S, alternate 2 */
12   cc = rc; co = ro; oracleSet(2);
13   if (match("a")) {
14   if (parse_X()) {
15   if (match("@")) { return true; }}}
16
17   return false;
18 }
19
20 boolean parse_X() {
21   int rc = cc, ro = co;
22
23   /* Nonterminal X, alternate 1 */
24   cc = rc; co = ro; oracleSet(1);
```

```
25    if (match("x")) {
26    if (parse_X()) { return true; }}
27
28    /* Nonterminal X, alternate 2 */
29    cc = rc; co = ro; oracleSet(2);
30    /* epsilon */ return true;
31  }
32
33  SBtest(String filename) throws FileNotFoundException {
34    input = readInput(filename);
35
36    System.out.printf("Input: '%s'%n", input);
37    cc = co = 0; builtIn_WHITESPACE();
38    if (!(parse_S() && input.charAt(cc) == '\0'))
39       { System.out.print("Rejected%n"); return; }
40
41    System.out.print("Accepted%n");
42    System.out.print("Oracle:");
43    for (int i = 0; i < co; i++) System.out.printf(" %d", oracle[i]);
44    System.out.printf("%n");
45  }
46
47  public static void main(String[] args) throws FileNotFoundException{
48    if (args.length < 1)
49          new SBtest("");
50    else
51          new SBtest(args[0]);
52    }
53  }
```

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

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

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

4.9 Using built in matchers

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

The parser generator translates pseudo-terminals such as `&ID` into calls to

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

Here is a slightly modified version of our test grammar.

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

It generates these parse functions.

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

  /* Nonterminal S, alternate 1 */
  cc = rc; co = ro; ora cleSet(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 (builtIn_ID()) {
  if (parse_X()) { return true; }}

  /* Nonterminal X, alternate 2 */
  cc = rc; co = ro; oracleSet(2);
  /* epsilon */ return true;
}
```

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

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

```
int te.
```

```
boolean builtIn_ID() {
  if (!Character.isJavaIdentifierStart(input.charAt(cc))) return false;
  ts = cc++;
  while (Character.isJavaIdentifierPart(input.charAt(cc)))
    cc++;
  te = cc;
  builtIn_WHITESPACE();
  return true;
}

boolean isxdigit(char c) {
  if (Character.isDigit(c)) return true;
  if (c >= 'a' && c<='f') return true;
  if (c >= 'A' && c<='F') return true;
  return false;
}

boolean builtIn_INTEGER() {
  if (!Character.isDigit(input.charAt(cc))) return false;
  ts = cc;
  /* Check for hexadecimal introducer */
  boolean hex = (input.charAt(cc) == '0' &&
                 (input.charAt(cc + 1) == 'x' ||
                  input.charAt(cc + 1) == 'X'));
  if (hex) cc += 2; // Skip over hex introducer
  /* Now collect decimal or hex digits */
  while (hex ? isxdigit(input.charAt(cc)) :
               Character.isDigit(input.charAt(cc)))
    cc++;
  te = cc;
  builtIn_WHITESPACE();
  return true;
}

boolean builtIn_REAL() {
  if (!Character.isDigit(input.charAt(cc))) return false;
  ts = cc;
  while (Character.isDigit(input.charAt(cc)))
    cc++;
  if (input.charAt(cc) != '.')
    return true;
  cc++; // skip .
  while (Character.isDigit(input.charAt(cc)))
    cc++;
```

```
45    if (input.charAt(cc) == 'e' || input.charAt(cc) == 'E') {
46      cc++;
47    while (Character.isDigit(input.charAt(cc)))
48      cc++;
49    }
50    te = cc;
51    builtIn_WHITESPACE();
52    return true;
53  }
54
55  boolean builtIn_CHAR_SQ() {
56    if (input.charAt(cc) != '\'') return false;
57    cc++;
58    ts = cc;
59    if (input.charAt(cc) == '\\')
60      cc++;
61    cc++;
62    if (input.charAt(cc) != '\'') return false;
63    te = cc;
64    cc++; // skip past final delimiter
65    builtIn_WHITESPACE();
66    return true;
67  }
68
69  boolean builtIn_STRING_SQ() {
70    if (input.charAt(cc) != '\'') return false;
71    ts = cc + 1;
72    do {
73      if (input.charAt(cc) == '\\')
74        cc++;
75
76      cc++;
77    }
78    while (input.charAt(cc) != '\'');
79    te = cc;
80    cc++; // skip past final delimiter
81    builtIn_WHITESPACE();
82    return true;
83  }
84
85  boolean builtIn_STRING_DQ() {
86    if (input.charAt(cc) != '"') return false;
87    ts = cc + 1;
88    do {
89      if (input.charAt(cc) == '\\')
90        cc++;
91      cc++;
```

```
92    }
93    while (input.charAt(cc) != '"');
94    te = cc;
95    cc++; // skip past final delimiter
96    builtIn_WHITESPACE();
97    return true;
98  }
99
100 boolean builtIn_ACTION() {
101   if (!(input.charAt(cc) == '[' &&
102         input.charAt(cc + 1) == '*'))
103     return false;
104   cc += 2;
105   ts = cc;
106   while (true) {
107     if (input.charAt(cc) == 0)
108       break;
109     if (input.charAt(cc) == '*' && input.charAt(cc) == ']') {
110       cc += 2;
111       break;
112     }
113     cc++;
114   }
115   te = cc - 2;
116   builtIn_WHITESPACE();
117   return true;
118 }
```

4.10 Using attributes and inline semantics

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

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

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

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

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

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

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

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

4.10.1 Attributes

Simply printing out messages showing where we are in a parse is interesting, but limited. If we want to perform useful computations, it turns out that we need to pass information *between* parse functions or, equivalently, around the derivation tree.

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

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

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

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

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

```
3|          '0' X:rv | # .
```

The language of this grammar is

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

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

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

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

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

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

Semantics phase
Result is 2
```

4.10.2 A four function calculator

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

```
1| S ::= exprs:val [* System.out.printf("Final result: %d\n", val); *] .
2|
3| exprs:int ::= add:val ';' [* System.out.printf("Result: %d\n", val); *] exprs:rv |
4|                add:rv [* System.out.printf("Result: %d\n", rv); *] .
5|
6| add:int ::= mul:l '+' add:r [* rv = l + r; *] |
7|             mul:l '-' add:r [* rv = l - r; *] |
8|             mul:rv .
```

```
9
10 mul:int ::= op:l '*' mul:r [* rv = l * r; *]|
11             op:l '/' mul:r [* rv = l / r; *]|
12             op:rv .
13
14 op:int ::= integer:rv |
15            '(' exprs:rv ')' .
16
17 integer:int ::= digit:hi digit:mid digit:lo
18                    [* rv = hi*100 + mid*10 + lo; *] |
19                digit:mid digit:lo [* rv = mid*10 + lo; *] |
20                digit:rv .
21
22 digit:int ::= '0' [* rv = 0; *] |
23               '1' [* rv = 1; *] |
24               '2' [* rv = 2; *] |
25               '3' [* rv = 3; *] |
26               '4' [* rv = 4; *] |
27               '5' [* rv = 5; *] |
28               '6' [* rv = 6; *] |
29               '7' [* rv = 7; *] |
30               '8' [* rv = 8; *] |
31               '9' [* rv = 9; *] .
```

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

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

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

4.11 Implementing inline semantics

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

As a result of this retry behaviour, we cannot simply execute inline semantics during the parse, even though when we design grammars and their semantic

actions we tend to think of the action being executed as a side-effect of parsing. Instead, we need to complete the searching associated with parsing and only then run through the grammar the 'correct' way to execute the actions. This is the purpose of the oracle: during parsing we construct the oracle as we go, adjusting it as necessary when we backtrack. By the end of the parse we have a map of where the parser *should* have gone. We call the control data structure an oracle because it is as if we had a parser which instead of guessing where to go could simply ask an all-powerful oracle for advice.

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

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

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

Here are the associated semantics functions.

```
void semantics_S() {
int result;
  switch(oracle[co++]) {
    case 1:
      match("b");
      break;

    case 2:
      match("a");
      result = semantics_X();
   System.out.printf("Result is %d%n", result);

      match("@");
      break;
  }
}
int semantics_X() {
  int rv = 0;
int sum;
  switch(oracle[co++]) {
    case 1:
      match("1");
      sum = semantics_X();
   rv = sum + 1;

      break;
```

```
27
28      case 2:
29        match("0");
30      rv = 0;
31
32        rv = semantics_X();
33        break;
34
35      case 3:
36        /* epsilon */
37        break;
38    }
39    return rv;
40  }
```

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

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

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

4.12 Making explicit trees

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

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

```
1  TreeNode tree_S() {
2    TreeNode leftNode = null, rightNode = null;
3    switch(oracle[co++]) {
4      case 1:
5        /* 'b' */ leftNode = rightNode =
6        new TreeNode("b", null, rightNode, TreeKind.TREE_TERMINAL,
7                     TIFKind.TIF_NONE, null);
8        match("b");
9        break;
10
11     case 2:
12       /* 'a' */ leftNode = rightNode =
13       new TreeNode("a", null, rightNode, TreeKind.TREE_TERMINAL,
14                    TIFKind.TIF_NONE, null);
15       match("a");
16       /* X */ rightNode =
17       new TreeNode("X", tree_X(), rightNode, TreeKind.TREE_NONTERMINAL,
18                    TIFKind.TIF_NONE, null);
19       /* '@' */ rightNode =
20       new TreeNode("@", null, rightNode, TreeKind.TREE_TERMINAL,
21                    TIFKind.TIF_NONE, null);
22       match("@");
23       break;
24
25   }
26   return leftNode;
27 }
28 TreeNode tree_X() {
29   TreeNode leftNode = null, rightNode = null;
30   switch(oracle[co++]) {
31     case 1:
32       /* '1' */ leftNode = rightNode =
33       new TreeNode("1", null, rightNode, TreeKind.TREE_TERMINAL,
34                    TIFKind.TIF_NONE, null);
35       match("1");
36       /* X */ rightNode =
37       new TreeNode("X", tree_X(), rightNode, TreeKind.TREE_NONTERMINAL,
38                    TIFKind.TIF_NONE, null);
39       break;
40
41     case 2:
42       /* '0' */ leftNode = rightNode =
43       new TreeNode("0", null, rightNode, TreeKind.TREE_TERMINAL,
44                    TIFKind.TIF_NONE, null);
45       match("0");
46       /* X */ rightNode =
47       new TreeNode("X", tree_X(), rightNode, TreeKind.TREE_NONTERMINAL,
```

```
48                          TIFKind.TIF_NONE, null);
49          break;
50
51        case 3:
52          /* # */ leftNode = rightNode =
53          new TreeNode("#", null, rightNode, TreeKind.TREE_EPSILON,
54                          TIFKind.TIF_NONE, null);
55          break;
56
57      }
58      return leftNode;
59  }
```

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

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

4.12.1 The TreeNode class

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

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

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

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

```
1  TreeNode tree_X() {
2    TreeNode leftNode = null, rightNode = null;
3    switch(oracle[co++]) {
4      case 1:
5        /* '1' */ leftNode = rightNode =
6        new TreeNode("1", null, rightNode, TreeKind.TREE_TERMINAL,
7                     TIFKind.TIF_NONE, null);
8        match("1");
9        /* X */ rightNode =
10       new TreeNode("X", tree_X(), rightNode, TreeKind.TREE_NONTERMINAL,
11                    TIFKind.TIF_NONE, null);
12       break;
13
14     ...
15
16   }
17   return leftNode;
18 }
```

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

The constructor takes as paramaters the node's label, a reference to the first child node and a reference to the left sibling. Inside the constructor, the left sibling's `sibling` reference is updated to point to the newly created `TreeNode`. That part of the code is perhaps the most subtle element of sandbox's imple-

mentation: it repays study. The other fields are the 'kind' of the node (nonterminal, terminal, builtin terminal or epsilon), a TIF operator and the substring matched by a builtin.

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

```
enum TreeKind {TREE_EPSILON, TREE_TERMINAL, TREE_BUILTIN, TREE_NONTERMINAL};
enum TIFKind {TIF_NONE, TIF_FOLD_UNDER, TIF_FOLD_OVER, TIF_FOLD_ABOVE};

class TreeNode{
  String label; int nodeNumber; TreeNode child; TreeNode sibling;
  TreeKind kind; TIFKind tifOp; String attribute;

  TreeNode(String label, TreeNode child, TreeNode previousSibling,
           TreeKind kind, TIFKind tifOp, String attribute) {
    if (previousSibling != null) previousSibling.sibling = this;
    this.label = label; this.child = child; this.sibling = null;
    this.kind = kind; this.tifOp = tifOp; this.attribute = attribute;
    nodeNumber = nextNode++;
  };

  TreeNode(TreeNode old) {
    label = old.label; kind = old.kind; tifOp = old.tifOp;
    child = sibling = null;
    attribute = old.attribute;
    nodeNumber = nextNode++;
  };

};
```

4.12.2 Cloning trees

```
  TreeNode clone(TreeNode parent, TreeNode previousSibling) {
    TreeNode ret = new TreeNode(this);
    if (previousSibling != null) previousSibling.sibling = ret;
    else if (parent != null) parent.child = ret;
    TreeNode rightNode = null;
    for (TreeNode srcNode = child; srcNode != null; srcNode = srcNode.sibling)
      rightNode = srcNode.clone(ret, rightNode);
    return ret;
  }
```

4.12.3 Visualising trees on the console

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

```
void print(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));
    if (attribute != null) System.out.printf(":%s", attribute);
    System.out.printf("%s\n", tifString(tifOp));

    if (child != null) child.print(++indent);
    if (sibling != null) sibling.print(indent);
  };
```

4.12.4 Visualising trees with the GraphViz tools

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

4.12.5 Implementing TIF operators

```
  TreeNode evaluateTIF(TreeNode parent, TreeNode previousSibling,
                              boolean parentSuppressed) {

    // Special case: don't promote root node
    if (parent != null && (tifOp == TIFKind.TIF_FOLD_UNDER ||
                                    tifOp == TIFKind.TIF_FOLD_OVER))
    {
      /* Link the children in to the previousSibling's chain */
      TreeNode rightNode = null;
      if (previousSibling != null) rightNode = previousSibling;
      else if (parent != null) rightNode = parent.child;

      boolean suppress = tifOp == TIFKind.TIF_FOLD_UNDER ||
            (parentSuppressed && tifOp == TIFKind.TIF_FOLD_OVER);

      for (TreeNode srcNode = child; srcNode != null; srcNode = srcNode.sibling)
        rightNode = srcNode.evaluateTIF(parent, rightNode, suppress);
```

```
18
19          if (tifOp == TIFKind.TIF_FOLD_OVER && !parentSuppressed) {
20            parent.label = label; /* What about tifOp? */
21            parent.kind = kind;
22            parent.attribute = attribute;
23          }
24
25          return rightNode;
26        }
27        else { /* make a new node and scan our children */
28          TreeNode ret = new TreeNode(this); ret.tifOp = TIFKind.TIF_NONE;
29          if (previousSibling != null) previousSibling.sibling = ret;
30          else if (parent != null) parent.child = ret;
31          TreeNode rightNode = null;
32          for (TreeNode srcNode = child; srcNode != null; srcNode = srcNode.sibling)
33            rightNode = srcNode.evaluateTIF(ret, rightNode, false);
34          return ret;
35        }
36      }
37
38      void foldunderEpsilon(){
39        if (kind == TreeKind.TREE_EPSILON)
40          tifOp = TIFKind.TIF_FOLD_UNDER;
41        for (TreeNode srcNode = child; srcNode != null; srcNode = srcNode.sibling)
42          srcNode.foldunderEpsilon();
43      }
```

4.13 A Sandbox grammar for Sandbox

```
1  grammar ::= ruleOrActions^ .
2
3  ruleOrActions ::= action ruleOrActions^ | rule ruleOrActions^ | # .
4
5  rule ::= nonterm '::='^ cats^ '.'^ .
6
7  cats ::= cat catTail^ .
8  cat::= element cat^ | # .
9  catTail ::= '|'^ cat catTail^ | # .
10
11 element ::= action^^ | subrule^^ | nonterm^^ tif | term^^ tif |
12                builtIn^^ tif | epsilon^^ tif .
13
14 action ::= &ACTION .
15
16 subrule ::= subruleWrapper .
17 subruleWrapper ::= '('^ cats^ subruleKind^^ .
```

```
18 subruleKind ::= ')'^^ | ')?'^^ | ')+'^^ | ')*'^^ .
19
20 nonterm ::= nontermWrapper .
21 nontermWrapper ::= &ID^^ optionalAttribute .
22
23 term ::= termWrapper .
24 termWrapper ::= &STRING_SQ^^ optionalAttribute .
25
26 tif ::= '^^^'^^ | '^^'^^ | '^'^^ | #^^ .
27
28 builtIn ::= '&'^ &ID .
29
30 epsilon ::= '#'^.
31
32 optionalAttribute ::= ':'^ &ID^^ | #^^ .
```

4.14 The Gather-Insert-Fold-Tear formalism

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

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

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

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

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

4.14.1 Fold operators

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

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

A rule such as

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

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

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

and rule such as

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

will generate derivation subtree of the form

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

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

For fold-under, then, we have

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

gives

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

and

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

gives

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

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

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

For fold-under

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

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

For fold-over

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

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

4.14.2 The Tear operator

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

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

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

4.14.3 Insertions

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

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

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

4.14.4 The Gather operator

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

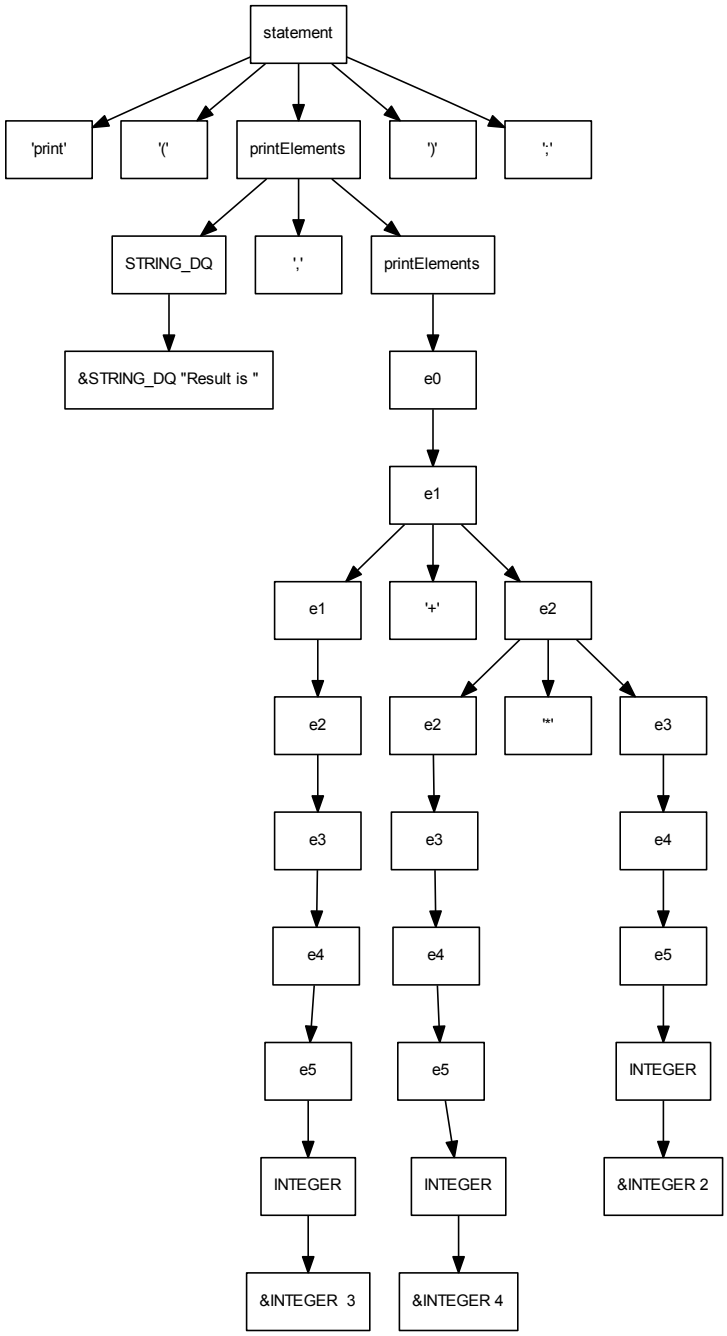
4.15 GIFT applications

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

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

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

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

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

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

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

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

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

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

STRING_DQ ::= &STRING_DQ ;

INTEGER ::= &INTEGER ;
```



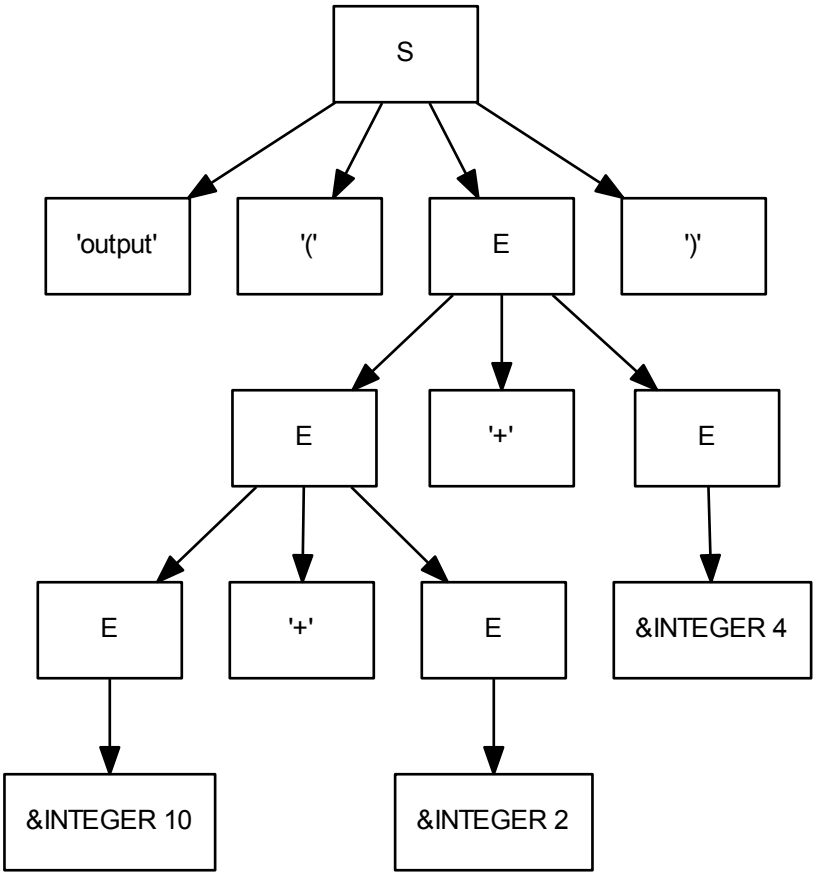

```
statement ::= 'print'^^ '('^ printElements^ ')'^ ';'^ ; (* print statement *)

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

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

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

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

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

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

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

STRING_DQ ::= &STRING_DQ ;

INTEGER ::= &INTEGER ;
```

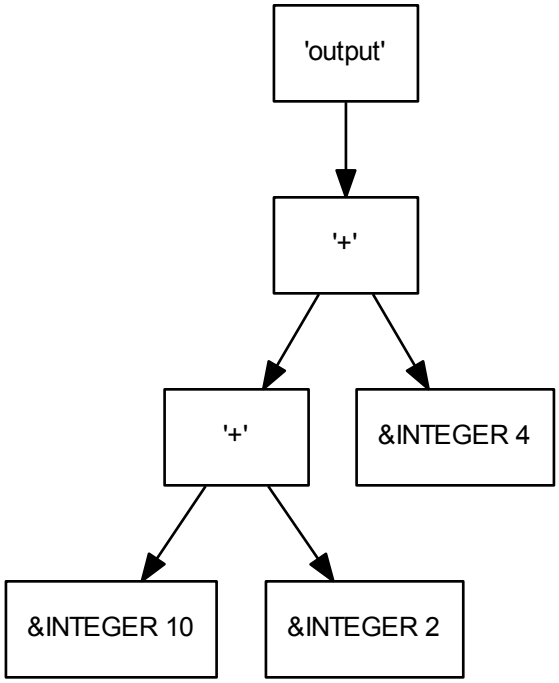
A *term* is a well formed formula (wff) is some formal language Γ. Our formal view of semantics will be a game in which terms representing program fragments and semantic entities such as the store and output will evolve through transitions. We shall move between three different ways of representing terms: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

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

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



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



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

4.16 A languages is a set of strings

4.17 A Context Free Grammar generates a language

4.18 A derivation records rule applications

4.19 A parser constructs derivations

4.20 GIFT operators rewrite derivations

4.21 Paraterminals specify the lexer-parser interface

4.22 Choosers reduce ambiguity

Chapter 5

Reduction semantics

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

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

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

5.1 The basic idea

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

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

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

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

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

5.2 Execution via substitution

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

In Chapter **??** we have already seen (in an informal setting) that we can describe program execution as a set of states with transitions between them representing the execution steps of a program. More formally, we developed the idea of a program execution trace as a series of steps that walk a *transition relation*

over *configurations*. A configuration represents the state of a computer, and configuration Γ_1 is related to Γ_2 if and only if Γ_2 can appear immediately after Γ_1 in *some* execution of *some* program. Configurations always contain a program term, and in addition we add entities that represent whatever side effects of program execution we need to record.

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

5.2.1 Configurations

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

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

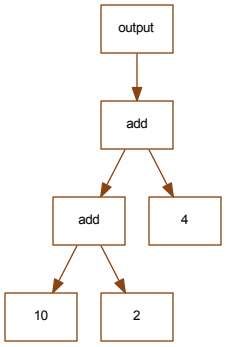
For example, this program fragment when interpreted will eventually result in the value 16 being output.

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

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

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

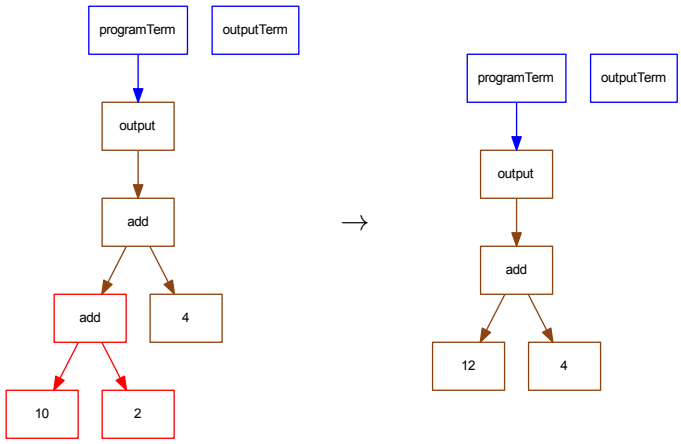
This sort of prefix functional term corresponds directly to a tree if we think of the written term as being the textual trace of a pre-order tree traversal, with parentheses being added to show when we pass down or up a tree edge:



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

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

which we can show graphically as



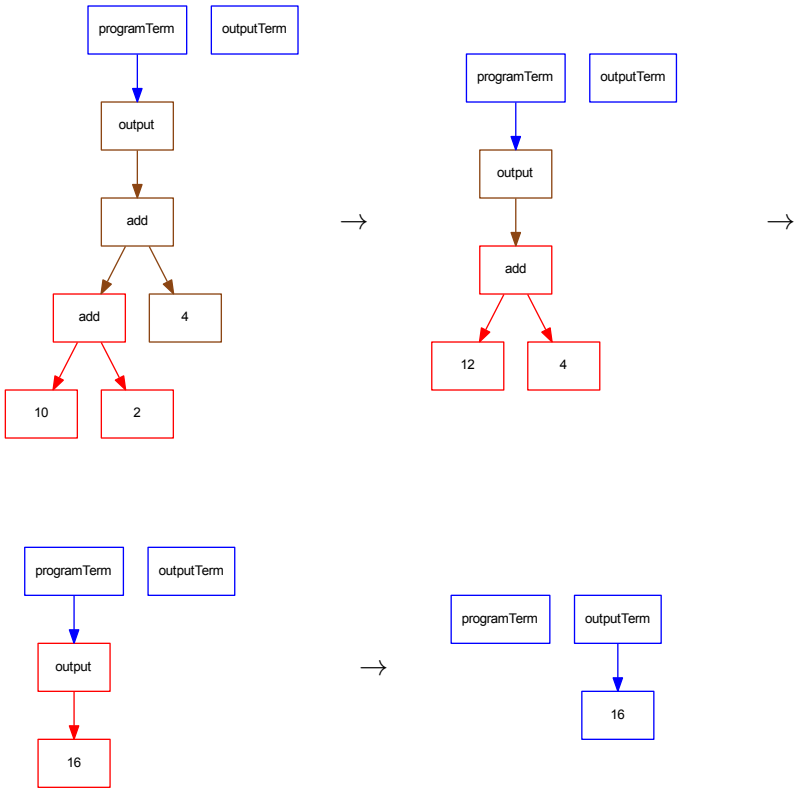
As in Chapter **??**, we have highlighted the program part that is about to be rewritten in red. We also have blue nodes representing the various state components of a running program: in this case limited to the program term and a presently empty term intended to represent the list of outputs made by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

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

$$\langle \mathsf{output}(\mathsf{add}(\mathsf{add}(10,\ 2),4)), [\ ] \rangle$$

⟨output(add(12,4)), [ ]⟩
⟨output(16), [ ]⟩
⟨, [16]⟩

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



5.3 Avoiding empty terms – the special value done

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

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

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

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

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

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

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

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

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

5.4 Term variables are metavariables

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

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

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

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

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

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

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

5.5 Pattern matching of terms

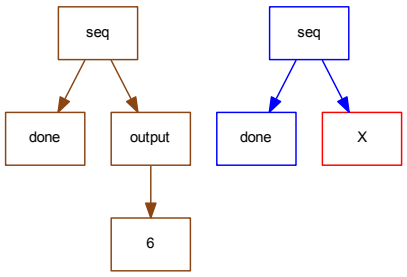
Since we are trying to build a formal game which will execute a program without human intervention, we need some syntax in which to write the rules of the game. As we have already seen, for this reduction of a sequence with done as its left hand argument we write

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

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

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

We can represent this process visually as follows:



The sepia coloured closed term is to be matched against the blue pattern. Some of the leaves of the pattern term may be coloured red: these are nodes labeled

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

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

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

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

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

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

We shall write

$$\theta \triangleright \pi$$

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

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

returns

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

and

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

returns

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

whereas

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

returns

$$\bot$$

because output(6) does not match done.

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

5.6 Pattern substitution

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

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

We shall write

$$\pi \triangleleft \rho$$

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

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

returns

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

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
  }
```

5.7 Rules and rule schemas

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

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

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

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

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

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

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

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

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

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

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

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

In pseudo code, we might say something like this:

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

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

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

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

In this case we are using two new functions: `is012(x)` which returns a term *true* or a term *false* depending on whether `x` is in the set $\{0, 1, 2\}$ or not; and `addOp(x, y)` which returns a term labeled with the number formed from the addition of terms `x` and `y`. Notice how everything we are doing reduces to operations over terms: our functions are not returning values such as *true* or *false*; they are returning trees made up of a single node *labeled* with *true* or *false*. Notice also that we are using names such as `addOp` for the function that computes the operation of addition, as opposed to the term constructer `add` which is a tree label from a program term. You should think of `add` as the piece of syntax that requests an addition, and `addOp` as the name of the function (or perhaps even machine instruction) which will actually perform the addition. We shall follow this convention throughout: names with the `Op` suffix are used for functions that perform computations, and they must not also appear as the names of a term (tree) element.

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

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

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

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

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

The inference rule representation of our schema is

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

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

This allows us to abbreviate our inference rule to:

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

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

5.8 The interpreting function F_{SOS}

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

5.8.1 Managing the local environment

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

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

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

5.8.2 Procedural pseudo-code for F_{SOS}

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

```
1  let R be the set of rule schemas
2
3  Fsos(C: configuration) returns configuration OR bottom
4    for r in R
5      if C |> r.conclusion.lhs then
6        let E be an empty set of bindings
7        for c in R.conditions
8          if isSideCondition(c)
9            let res be (c.lhs <| E) |> c.rhs
10           if res = bottom then next r else add res to E
11         else
12           let T be c.lhs <| E
13           if isvalue(T) then return T
14           let res be Fsos(T) |> c.rhs
15           if res = bottom then next r else add res to E
16       return r.conclusion.rhs <| E
17   return bottom
```

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

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

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

Conditions can be either side conditions or transitions.

- ◇ For a side condition we call the left hand side function and then pattern match the result against the condition's right hand side.

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

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

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

If we exhaust the rules set R then we have arrived at a terminal configuration, that is one from which no further transitions may be made. For a correct program, this terminal transition will correspond to the program's final version. If the rules were ill-formed, it would be possible to run out of applicable transitions prematurely: such a configuration is called a *stuck configuration* and would be reported as an error by the interpreter which requires the rules to be changed. In detail, we nominate some program terms as *values*. If an evaluation terminates with a value term, then we have a normal execution. If an evaluation terminate with a non-value term, then we have a stuck execution. The *done* constructor in our rule is an example of a value: it marks normal termination of commands. Numeric and other literals such as strings are also usually values; and we may indicate the successful completion of expression evaluations by reducing them to one of these values.

5.8.3 Program term rewrites - the outer interpreter

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

```
interpret(C: configuration) returns configuration
  C' = Fsos(C)

  while C' not bottom
    C = C'
    C' = Fsos(C)

  if not isValue(C.theta) error('Stuck configuration')

  return C
```

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

5.9 Structural Operational Semantics and F_{SOS} traces

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

Using a technique due to Plotkin called *Structural Operational Semantics*, we shall specify similar inside-out steps often using repeated term traversals. The basic idea is to conditionally rewrite the term by isolating a subcomputation that can be performed immediately, and we ensure that the rewrites are done in the inside-out order by building inference rules that have the abstract syntax of our language embedded in the conclusions.

5.9.1 SOS rules for an addition language

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

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

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

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

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

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

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

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

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

This is the first rule we have seen that has a transition as one of its conditions rather than a simple side-condition. Looking at the Fsos() function pseudo-code,

you can see that this rule simply takes any program with output() as its outer constructor, pulls out the argument (which can be of arbitrary complexity) by pattern matching it to term variable E, and then recursively calls Fsos() on the configuration $\langle E, \alpha \rangle$. Whatever comes back from that call is then used to rewrite the program term into a simpler form.

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

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

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

done is a value, so interpretation terminates.

5.9.2 Expression nesting

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

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

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

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

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

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

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

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

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

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

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

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

5.10.1 Configurations

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

5.10.2 Variable handling

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

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

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

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

this effect by placing the rule at the end of a specification.

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

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

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

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

5.10.3 Arithmetic operations

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

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

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

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

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

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

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

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

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

5.10.4 Boolean relations

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

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

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

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

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

5.10.5 Sequential flow control

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

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

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

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

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

5.10.6 Conditional flow control

An if statement is a language takes an expression as a predicate, and a command to be executed. If the predicate yields true then the command is executed; otherwise it is skipped. We can express these semantics using two rules that operate on the degenerate predicates true and false. These rules are unconditional; that is they are *axioms*.

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

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

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

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

5.10.7 Loops

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

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

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

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

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

5.11 Using big steps to simplify the rules

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

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

$$\frac{\langle E,\sigma\rangle \Rightarrow \langle \mathsf{true},\sigma\rangle}{\langle \mathsf{if}(E,C),\sigma\rangle \rightarrow \langle C,\sigma\rangle} \tag{5.1}$$

$$\frac{\langle E,\sigma\rangle \Rightarrow \langle \mathsf{false},\sigma\rangle}{\langle \mathsf{if}(E,C),\sigma\rangle \rightarrow \langle \mathsf{done},\sigma\rangle} \tag{5.2}$$

$$\frac{\langle E,\sigma\rangle \Rightarrow \langle \mathsf{true},\sigma\rangle}{\langle \mathsf{while}(E,C),\sigma\rangle \rightarrow \langle \mathsf{if}(E,seq(C,\mathsf{while}(E,C))),\sigma\rangle} \tag{5.3}$$

$$\frac{\langle E,\sigma\rangle \Rightarrow \langle \mathsf{false},\sigma\rangle}{\langle \mathsf{while}(E,C),\sigma\rangle \rightarrow \langle \mathsf{done},\sigma\rangle} \tag{5.4}$$

$$\langle \mathsf{seq}(\mathsf{done},C_2),\sigma\rangle \rightarrow \langle C_2,\sigma\rangle \tag{5.5}$$

$$\frac{\langle C_1,\sigma_1\rangle \rightarrow \langle C_1',\sigma_2\rangle}{\langle \mathsf{seq}(C_1,C_2),\sigma_1\rangle \rightarrow \langle \mathsf{seq}(C_1',C_2),\sigma_2\rangle} \tag{5.6}$$

$$\frac{\langle E_1,\sigma\rangle \Rightarrow \langle I_1,\sigma\rangle \quad \langle E_2,\sigma\rangle \Rightarrow \langle I_2,\sigma\rangle \quad \mathtt{isgt}(I_1,I_2) \triangleright \mathsf{false}}{\langle \mathsf{gt}(E_1,E_2),\sigma\rangle \Rightarrow \langle \mathsf{false},\sigma\rangle} \tag{5.7}$$

$$\frac{\langle E_1,\sigma\rangle \Rightarrow \langle I_1,\sigma\rangle \quad \langle E_2,\sigma\rangle \Rightarrow \langle I_2,\sigma\rangle \quad \mathtt{isgt}(I_1,I_2) \triangleright \mathsf{true}}{\langle \mathsf{gt}(E_1,E_2),\sigma\rangle \Rightarrow \langle \mathsf{true},\sigma\rangle} \tag{5.8}$$

$$\frac{\langle E_1,\sigma\rangle \Rightarrow \langle I_1,\sigma\rangle \quad \langle E_2,\sigma\rangle \Rightarrow \langle I_2,\sigma\rangle \quad \mathtt{addOp}(I_1,I_2) \triangleright V}{\langle \mathsf{add}(E_1,E_2),\sigma\rangle \Rightarrow \langle V,\sigma\rangle} \tag{5.9}$$

$$\frac{\langle E_1,\sigma\rangle \Rightarrow \langle I_1,\sigma\rangle \quad \langle E_2,\sigma\rangle \Rightarrow \langle I_2,\sigma\rangle \quad \mathtt{mulOp}(I_1,I_2) \triangleright V}{\langle \mathsf{mul}(E_1,E_2),\sigma\rangle \Rightarrow \langle V,\sigma\rangle} \tag{5.10}$$

$$\frac{\langle E, \sigma_1\rangle \Rightarrow \langle V, \sigma_1\rangle \quad \texttt{put}(\sigma_1, X, V) \triangleright \sigma_2}{\langle \mathsf{assign}(X,E),\sigma_1\rangle \rightarrow \langle \mathsf{done},\sigma_2\rangle} \tag{5.11}$$

$$\frac{\texttt{get}(R, \sigma) \triangleright V}{\langle R,\sigma\rangle \Rightarrow \langle V,\sigma\rangle} \tag{5.12}$$

5.12 Interpretation traces for our language

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

5.12.1 Example 1 – assignment to literal

Initial configuration

⟨assign(X, 3), {}⟩

Trace

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

5.12.2 Example 2 – assignment to variable

Initial configuration

⟨assign(Y, X), $\{X \mapsto 3\}$⟩

Trace

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

5.12.3 Example 3 – sequence over assignments

Initial configuration

⟨seq(assign(X, 3), assign (Y, X)), {}⟩

Trace

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

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

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

5.12.4 Example 4 - conditional assignment

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

⟨if(gt(X,0)), assign(Y, X)), $\{X \mapsto 2\}$⟩

Trace

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

```
        [4.12].SC1 yields 3
        [4.12] rewrites to 3
      [4.8]C2 yields 0 (already a value — no recursive call)
      [4.8]SC1 yields true
      [4.8] rewrites to true, { X |–> 2 }
    [4.1] rewrites to assign(Y, X)), { X |–> 2 }
```

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

5.12.5 Example 5 - loops

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

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

Trace

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

...

5.13 An eSOS specification for MiniGCD

```
seq ::= statement^^ | statement seq
statement ::= assign^^ | while^^ | if^^ | '{'^ seq '}'^
assign ::= &ID ':='^ expression ';'^
while ::= 'while'^ expression 'do'^ statement
if ::=
  'if'^ expression 'then'^ statement
| 'if'^ expression 'then'^ statement 'else'^ statement
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^^
__int32 ::= &INTEGER
deref ::= &ID

--- seq(_C1->__done, _C2) -> _C2
--- if(_E->True, _C1,_C2) -> _C1
--- if(_E->False,_C1,_C2) -> _C2
--- while(_E, _C) -> if(_E, seq(_C, while(_E,_C)), __done)
--- assign(_X, _E->_n:__int32) -> __done, __put(_sig, _X, _n)
--- deref(_R) -> __get(_sig, _R), _sig
--- gt (_E1->_n1:__int32, _E2->_n2:__int32) -> __gt (_n1, _n2)
--- ne (_E1->_n1:__int32, _E2->_n2:__int32) -> __ne (_n1, _n2)
--- sub(_E1->_n1:__int32, _E2->_n2:__int32) -> __sub(_n1, _n2)

!try "a := 6; b := 9; while a != b do if a > b then a := a - b; else b := b - a;"
```

Figure 5.1 An eSOS specification for MiniGCD

Chapter 6

Attribute interpreters

As we have seen, an SOS interpretion of a program starts with a derivation tree and progressively rewrites it until only a normal form remains whilst accumulating side effects in subsidiary semantic entities. This approach has the great benefit of compartmentalising the meanings of phrases in a way that naturally supports hierarchical composition of semantics, reflecting the nesting principle that is so widely used in programming languages. The nesting is primarily expressed in the hierarchy of string rewrite rules that forms the grammar; each rewrite rule only needs to express the meaning of an isolated syntactic phrase.

An *attribute interpreter* is an alternative way of developing a compositional semantics from a derivation tree. Instead of having a single current configuration which is rewritten at each step we associate data elements called *attributes* with each node of the derivation tree, and then write equations or small code fragments called *actions*. The derivation tree itself does not change during an attribute-based interpretation, and this means that an attribute interpreter may be much faster than a reduction-style interpreter.

attributes

actions

The general ideas behind attribute evaluators can be seen in several early compilers which implement semantics by traversing derivations and accumulating information in an *ad hoc* way. in 1968, Donald Knuth described a formalisation of this approach called an *Attribute Grammar (AG)* [**?**, **?**, **?**]. The values of attributes in an Attribute Grammar are specified using a purely equational approach, and the equations for attributes at a particular derivation tree node must only depend on other attributes which are locally visible. For a subtree comprising some parent node and its children, attributes of the parent node whose equations depend only on attributes of the child nodes are called *synthesized* attributes. Attributes of child nodes which are updated by equations are called *inherited* attributes.

Attribute Grammar (AG)

A substantial research literature on Attribute Grammars exists, much of which focusses on procedures to evaluate attributes in an order that optimises evaluation time. One important special case (called an L-attributed grammar) allows evaluation is a single post-order traversal of the tree.

Many parser generator tools incorporate a modified notion of Attribute Grammars that we call an *Attribute-Action Grammar (AAG)*

Attribute-Action Grammar (AAG)

6.1 Attribute Grammars

6.1.1 Derivation traversers

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

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

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

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

6.2 Attribute Grammars

It is natural to think of the leaves of a derivation tree as being associated with values: for instance an INTEGER token matching the string `"0123"` is associ-

ated with the value 123 and so on. When implementing arithmetic expressions, it is useful to think of these values as percolating up through the tree, being transformed by operators as we go.

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

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

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

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

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

6.2.1 The formal attribute grammar game

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

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

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

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

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

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

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

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

6.2.2 Attribute grammars in practice

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

Consider a rule of the form

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

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

right hand side instances are numbered in a single sequence; in tools we often maintain separate sequences for each unique nonterminal name as here.

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

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

6.2.3 Attribute grammar subclasses

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

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

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

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

6.3 Semantic actions in ART

Semantic actions and attributes in ART use the parser's implementation language to model the decalarative, equational attribute grammar formalism. Back

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

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

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

6.4 Syntax of attributes in ART

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

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

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

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

So, for instance,

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

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

not be picked up by either ART or the Java compiler, but instead generate a run-time exception.

6.4.1 Special attributes in ART

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

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

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

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

6.5 Accessing user written code from actions in ART generated parsers

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

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

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

The `support{...}` declaration specifies that material within the braces be

copied into the generated code within the ART generated parser class itself, allowing us to declare methods and variables which are visible throughout the generated parser's actions.

6.6 A naïve model of attribute evaluation

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

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

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

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

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

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

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

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

6.7 The representation of attributes within ART generated parsers

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

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

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

in module M generates the class

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

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

6.8 The ART RD attribute evaluator

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

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

The ART attribute evaluator correctly evaluates *L-attributed* grammars. In an L-attributed grammar, in every production

$$X \rightarrow y_0 y_1 y_2 \ldots y_k \ldots y_n$$

every inherited attribute of y_k depends only on the attributes of $y_0 \ldots y_k - 1$ and the inherited attributes of X.

There is quite a strong parallel here with parsing: a general parsing algorithm such as GLL (the algorithm ART implements) can handle any specification, but with the risk of poor performance on some grammars. A non-backtracking Recursive Descent parser, on the other hand, can only handle deterministic LL(1) grammars (or ordered grammars which are nearly LL(1)) but will run in linear time.

Our evaluator is essentially a recursive descent evaluator. It will only traverse the tree once. As long as the equations may be fully resolved in a single pass, all will be well. The ART evaluator is limited to attribute schemes that are essentially the L-attributed schemes. However, we can do a lot with such schemes, and the evaluation time is linear in the size of the tree.

In detail, the ART evaluator recurses over the datastructure constructed by the GLL parser. This is not a single derivation, but a (potentially infinite) set of derivation trees embedded within a structure called a Shared Packed Parse Forest (SPPF). However, prior to starting the evaluator, we will have marked some parts of the SPPF as suppressed, and some parts as selected, and the net effect is that the evaluator can assume that it is recursing over a single derivation tree.

As the evaluator enters a node labeled X, it creates the attribute block for each nonterminal child below it (corresponding to the nonterminal instances in the derivation step $X \Rightarrow \alpha$ encoded in this height-1 sub-tree). These newly-created attribute blocks are assigned to variables with names like $Y1$ and $Z2$ corresponding to the first and second instances of Y and Z in some production like `X::= Y Z Z`

The evaluator is a nest of functions, one for each nonterminal, in a way that is isomorphic with our OSBTRD parser functions. In our example above, the evaluator function for X will be called and make attribute blocks for the children $Y1$, $Z1$ and $Z2$. It will then call the evaluator function for Y passing block $Y1$ as an argument. The evaluator functions all take a single parameter block whise name is the same as that of the nonterminal. By this means, the block for Y allocated in X is called $Y1$ in the evaluator for X but called Y in the evaluator for Y.

Just like an RD parser, the evaluator functions call each other in the same order as instances are encountered within the grammar, and the semantic actions are inserted directly into the evaluator functions.

6.9 Higher order attributes

There is a well-developed theory of higher order attributes, which are attributes that represent parts of derivation trees rather than simple values. There are essentially two classes of HO attributes: attributes which capture part of an existing derivation tree, and attributes which contain new pieces of tree which can be used to extend a derivation tree from the parser. In ART, we support

the former, but not yet the latter. This means that the shape and labelling of a derivation tree in ART cannot be modified by an attribute grammar: only the attribute values associated with tree nodes can be modified. In a later section we shall describe ART's GIFT operators which do allow trees to be modified. In the present implementation, the attribute evaluator works on the full derivation and completes evaluation before the GIFT rewriter changes the tree.

ART's notion of higher order attributes requires only two things: a way of marking tree nodes as having a higher-order attribute associated with them, and a way to allow the user to activate the evaluator function under the control of semantic actions.

The first is achieved by adding an annotation `<` to any right hand side instance of a nonterminal in a grammar rule. The second is achieved by providing a method `artEvaluate()` which takes as an argument a higher order attribute.

When the evaluator function arrives at a node with a higher-order attribute, it does not descend into it (although it will construct the attribute block for it). The idea is that instead of automatically evaluating a subtree, the outer evaluator will ignore it, but the user may specify semantic actions to trigger its evaluation on demand.

Why is this useful? Well one application is to allow our recursive evaluator to interpret flow-control constructs. Consider an `if` statement. It comprises a predicate, and a statement which is only to be executed if the predicate is true. We can specify this as follows:

```
ifStatement ::= 'if' e0 'then' statement<
        { if (e01.v != 0) artEvaluate(ifStatement.statement1, statement1); } ;
```

The `<` character after the instance of statement creates a higher order attribute called `statement1` in the attribute block for `ifStatement`. ART will also have created an attribute block called `statement1`. The evaluator will automatically descend into the subtree for `e0`, but will not descend into the subtree for `statement`: instead it loads a refernce to the subtree for this instance of `statement` into the attribute statement1 in `ifStatement`.

In the action, we look at the result that was computed within `e0`, and if it is not zero (signifying false) we call the evaluator on the the subtree root node held in the attribute `ifStatement.statement1` and pass in parameter block `statement1`. This effectively emulates what would have happened automatically if we had left off the `<` annotation, but under the control of the result of `e0`. Hence the evaluation order of the tree is being dictated by the attributes and semantic actions themselves! This is exactly the sense in which our attributes are higher order. However, we can only traverse bits of tree that were built by the parser: we cannot make new tree elements and call the evaluator on them. Full higher order attributes do allow that. We call our restricted form *delayed* attributes so as to distinguish them from the more general technique.

```
!global variables:__map

statements ::= statement | statement statements


statement ::=
  ID ':=' subExpr ';'  statement.v = __put(variables, ID1.v, subExpr1.v)
| 'if' relExpr statement 'else' statement
    statement.v = relExpr.v ? statement1.v !! statement2.v
| 'while' relExpr statement  statement.v = relExpr.v @ statement.v !! __done

relExpr ::=
  subExpr                relExpr.v = subExpr1.v
| relExpr '>' subExpr    relExpr.v = __gt(relExpr.v, subExpr.v)
| relExpr '!=' subExpr   relExpr.v = __ne(relExpr.v, subExpr.v)

subExpr ::=
  operand               subExpr.v = operand.v
| subExpr '-' operand   subExpr.v = __sub(subExpr.v, operand.v)

operand::=
  ID                 operand.v = __get(variables, ID.v)
| INTEGER            operand.v = INTEGER1.v
| '(' subExpr ')'    operand.v = subExpr1.v
```

Figure 6.1 An attribute-action specification for MiniGCD

We can use these delayed attributes to build interpreters for languages with conditionals, loops and function calls, as we shall see in the laboratory exercises.

6.10 An attribute-action specification for MiniGCD

6.11 Exercises

Chapter 7

Software language design and pragmatics

7.1 The semantic facets of programming languages

We shall group our discussion of semantic components of high level languages into five *facets*: (a) values, types and expressions; (b) storage, assignment and commands; (c) identifiers, bindings and scope; (d) control flow and (e) abstraction mechanisms, both procedural and data.

7.1.1 Values, types and expressions

The ultimate purpose of a program is to compute *values*; those values might be numeric solutions to equations, textual outputs, visual effects on a screen or movements of a robot arm. At machine level, all of these correspond to patterns of binary digits in memory, but programming languages provide a range of abstractions which enable us to reason more effectively about program execution. Grouping the available values by class of abstraction naturally leads to the notion of *type*.

7.1.2 Storage, assignment and commands

In a von Neumann view of computing, values are stored in reusable cells, and in that world, storage is also fundamental. Most program texts are dominated by identifiers which stand for, for instance, constant values, locations in store,

7.1.3 Identifiers, scope and binding

Nested scope rules

7.1.4 Control flow

D-structures

Concurrency

Jumps

Exceptions

7.1.5 Procedural and data abstraction

Procedures

Higher order functions

Abstract data types, classes and packages

Generics

Appendix A

ART user manual

A.1 Downloading and running ART for the first time

1. ART is written in Java; therefore an up-to-date Java installation is required. At the time of writing, the UK Oracle download page for Java is at

   https://www.oracle.com/uk/java/technologies/downloads/

   Select and install the appropriate version for your operating system.

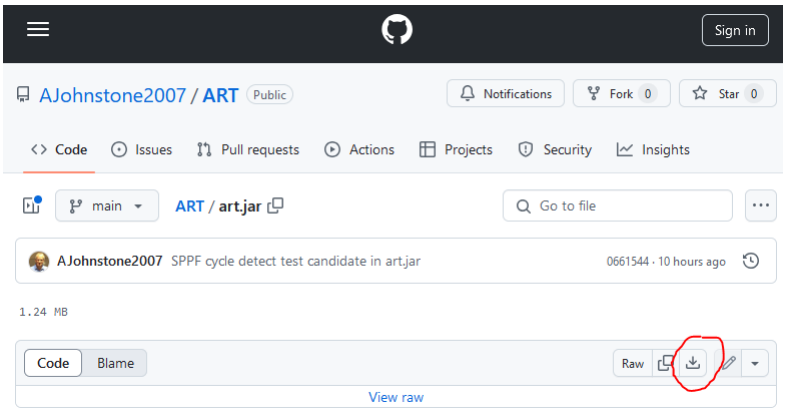
   Other Java implementations are available and locatable via search engines.

2. Make a work directory which we shall call ***artwork***.

   Download the `art.jar` file by opening a Web browser on:

   https://github.com/AJohnstone2007/ART/blob/main/art.jar

   Click the GitHub download button (circled in red below) to download a copy of `art.jar` to your work directory ***artwork***.



3. Test the download and your Java installation by opening a command window, changing your directory to ***artwork*** and typing the command

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

   The expected output is a version number, a build timestamp and summary usage information which will look like this:

   ```
   ART 5_0_241 2024-11-01 08:12:44
   Usage:
     ...
   ```

4. Instead of the ART message you may see something like this:

   ```
   Class has been compiled by a more recent version of the Java Environment
   (class file version xy.0), this version of the Java Runtime only recognizes
   class file versions up to pq.0.
   ```

   This means that your Java installation is for an old version of Java, and you will need to install a current version: see step 1.

5. The official ART repository is at

   https://github.com/AJohnstone2007/ART

   It includes the latest version of this document at

   https://github.com/AJohnstone2007/ART/blob/main/doc/slewa.pdf

   with the examples from this document in a single ART script at

   https://github.com/AJohnstone2007/ART/blob/main/doc/slewa.art

   and the current source code under

   https://github.com/AJohnstone2007/ART/tree/main/src

A.2 IDE and graphical component installation

The `art.jar` file includes an Integrated Development Environment (IDE) and support for building languages that display 2D and 3D graphics. Graphical support requires JavaFX, and the IDE in addition uses the RichTextFX Java component for text editing.

1. JavaFX is no longer part of the main Java installation, and must be separately installed via the page at

   https://gluonhq.com/products/javafx/

2. The RichTextFX component repository is at

   https://github.com/FXMisc/RichTextFX

   ART looks for a copy of the 'fat jar' (which contains all RichTextFx's dependencies) when it starts up. This can be directly downloaded to your ***artwork*** directory from

   https://github.com/AJohnstone2007/ART/blob/main/richtextfx.jar

3. Running ART with JavaFX requires a very long command line because of the need to specify class and module paths.

   The Windows batch script `art.bat` contains this line:

   ```
   java --module-path %jfxHome%\lib --add-modules javafx.controls
           -cp .;%artHome%\art.jar;%artHome%\richtextfx.jar
           uk.ac.rhul.cs.csle.art.ART %*
   ```

A.3 Command line interface

A.4 ART script language

An ART script is built from four kinds of elements:

1. Rewrite rules, of the form *premises* `---` *conclusion*
2. Context Free Grammar Rules, of the form *identifier* `::=` *cfgExpression*
3. Choose rules, of the form *slotSet* `>` *slotSet* or *slotSet* `>>` *slotSet*
4. Directives, which begin with an exclamation mark `!`

The script language is free format, and arbitrary whitespace or comments may appear before and after each script language token. The ART script language specification is available from the repository at

https://github.com/AJohnstone2007/ART/blob/main/src/uk/ac/rhul/cs/csle/art/script/ARTScriptSpecification.art

A.4.1 Lexical structure

1. **identifier** zzz
2. **signed integer**
3. **signed real**
4. **string literal**
5. **character literal**
6. **filename**

A.4.2 Rewrite rules

A.4.3 Context free grammar rules

A.4.4 Choose rules

A.4.5 Directives

1. **!include**
2. **!whitespace**

3. **!paraterminal**
4. **!lexer**
5. **!parser**
6. **!interpreter**
7. **!start**
8. **!configuration**
9. **!clear**
10. **!trace**
11. **!print**
12. **!show**
13. **!prompt**
14. **!try**
15. **!nop**

A.5 Lexical builtins

Lexical builtins are hardcoded recognisers for certain classes of substring which may be used as shorthands for common lexical patterns on the right hand side of Context Free Grammar rules. Builtin names begin with an ampersand `&` character.

1. **&CHAR_BQ** ‘C
2. **&ID** AlphanumericIdentifier
3. **&INTEGER** 123
4. **&REAL** 12.3
5. **&STRING_BRACE** {A string delimited by braces}
6. **&STRING_BRACE_NEST** {A string {with nested instances} delimited by braces}
7. **&STRING_DOLLAR** $A string delimted by dollar signs$
8. **&STRING_DQ** ”A string delimited by double quotes”
9. **&STRING_PLAIN_SQ** ’A string delimited by single quotes with no escapes’

10. **&STRING_SQ** 'A string delimited by single quotes'

    The following lexical builtins can only appear as an argument to the `!whitespace` directive. They are discarded by the lexer, and will never appear in a lexicalisation (and so it would be an error for them to appear within a Context Free Grammar rule).

11. **&SIMPLE_WHITESPACE**
12. **&COMMENT_BLOCK_C** /* a C-style block comment */
13. **&COMMENT_LINE_C** // a C-style line comment
14. **&COMMENT_NEST_ART** (* An ART style comment (* nestable *) *)

A.6 The ART value system

ART provides several builtin types and operations which may be used instead of rewrite rules to perform more efficient basic arithmetic and collection operations.

A.6.1 Operations

A.6.2 Types

A.7 ART value plugins

Constructor	Returns	Action
__**eq**(L, R)	__**bool**	value of L equal to value of R
__**ne**(L, R)	__**bool**	value of L not equal to value of R
__**gt**(L, R)	__**bool**	value of L greater then value of R
__**lt**(L, R)	__**bool**	value of L less than value of R
__**ge**(L, R)	__**bool**	value of L greater that or equal to value of R
__**le**(L, R)	__**bool**	value of L less than or equal to value of R
__**compare**(L, R)	__**int32**	if $L < R$ then -1 else if $L > R$ then +1 else 0
__**not**(L)	$T(L)$	Logical or bitwise inversion
__**and**(L, R)	$T(L)$	Logical or bitwise conjunction
__**or**(L, R)	$T(L)$	Logical or bitwise disjunction
__**xor**(L, R)	$T(L)$	Logical or bitwise exclusive OR
__**lsh**(L, R)	$T(L)$	Left shift L by R bits
__**rsh**(L, R)	$T(L)$	Right shift L by R bits, propagating zeroes
__**ash**(L, R)	$T(L)$	Right shift L by R bits, propagating sign bit
__**neg**(L)		
__**add**(L, R)		
__**sub**(L, R)		
__**mu**(L, R)**l**		
__**div**(L, R)		
__**mod**(L, R)		
__**exp**(L, R)		
__**size**(L)		
__**cat**(L, R)		
__**slice**(L, R)		
__**get**(L, R)		
__**put**(L, K, V)		
__**contains**(L, K)		
__**remove**(L, K)		
__**extract**(L)		
__**union**(L, R)		
__**intersection**(L, R)		
__**difference**(L, R)		
__**cast**(L, R)		

Table A.1 ART value operations

Constructor	Rôle	Operations
__bottom	Match failure	**__eq __ne**
__done		
__empty		
__quote		
__proc(M**,**B**)**		
__bool(V**)**		
__char(V**)**		
__intAP(V**)**		
__int32(V**)**		
__realAP(V**)**		
__real64(V**)**		
__string(V**)**		
__list(V, N**)**		
__map(K, V, N**)**		
__hmap(P**,** K, V, N**)**		
__adtProd(V, N**)**		
__adtSum(V, N**)**		
__blob(N**)**		

Table A.2 ART value types and allowed operations

Appendix B

The Royal Holloway course

This chapter is for students studying Software Language Engineering at Royal Holloway where we approach the material in a particular order designed to allow students to complete their projects within the footprint of a one semester course.

Holloway students start with internal syntax and reduction semantics and only then learn about external syntax parsers and the use of GIFT operators to generate terms in their chosen internal syntax, before moving on to attribute-action systems. After studying that core material we look at topics in lexicalisation and ambiguity management.

Experienced readers will note that this is a 'semantics-first' approach: we encourage students to first enumerate the features of their language as a set of signatures, then write reduction rules to interpret those signatures, and only then to consider the external appearance of phrases in their language. We justify and expand on this general approach in Chapter 7.

> **For readers who are not following the Holloway course:** if you are using ART just as a parser, or have a particular interest in one or other approach to semantics you may want to take a different route.

B.1 Aims and motivation

Welcome to the Software Language Engineering course. You have been engineering *with* software languages for at least two years now. This course, though, is about the engineering *of* software languages: you will learn how to build languages using concise notations from which the implementation could be automatically generated.

All forms of engineering are a mixture of creative insight and disciplined implementation. For instance, the architect of a bridge will try to design an aesthetically pleasing structure that meets the requirements, but ultimately they will rely on detailed structural engineering calculations on that structure to test whether it will withstand daily use. Ideally, this would be true for software too: our creativity would be expressed only through sound and principled techniques; that is techniques that have been found to be safe and efficient using mathematical and other forms of analysis.

In practice we mostly write software in a hopeful way, and then use testing to try and fill the gaps in our understanding. Unfortunately, programming languages are inherently difficult to test since they are designed to be flexible notations with very many combinations of interacting features.

We aim to tame this complexity by using *high level abstractions* that allow us to see the specification of a complete language in a few pages. We can then use this specification to guide either a hand crafted, efficient implementation, or automatically generate interpreters for the language which will probably be less efficient, but may be adequate for many applications.

The overall goal is to make language processors that can be comfortably maintained and extended by the engineers that take forward our work after we have moved on to other projects. To do that, we need to provide concise, self documenting descriptions of the syntax and semantics of our languages that everybody can understand and work with.

B.2 Learning outcomes

After working through these notes, you will

1. know how to use *Context Free Grammar* rules to define programming language syntax;

2. be able to use grammar idioms and the *GIFT annotations* to create *derivation trees* with useful properties;

3. understand how to write *reduction semantics* rules that the rewriter uses to interpret programs;

4. be able to write *attribute-action rules* that may provide more efficient implementations;

5. understand the types and operations of a *Value system*, and how to use a *plugin* to connect to Java classes; and

6. be able to recognise ambiguity in language specifications and progressively eliminate it using *choosers*.

B.3 Assessment

Your command of these learning outcomes will be assessed *via* a personal project and an invigilated examination.

The focus of this course is on the constructs of general purpose programming languages, but to motivate the project work we offer three project variants to construct a *Domain Specific Language* for

1. music generation *via* the Java MIDI interface;
2. image processing using the two-dimensional (2D) features of Java FX; and
3. Computer Aided Design for 3D printing using an extended form of the three-dimensional (3D) features of Java FX.

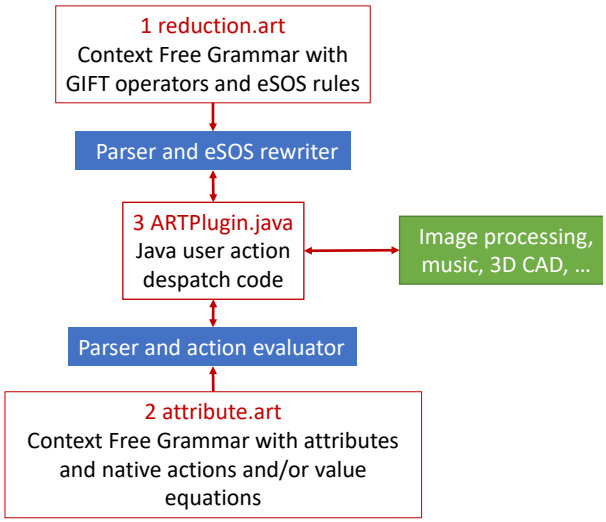
You must commit to one of these topics by the end of the week 4 lab. To help you decide, please read the background material on each of these in Appendices D, E and F.

The project work is to be submitted at the end of the course, and requires the following four deliverables.

1. `reduction.art` - a reduction semantics interpreter specified by eSOS rules and a context free grammar annotated with GIFT operators.
2. `attribute.art` - an attribute-action interpreter specified by a context free grammar annotated with attributes and actions written as Value expressions.
3. `ARTPlugin.java` - a plugin which connects your interpreters to your chosen domain specific actions

It is important to remember that this course is about language design and implementation: a project that has s trivial language but a very rich plugin will not score as highly as a rich programming language with a simple backend. Don't be tempted to write a fancy set of domain specific Java code but neglect the core programming language deliverables; on the other hand your plugin should have real functionality and not just report calls as the example.

Here is a diagram showing how the three deliverables interact with the ART system:



The blue boxes above represent language interpretation mechanisms that are built into the ART tool; their behaviour is specified by the rules that you write in `reduction.art` and `attribute.art`. The green box represents arbitrary Java code that you can connect to *via* a despatch routine in `ARTPlugin.java`.

After each weekly lab session, you will submit a snapshot of your work. These snapshots will not be assessed, but will be automatically analysed so that your progress can be tracked. If you are falling behind then you *must* ask for help.

B.3.1 Marking scheme

The invigilated, unseen examination accounts for 50% of the marks on this course. The remaining 50% of the marks come from the three project deliverables.

Project marking scheme

There are two submission points, A and B shown in blue and green respectively in this table:

Deliverable	Basic	Extension	Beyond	Totals
Internal	2.5			2.5
reduction.art	5	5	5	15
ARTplugin.java	5	5	5	15
attribute.art	5	5	5	15
Examples			2.5	2.5
Blue - submission A				50
Green - submission B				

For each deliverable, the markers look for (a) basic work, which means 'filling-

in' the missing components of the examples, (b) extension work which means bringing material into your project which is in the examples of this book and (c) 'beyond' work which means working out things for which examples are not given in the book.

A complete set of basic language features corresponds to a pass mark of around 40%. Satisfactorily implementing extension features found elsewhere in the course materials will yield marks in the 40-70% range. Adding in features that you have thought about and implemented yourself will help you achieve marks above 70%.

What does this mean in practice? Consider the different facets of language design.

For the expressions facet: the project templates include a few integer-only arithmetic operators. For expressions, a basic pass mark will be achieved if you expand the example to include all of the simple integer operators that are present in Java. An extension-level mark will be achieved if you add in more complex operations such as ++ and –, extend the operations to floating point numbers where appropriate and use booleans rather than integers for predicates. A 'beyond' mark would require you to also implement arbitrary precision arithmetic with a set of casting operations to allow only sensible conversions - hence some type checking would be required.

For the control flow facet: the project templates only support a basic while loop and an if-then-else statement. A basic pass mark will be achieved if you expand the project template to include other kinds of loop. An extension level mark will be achieved if you add a case (switch) statement and procedure call. A 'beyond' mark would require more subtle forms of control flow such as exceptions, lambdas or co-routines.

For the typing facet:

B.4 Teaching week by week

So as to help you prepare your project submission in a timely manner, we approach the learning outcomes in this order.

Week	Chapter	Lecture A	Lecture B
1	1&2	Software language processors	The course; models of program execution
2	3&5	Simple string and tree rewrites	SOS arithmetic, variables, big and little step
3	5	SOS sequencing, conditionals and loops	Internal syntax
4	5&4	__user and the plugin	Context free grammars and GIFT
5	4	Parsing - RDSOB	General parsing and lexing
6	4&6	Choosers	Attribute grammar history and theory
7	6	Attribute action systems	Mini series 1
8	6	Mini series 2	Function call, types, static vs dynamic
9	7	Pragmatics 1	Pragmatics 2
10	ARTInt	ART internals	ART internals
11		(Contingency)	(Contingency)

Week	Section	Lab sessions and project submission	
1	C.1	Lab 1 - OpenSCAD	
2	C.2	Lab 2 - rewriting	
3	C.3	Lab 3 - eSOS 1	
4	C.4	Lab 4 - internal syntax and backend actions	
5	C.5	Lab 5 - Context free grammars and GIFT	
6		Part A project support	
7	C.6	Lab 6 - attribute action systems	**Project part A submission in week 7**
8	C.7	Lab 7 - delayed attributes and control flow	
9		Part B project support	
10		Part B project support	
11		No lab	**Project part B submission in week 11**

B.5 Protocol for asking questions

You may ask for help over email, but I will need to be able to reproduce your issue on my machine. Please send a single email, the body of which should contain a concise explanation of your concern with copies of the relevant deliverables as text attachments.

Do not send scripts as screenshots since I cannot run those; you must send scripts as text attachments to your emails.

Appendix C

Lab exercises

C.1 Solid modelling with OpenSCAD

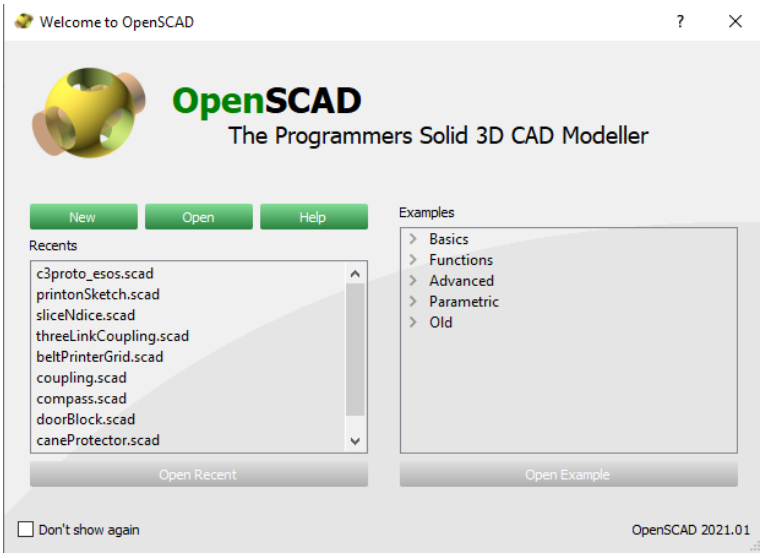
This laboratory session aims to help you understand the users' view of Domain Specific Languages. We shall look at the domain of solid modelling languages which may be used, amongst other things, to create objects suitable for 3D printing.

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?

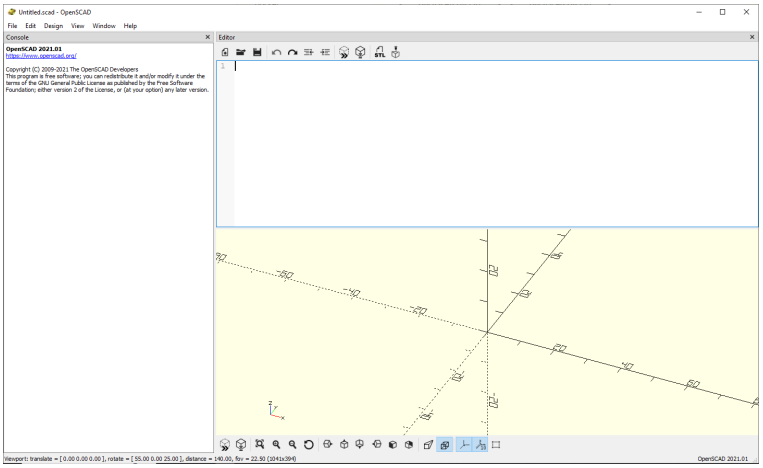
Install OpenSCAD for your system by accessing the download page at

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:



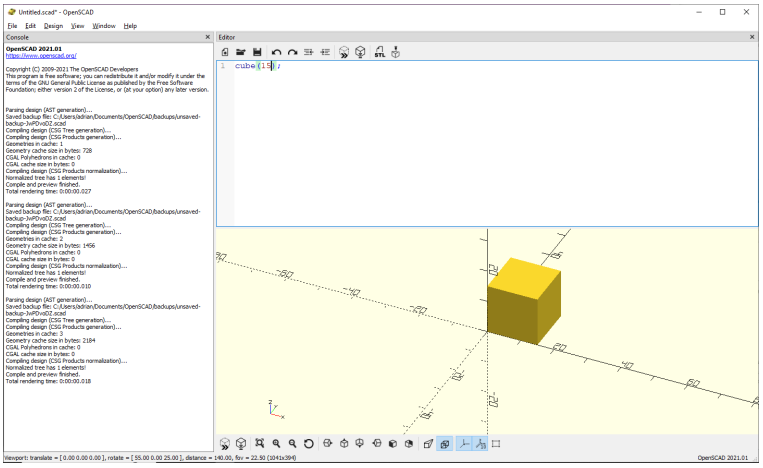
You see three windows: to the left a console and to the right a text editor above a graphics display window.

C.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:



C.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.

C.1.3 Changing size and colour

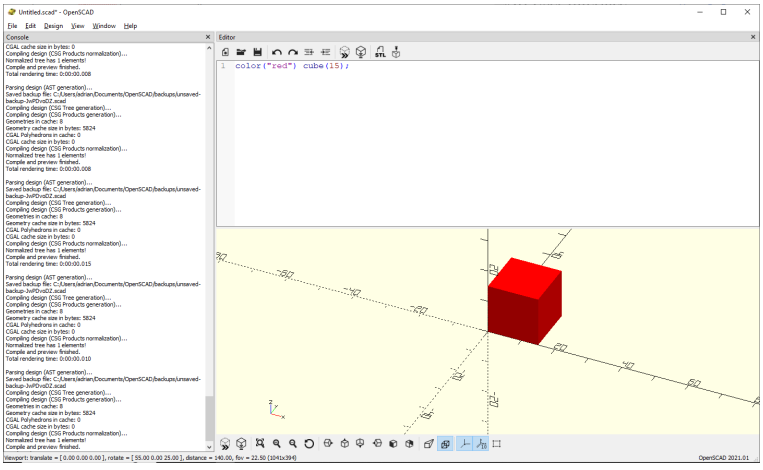
The numeric argument sets the length of the cube's edge. Change it to 25 and re-view with F5. The cube will become larger.

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:

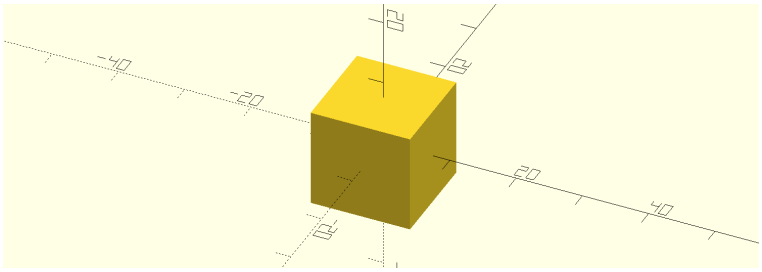


C.1.4 Where is the centre?

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):

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

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

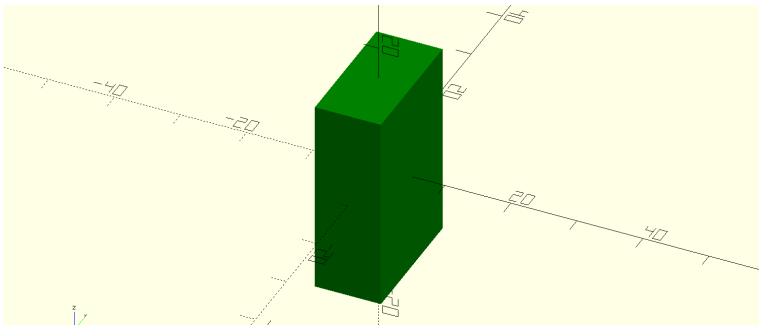


C.1.5 Cubes are really cuboids

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

```
1 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]`.

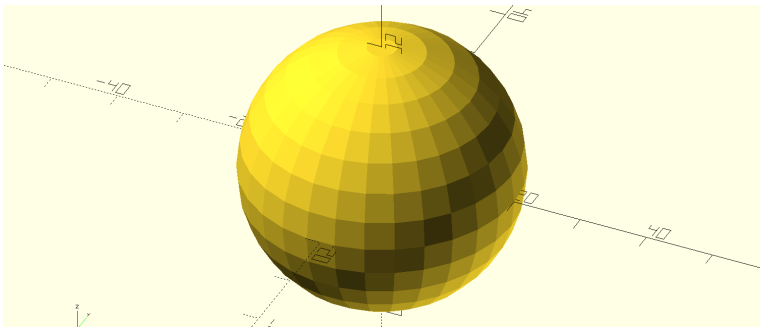


C.1.6 Spheres

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

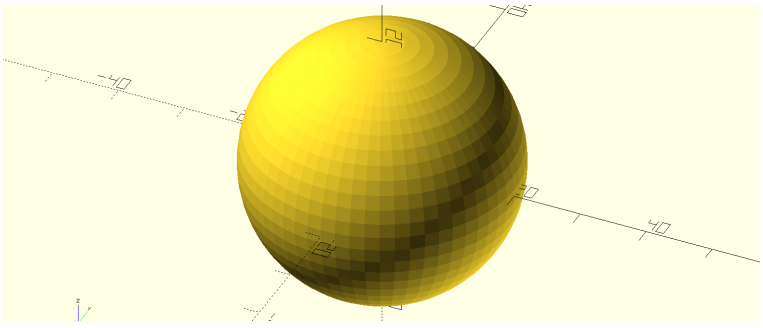
We can model spheres as polyhedra with sufficient faces to make the surface look smooth. The default for a sphere is only 30, which looks blocky if the sphere is large. The `sphere()` primitive constructs a spherical mesh:

```
1 sphere(20);
```



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

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



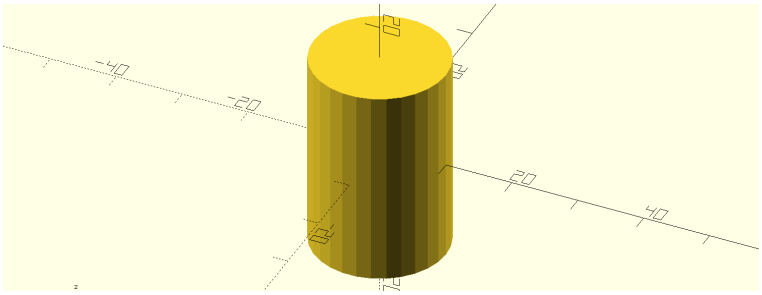
Try raising the value of `$fn` to 120, and then to 360. You will see that the sphere gets much smoother.

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.

C.1.7 Cylinders and polyhedral bars

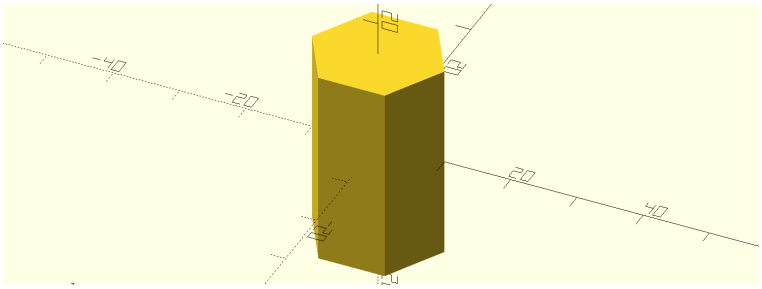
The `cylinder()` primitive takes a height `h` and a radius `r`:

```
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()`?

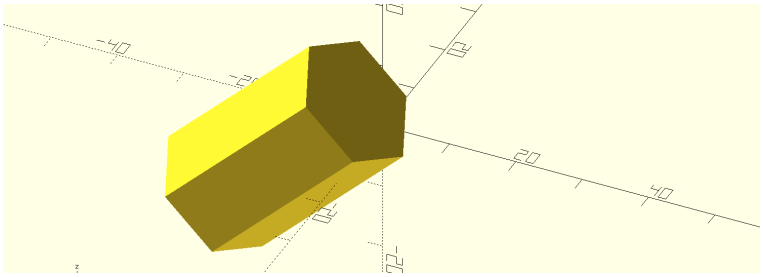
C.1.8 Translation and rotation

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.

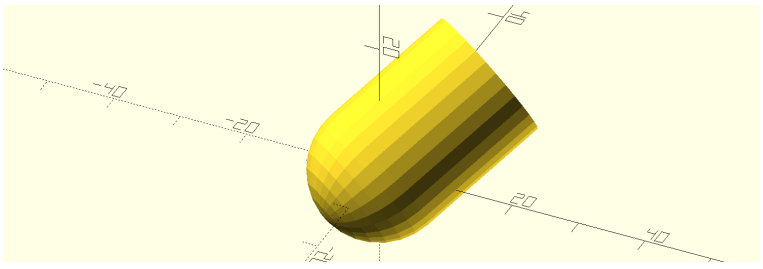


Rotations always occur around axes, so you will get very different effects if you rotate and then translate (as here) as opposed to translating and then rotating. The translate and rotate operations work like prefix operators, so they are effectively executed right-to-left. Experiment with changing the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

C.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

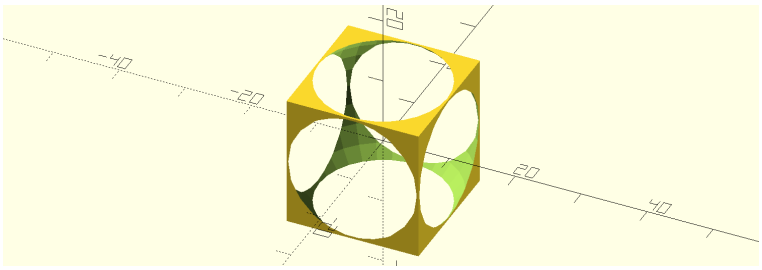
```
rotate([−45,45,0])
union() {
   cylinder(h=20, r = 10);
   sphere(10);
}
```



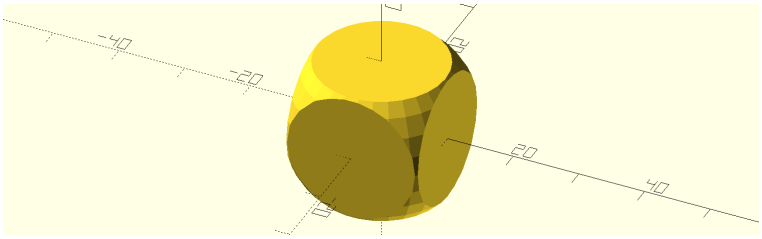
C.1.10 Computational solid geometry

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);
}
```



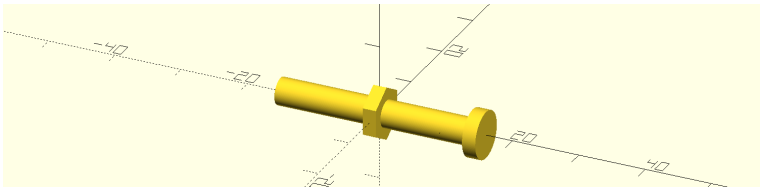
The `difference()` operation is widely used to make holes in objects by subtracting a cylinder from them.

C.1.11 Using modules to structure a design

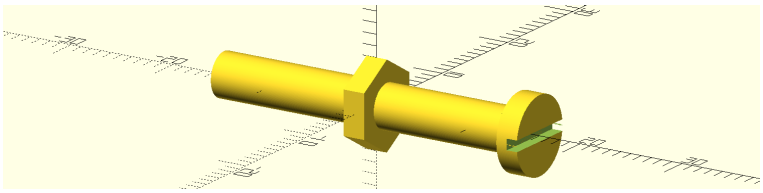
When programming, we use functions, procedures and methods to break our task down into manageably small pieces. In OpenSCAD, the equivalent construct is the *module* which wraps some 3D object descriptions up together with an implicit union, and allows arguments to *parameterise* the resulting objects.

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.

```
1  module nut() { cylinder(h=2, r = 4, $fn=6, center=true); }
2
3  module bolt(length) {
4    cylinder(h=length, r=2, $fn=200, center=true);
5    translate([0,0,length/2]) cylinder(r=3.5, h=2, $fn=200, center=true);
6  }
7
8  rotate([0,90,0]) union() {
9    nut();
10   bolt(30);
11 }
```



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

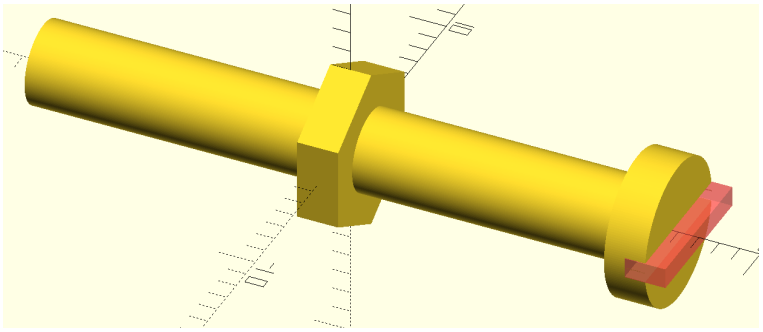


Hint: you need to subtract a cuboid from the larger of the two cylinder above.

C.1.12 The # 'ghost' modifier

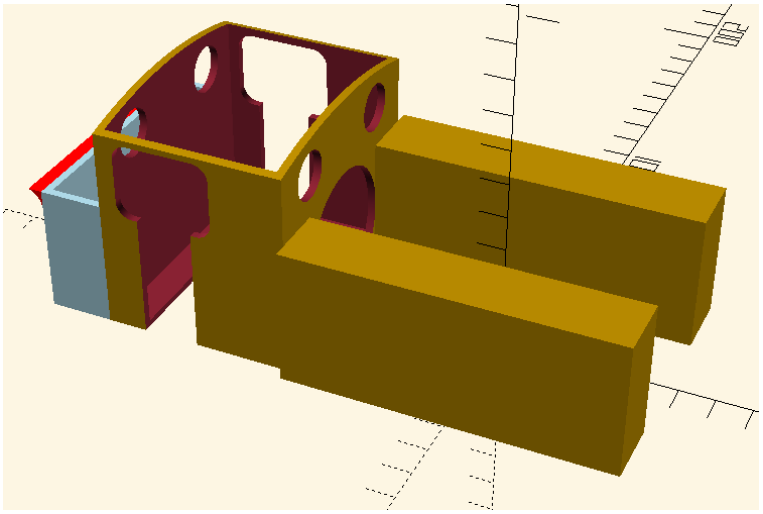
When trying to use the difference and intersection commands, it can be hard to visualise what is going on as some of the elements will be invisible.

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:

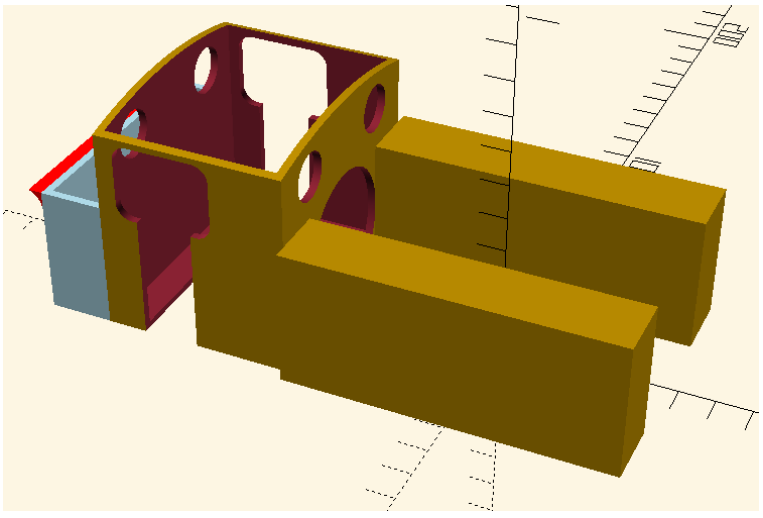


C.1.13 Your exercise: the J69

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



and here is a completed model:

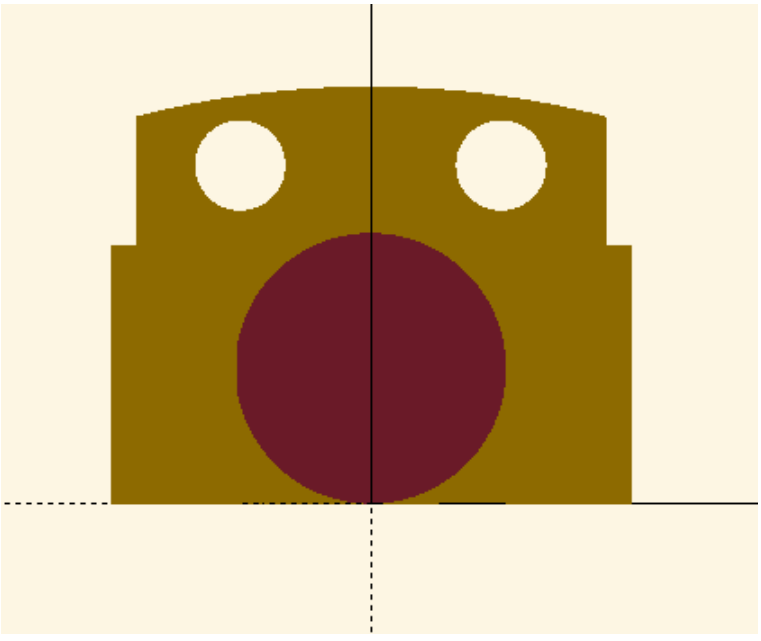


It is based on a prototype, real world engine called the J69 class. You can read more about them at https://www.lner.info/locos/J/j67j69.php. There is just one real example left, shown here at Bressingham:

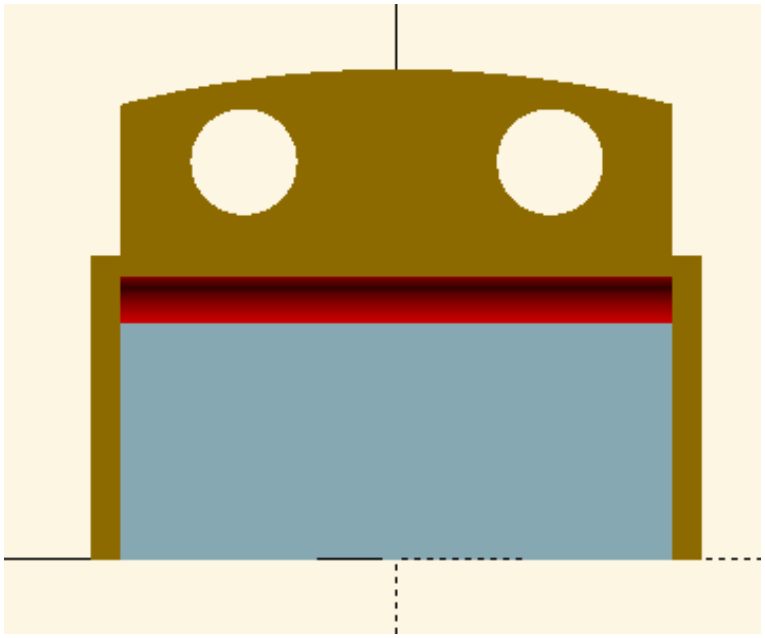
Your task now is to build your own version of the simple steam engine body, and if you are so inclined to embellish it so that it captures more of the prototype. Begin by defining the tanks, which are simple cuboids, and then make the cab as a cuboid which has another cuboid subtracted from it to make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

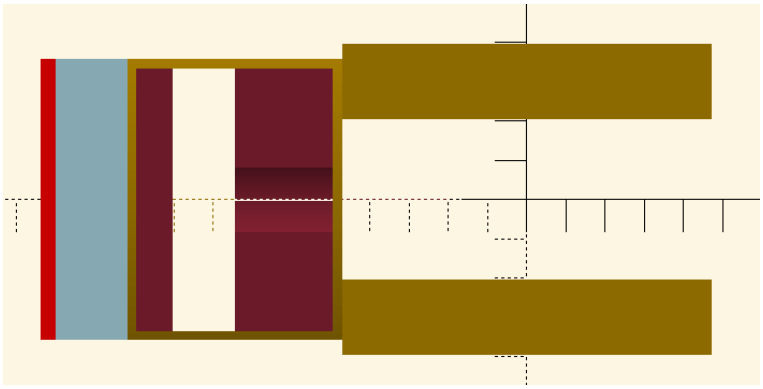
Front:



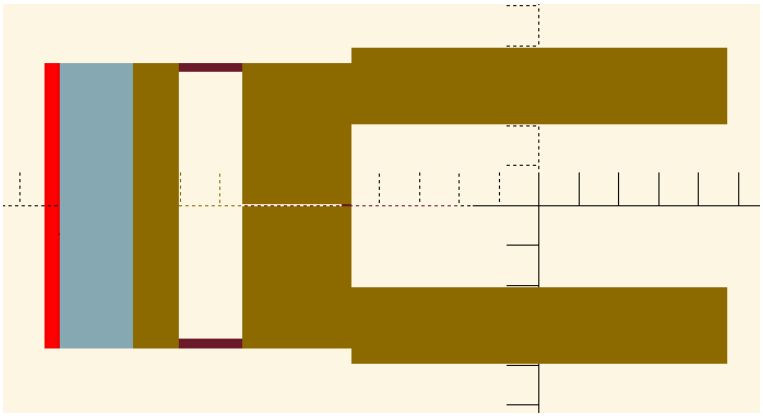
Rear:

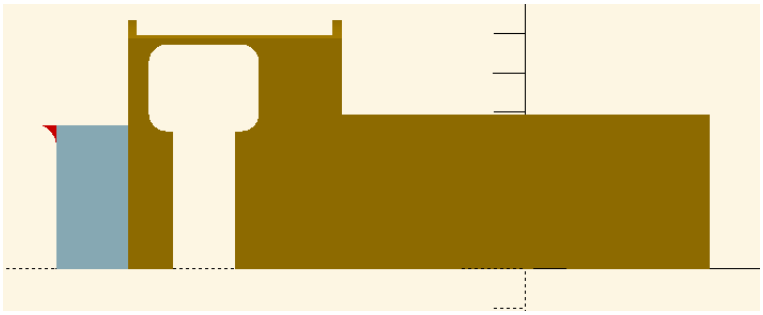


Above:



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.

Appendix D

Music making with Java MIDI

In this section we lay the foundations for project work in music by looking at elementary aspects of western music, and exploring the capabilities of Java's built in synthesizer. Our goal is to build a domain specific language which allows convenient specification, performance, and display of musical pieces.

When working with the Java sound API, please use headphones or earbuds so as not to disturb other people in the lab.

D.1 Musical instruments

Broadly speaking, music is a form of structured sound. Sound itself is our perception of vibrations in the air which create sympathetic vibrations of our ear drums, and via the workings of the inner ear into changes in brain activity. A young human can perceive frequencies between about 20 and 20,000 cycles per second (written 20-20kHz) though maximum sensitivity is between 2kHz and 5kHz. We perceive different frequencies as different pitches: an increase or decrease in frequency is perceived as an increase or decrease in pitch.

A musical instrument is a device for creating structured sound. Physical objects display frequencies at which they 'want' to vibrate: we call these their resonant frequencies. If you have ever pushed somebody on a swing you will understand resonance: the swing has a frequency at which it naturally arcs back and forth, and if you give a little push just at the top of the arc then you can maintain steady smooth motion with little effort. If on the other hand you shove the swing before it reaches the high point, then the motion can become very irregular, and even cause the person on the swing to be thrown off. It turns out that you can change the resonant frequency of a swing by changing the length of the ropes holding the seat: longer ropes give a slower swing frequency.

It is a general rule that large physical objects resonate at lower frequencies than small physical objects, and that if you want to keep an object vibrating whilst using as little energy as possible, then you should stimulate it at its resonant frequency. This is, perhaps, why tiny humming birds' wings move so quickly that we perceive the disruption in the air as a hum, whilst an albatross with a 3m wingspan beats its wings slowly.

If we tension a string and pluck it, then it will vibrate at its resonant frequency. If we shorten the string, or increase its tension. or replace it with a lighter string,

then the resonant frequency will increase. These are the principles behind stringed instruments including guitars, violins and pianos.

If we take an open bottle and blow across the top of it then the air in it will resonate. By using a smaller bottle (or perhaps two identical bottles with one half full of water) we can try smaller mass of air which will resonate at a higher frequency. This is the operating principle of woodwind instruments.

There are many variations on these basic themes: strings may be plucked, hit with hammers or excited with a bow. Air resonators may be fed with pumped air (as in a pipe organ) or blown into; their resonant frequencies may be modified by opening and closing valves between air spaces or by deforming the resonator, as in a trombone. For stringed instruments in particular, it is common to provide a coupled mass of air in a hollow *sound box* that will resonate in sympathy with the primary resonator, strengthening the amplitude of the air vibrations. Some modern instruments (such as the electric guitar) directly convert resonator vibrations into electrical signals which can be amplified, transmitted long distances and fed to a loudspeaker; the microphone is a general device for converting air pressure waves into an electrical signal.

Even the highest perceivable sound frequency is very low compared to the instruction execution frequencies of current computers. At 20kHz, each sound cycle lasts for $50\mu s$. A modern desktop processor can achieve instruction execution frequencies of more than 2×10^9 Hz, and can thus execute more than 100,000 instructions during each audio cycle. This makes it feasible to use software to generate quite complicated musical waveforms in real time. Such a device is called a digital music synthesizer, and the standard Java distribution ships with libraries for music synthesis.

Ensuring the correct synchronisation between multiple synthesized instruments and input devices such as keyboards requires careful coding and protocol design: in 1982 the *Musical Instrument Digital Interface* standard was published, and this has become a very broadly implemented system for controlling musical instruments. The Java distribution contains classes which may be used with MIDI controllers (such as keyboards) to make entirely electronic music. We shall give examples of the use of this library; and we shall write a very simple Domain Specific Language is to give the non-Java programmer access to these facilities using a small self-contained notation.

D.2 The perception of pitch

When presented with a complex repetitive waveform, the ear resolves it into multiple pitches, that is we can perceive the separate frequencies individually. In this respect, the ear is fundamentally different to the eye which 'averages' frequencies: when presented with a patch of green light overlaying a patch of red light, we perceive yellow. The separation of a waveform into its constituent frequencies is called Fourier Analysis: the ear effectively performs Fourier Analysis whereas the eye merges frequencies.

As we increase frequency, we hear an ascending pitch. Interestingly, and fundamentally, the ear perceives frequencies which are integer multiples of one another as 'the same but different' and nearly all music systems use this observation to split the frequency spectrum up into a sequence of 'octaves': one octave is the range of musical pitches between some frequency f and the frequency $2f$. This perception of frequency doubling naturally leads to a log-style description of pitch.

Some instruments offer a continuous range of frequencies (examples include the Theramin, fretless stringed instruments and the human voice) but most instruments provide a finite set of discrete pitches which may be based on individual tuned resonators (like the strings of a piano) or by discrete adjustments of an otherwise-continuous resonator (like the frets used in a guitar). When using a computer to generate audio waveforms, there are no constraints at all, but in practice we often use the computer to make sounds like traditional instruments.

D.3 The physics and psychology of pitch

Music appreciation is highly culturally conditioned, and perceptions of 'satisfying' music vary geographically and over time. As computer scientists, we understand how to systematise and implement behaviours which 'make sense' to our users by hiding low-level complexity and only providing control over high level concepts: for instance, when we connect our laptops to a network, we do not expect users to understand the complexities of the network protocol, or even to know the numeric address of the machine they are connecting to. Similarly, musicians organise the continuum of available frequencies into a set of conventions which 'feel right'; and growing up within a particular musical culture these conventions we may come to feel in some sense natural and fundamental. However, we must never lose sight of the fact that alternative conventions may be just as natural to people growing up in other cultures.

Most western music is organised around the twelve-tone-equal-temperament scale (12-TET) in which an octave is divided into twelve discrete frequencies. The ratio between octaves is 2; the ratio between two successive notes (called a *semitone* is thus $\sqrt[12]{2}$. Each discrete frequency is called a note. Western keyboard instruments such as the piano have individual resonators tuned to each note, and a key which when pressed causes that note to sound. Some instruments provide only a subset of the available notes in an octave, so to avoid these complications we shall use the keyboard as a reference instrument.

We noted before that the ear is particularly sensitive to frequencies up to about 5kHZ. If we start at, say, 20Hz and generate tones by multiplying by $\sqrt[12]{2}$, we get to 6kHz in around 100 steps, so we shouldn't be surprised to find that large concert pianos have 88 keys, and that nonstandard pianos have been constructed with 102 keys. Of course, we could have started at 25Hz or 19Hz. The particular mapping between the discrete notes and the continuum of frequencies is called the *tuning* of an instrument. For an equal tempered tuning such as 12-TET, it suffices to pick one particular frequency for one particular note: the other notes

are then fixed by the $\sqrt[12]{2}$ ratio between semitones.

Current orchestral practice is to use 440Hz as a tuning standard and to tune the 49th key of a standard 88 note keyboard to to that frequency. This then results in the leftmost key generating 27.5Hz, and rightmost key generating 4186.01Hz. (A 102 key keyboard, rarely implemented, ranges from 16.3516Hz to 5587.65Hz.)

The MIDI standard defines 128 notes numbered from zero to 127, with twenty notes below and twenty notes above the standard piano keyboard. Key zero generates 8.17Hz and key 127 12,543.85Hz. Key 69 gives the 440Hz concert tuning standard.

A note on alternative tunings

There is nothing inherently perfect about this particular tuning. The 440Hz standard was internationally agreed in 1939, and became an ISO standard in 1955. However frequencies from 400Hz to 460hz are known to have been used historically, and those frequencies are more more than $\sqrt[5]{2}$ apart, which is greater than two semitones: playing an historical piece to modern tuning may therefore yield a performance that differs significantly from the composer's intent.

Apart from these shifts in reference frequency, equal temperament (the division of the octave using a constant ratio) is not the only way of mapping discrete notes on to the frequency continuum. Many musical traditions instead use ratios of small integers, since frequencies related in his way form harmonious combinations: these tunings are collectively called *just intonation* and arise naturally with certain classes of instrument. The ratio 3:2 forms the basis of the so-called *Pythagorean tuning*; many other systems exist.

The division of the octave into 12 notes is also merely a convention. Divisions into 15, 19, 34 and even 53 notes have been studied; one way of thinking about this is that by having more notes we can more closely approach the small-integer-ratio harmonics of just intonation. There is a vast literature on tunings which you can begin to explore through online articles: in the rest of this section we shall restrict ourselves to 12-TET.

D.4 Pure tones and instrument voices

The resonators used in musical instruments generate more than one frequency. Roughly speaking, the note that we hear is the lowest frequency produced, but there will usually be many related *overtones* present, usually integer multiples of the lowest frequency. The lowest frequency is called the fundamental; the integer-multiple frequencies are the harmonics. For a fundamental frequency f, the first harmonic has frequency $2f$, the second harmonic $3f$ and so on. Notice how these harmonics are at the octave intervals: hence a fundamental and its

overtones together sound like a single note rather than resolving into several independent notes.

A pure single tone sounds rather other-worldly: a tuning fork or the human whistle is probably the closest thing to a pure tone resonator that most people hear. Of course, we can use the computer to generate pure tones, once we know the waveform.

Fourier analysis and synthesis

In 1822, Jean-Baptiste Joseph Fourier published a claim that any periodic waveform could be decomposed into a (possibly infinite) set of sinusoidal waveforms, which when added together would reconstruct the original waveform. This observation has turned out to have many applications in physics and engineering. It is of direct relevance to sound synthesis, since it suggests that if we can write a program that generates sine waves at different frequencies and then add them together in various proportions we can numerically construct any audio waveform. This is the principle behind music synthesizers. Interestingly, there is a procedure by which we can take an arbitrary waveform and numerically decompose it into its constituent sine waves. We call a display of the strength of each frequency a *spectrum analysis*.

If our musical instrument resonator produces only a fundamental and harmonics, then we can characterise the sound of, say, a piano string by listing the proportions of sine waves of frequencies f, $2f$, $3f$,... kf where k is chosen to be beyond the limit of human hearing. Even for a very low fundamental note such as 20Hz, the tenth harmonic exceeds 20kHz. Hence we have the prospect of being able to accurately encode the detailed sound of a piano note into eleven real numbers, and then reconstituting the sound in realtime using software.

The conversion of a waveform to a spectrum display of frequencies is called Fourier Analysis; the reverse process of converting a series of proportions of sine waves to a single waveform is called Fourier Synthesis. We sometimes refer to operations on waveforms as 'working in the time domain' and operations on frequency proportions as 'working in the frequency domain'.

The Nyquist-Shannon criterion and making recordings

An ability to encode a single note of a single instrument accurately is clearly important for synthesis. However, it does not tell us how to capture the behaviour of an entire orchestra, or indeed how to encode non-musical sounds.

We can use a computer as a sound recorder by connecting a microphone and then measuring its output, say every microsecond. We use an Analogue to Digital Converter (ADC) to convert the voltage developed by the microphone into a number. Typical high quality digital audio systems use around 16 bits to repre-

sent each sample. By storing the resulting sequence of integer numbers, we can have a permanent record of a sound experience which may be reconstructed by converting the numbers back into voltages and applying the resulting waveform to a loudspeaker.

We can see that if we sample the original sound too slowly, then we may lose information. On the other hand, if we sample at very high speed then we shall require extra storage. A fascinating result which arises from Fourier analysis is that we can capture the full detail of any waveform, no matter how complex, up to some bounding frequency f by sampling the waveform at no more than $2f$. This is why CD quality audio samples waveforms at 44.1kHz: since the human ear can perceive sounds up to 20kHz, a sample rate of greater then 40kHz ensures that no information is lost. (In detail, the figure of 44.1kHz arises as a result of early experiments using video recorders to store audio information: you can read the story online.) This $2f$ requirement is called the Nyquist-Shannon criterion.

D.5 Tempo, rhythm and articulation

It is unusual for a note to be played continuously (although bagpipes and some other instruments have a *drone* which sounds continuously during a performance). Instead, the playing time is divided up into discrete beats which set the duration of the basic note.

In the western tradition, a piece of music will have an indicated *tempo*, sometimes expressed as *beats per minute* or bpm. As computer scientists, we might prefer to use Hz to specify the tempo, that is, beats per second. A very slow piece would be below 30bpm and a very fast piece above 200bpm, from which we can see that beat frequencies range from around 0.5 to 3.5Hz.

**** Todo: Rhythms**

Much of the character of a performance is embedded in the detailed way in which a performer uses the tempo. A straightforward approach is to leave a very short silence at the end of each beat period, and to sound each note uniformly throughout the rest of the beat period. This very simplistic approach is easy to program but sounds, well, synthetic.

A human performer even when attempting uniformity will display some small variations in the length of notes. More significantly, humans players deliberately vary the details of note timing within the basic rhythm framework, a technique known as *articulation*. For instance, some notes may be run together into a smoothly connected frequency shift whereas other are deliberately shortened so as to create a jumpy effect. In wind instruments articulation is achieved by controlling airflow with the tongue; in stringed instruments by dampening the vibrations with the hand. Other forms of articulation include rapid periodic changes in amplitude (called tremolo) and rapid periodic changes in pitch (called vibrato). A slower shift in pitch is often called a *pitch bend*: some electric

guitars (such as the fender Stratocaster) have an arm which allows the tension and length of the strings to be varied – this device is often called a tremolo bar (although really it is a vibrato bar).

D.6 Musical terminology for pitch

Musicians use a very large number of technical terms, and this can be rather overwhelming at a first encounter. However, we are interested in Domain Specific Languages, and musician's terminology certainly represents a very widely used language which is extremely domain specific, and as such can be the basis of some interesting case studies.

Musical nomenclature has grown up over a long historical period, and can seem rather arbitrary to outsiders even though there is usually an underlying logic. For instance, one might imagine that the divisions of an octave into 12-semitones might be represented by twelve unique names. In fact, in the western tradition there are seven unique names (the letters A through G inclusive), and two modifiers ♯ and ♭ (spoken sharp and flat) which raise (lower) by a semitone the note represented by a name.

This initially surprising naming convention arises from the observation that certain sequences of seven notes sound harmonious, and that the majority of western musical melodies are *mostly* constructed around seven note selections.

By appending a ♯ or a ♭ symbol to the seven basic names we can name the 'missing' five semitones. A conventional way of writing an ascending sequence of 12 semitones in an octave is:

A A♯ B C C♯ D D♯ E F F♯ G G♯

and a conventional way to write a descending sequence is:

A A♭ G G♭ F E E♭ D D♭ C B B♭

Using the 12-TET tuning (but not necessarily for other tunings), A♯ and B♭ represent the same frequency, and we can enumerate the full set of notes in an octave as

A A♯/B♭ B C C♯/D♭ D D♯/E♭ E F F♯/G♭ G G♯

**** Todo: Octave numbers**

D.7 Major and minor scales

Our basic pitch palette, then, comprises octaves of 12 fundamental notes each separated by a semitone. When played, notes come with a variety of harmonics which allow us to distinguish, say, a violin note from a guitar note.

Music can generate emotional responses in humans. It is clear that these responses are culturally conditioned, but nevertheless within a culture such as our

own in which individuals are exposed to many musical pieces, strong associations between particular musical progressions and particular emotional states seem to be almost universally recognised — in the western tradition the difference between a joyous and a sad piece is well understood by most listeners.

The first component of mood is harmony. Some sequences of notes sound harmonious and some do not: we say they are discordant (which literally means that they disagree with each other). We can test our response to sequences by playing subsets of the 12 tones in an octave in ascending or descending order: it turns out that some sound good and some are unpleasant. Such a sequenced subset of the tones is called a *scale*.

If we think of the 12 semitones laid out as a 12-bit vector representing the presence or absence of a note within some scale, it is easy to see that there are $2^{12} = 4096$ scales. The one with all twelve notes in it is called the chromatic scale; its dual with no notes in at all is simply silence (and therefore of little musical utility).

We have already noted that western music focuses on scales with seven notes. It turns out that only two families of such seven note scales find wide application in popular music. The *major* scales start with any of the twelve notes and then include notes according to the increments

$$+2\ +2\ +1\ +2\ +2\ +2\ +1$$

The *minor* scales begin with any note, and then include notes according to the increments:

$$+2\ +1\ +2\ +2\ +1\ +3\ +1$$

Scales are usually played so as to finish one octave above the root. Hence, we play a major scale rooted on keyboard key k by playing the keys

$$k, k+2, k+4, k+5, k+7, k+9, k+11, k+12$$

and a minor scale with

$$k, k+2, k+3, k+5, k+7, k+8, k+11, k+12$$

As a further example, consider the scale made up of two equally spaced notes. These are three whole tones apart and thus called a tritone. When played, this creates, at best, a sense of tension: some might even say it sounds wrong.

D.8 Chords

A chord is a set of notes played simultaneously. Just as with scales, particular combinations sound harmonious, and the most common way of forming a chord

for a root note k is to take the first, third and fifth elements of either the major or minor scale rooted on k. We write CM (spoken C-major) for the chord rooted on C using the major scale, and Cm (C-minor) for the chord rooted on C using the minor scale.

Chords formed of notes $k, k+6, k+12$ sound particularly inharmonious, made up as they are of a pair of tritones.

We can play a simple melody by picking out single notes. If we replace each note by the major chord rooted at that note then we get a fuller sound. We can do the same with the minor chords; and in general a melody and chords based on the minor scale will sound darker and perhaps more gentle: the major scale sounds brighter.

D.9 Synthesizing music with Java and MIDI

**** Todo: Overview of MIDI**

```
1  package uk.ac.rhul.cs.csle.artmusic;
2
3  import javax.sound.midi.MidiChannel;
4  import javax.sound.midi.MidiSystem;
5  import javax.sound.midi.Synthesizer;
6
7  public class ARTMiniMusicPlayer {
8    private Synthesizer synthesizer;
9    private MidiChannel[] channels;
10   private int defaultOctave = 5;
11   private int defaultVelocity = 50;
12   private int bpm;
13   private double bps;
14   private double beatPeriod;
15   private double beatRatio = 0.9;
16   private int beatSoundDelay = (int) (1000.0 * beatRatio / bps);
17   private int beatSilenceDelay = (int) (1000.0 * (1.0 - beatRatio) / bps);
18
19   public ARTMiniMusicPlayer() {
20     try {
21       System.out.print(MidiSystem.getMidiDeviceInfo());
22       synthesizer = MidiSystem.getSynthesizer();
23       synthesizer.open();
24       channels = synthesizer.getChannels();
25     } catch (Exception e) {
26       System.err.println("miniMusicPlayer exception: " + e.getMessage());
27       System.exit(1);
28     }
29
```

```
30      setBeatRatio(0.9);
31      setBpm(100);
32      setDefaultVelocity(50);
33    }
34
35    public int getDefaultOctave() {
36      return defaultOctave;
37    }
38
39    public void setDefaultOctave(int defaultOctave) {
40      this.defaultOctave = defaultOctave;
41    }
42
43    public int getDefaultVelocity() {
44      return defaultVelocity;
45    }
46
47    public void setDefaultVelocity(int defaultVelocity) {
48      this.defaultVelocity = defaultVelocity;
49    }
50
51    public int getBpm() {
52      return bpm;
53    }
54
55    public void setBpm(int bpm) {
56      this.bpm = bpm;
57      bps = bpm / 60.0;
58      beatPeriod = 1000.0 / bps;
59      beatSoundDelay = (int) (beatRatio * beatPeriod);
60      beatSilenceDelay = (int) ((1.0 - beatRatio) * beatPeriod);
61    }
62
63    public void setBeatRatio(double beatRatio) {
64      this.beatRatio = beatRatio;
65      beatSoundDelay = (int) (beatRatio * beatPeriod);
66      beatSilenceDelay = (int) ((1.0 - beatRatio) * beatPeriod);
67    }
68
69    private int noteNameToMidiKey(String n, int octave) {
70  // @formatter:off
71  int key = octave * 12 +
72          ( n.equals("C") ? 0
73          : n.equals("C#") ? 1
74          : n.equals("Db") ? 1
75          : n.equals("D") ? 2
76          : n.equals("D#") ? 3
```

```
77            : n.equals("Eb") ? 3
78            : n.equals("E") ? 4
79            : n.equals("F") ? 5
80            : n.equals("F#") ? 6
81            : n.equals("Gb") ? 6
82            : n.equals("G") ? 7
83            : n.equals("G#") ? 8
84            : n.equals("Ab") ? 8
85            : n.equals("A") ? 9
86            : n.equals("A#") ? 10
87            : n.equals("Bb") ? 10
88            : n.equals("B") ? 11
89            : -1);
90    // @formatter:on
91
92        if (key < 0 || key > 127) {
93          System.err.println("miniMusicPlayer exception: attempt to access out of range MIDI key " + n + octave
94          System.exit(1);
95        }
96        return key;
97      }
98
99      // Silence
100     public void rest(int beats) {
101       try {
102         Thread.sleep((long) (beats * beatPeriod));
103       } catch (InterruptedException e) {
104         /* ignore interruptedException */ }
105     }
106
107     // Single notes
108     public void play(int k) {
109       try {
110         channels[0].noteOn(k, defaultVelocity);
111         Thread.sleep(beatSoundDelay);
112         channels[0].noteOn(k, 0);
113         Thread.sleep(beatSilenceDelay);
114       } catch (InterruptedException e) {
115         /* ignore interruptedException */ }
116     }
117
118     public void play(String n) {
119       play(noteNameToMidiKey(n, defaultOctave));
120     }
121
122     public void play(String n, int octave) {
123       play(noteNameToMidiKey(n, octave));
```

```
124   }
125
126   // Arrays of notes
127   public void play(int[] k) {
128     try {
129       for (int i = 0; i < k.length; i++)
130         channels[1].noteOn(k[i], defaultVelocity);
131       Thread.sleep(beatSoundDelay);
132       for (int i = 0; i < k.length; i++)
133         channels[1].noteOn(k[i], 0);
134       Thread.sleep(beatSilenceDelay);
135     } catch (InterruptedException e) {
136       /* ignore interruptedException */ }
137   }
138
139   public void playSequentially(int[] k) {
140     try {
141       for (int i = 0; i < k.length; i++) {
142         channels[i].noteOn(k[i], defaultVelocity);
143         Thread.sleep(beatSoundDelay);
144         channels[i].noteOn(k[i], 0);
145         Thread.sleep(beatSilenceDelay);
146       }
147     } catch (InterruptedException e) {
148       /* ignore interruptedException */ }
149   }
150
151   // Scales
152   public void playScale(String n, ARTScale s) {
153     playScale(noteNameToMidiKey(n, defaultOctave), s);
154   }
155
156   public void playScale(String n, int octave, ARTScale s) {
157     playScale(noteNameToMidiKey(n, octave), s);
158   }
159
160   public void playScale(int k, ARTScale s) {
161     int[] keys;
162     switch (s) {
163     case CHROMATIC:
164       keys = new int[] { k, k + 1, k + 2, k + 3, k + 4, k + 5, k + 6, k + 7, k + 8, k + 9, k + 10, k + 1
165       break;
166
167     case MAJOR: // TTSTTTS
168       keys = new int[] { k, k + 2, k + 4, k + 5, k + 7, k + 9, k + 11, k + 12 };
169       break;
170
```

```
171       case MINOR_NATURAL: // TSTTSTT
172         keys = new int[] { k, k + 2, k + 3, k + 5, k + 7, k + 8, k + 10, k + 12 };
173         break;
174       case MINOR_HARMONIC: // TSTTS3S
175         keys = new int[] { k, k + 2, k + 3, k + 5, k + 7, k + 8, k + 11, k + 12 };
176         break;
177       case MINOR_MELODIC_ASCENDING: // TSTTS3S – harmonic with with sixth sharpened
178         keys = new int[] { k, k + 2, k + 3, k + 5, k + 7, k + 9, k + 11, k + 12 };
179         break;
180       case MINOR_MELODIC_DESCENDING: // TSTTS3S – harmonic with seventh flattened making it the sa
181         keys = new int[] { k + 12, k + 10, k + 8, k + 7, k + 5, k + 3, k + 2, k };
182         break;
183
184       default:
185         keys = new int[] { 0 };
186         break;
187       }
188       playSequentially(keys);
189     }
190
191     // Programmed chords
192     public void playChord(String n, ARTChord type) {
193       playChord(noteNameToMidiKey(n, defaultOctave), type);
194     }
195
196     public void playChord(String n, int octave, ARTChord type) {
197       playChord(noteNameToMidiKey(n, octave), type);
198     }
199
200     public void playChord(int k, ARTChord type) {
201       int[] keys;
202       switch (type) {
203       case NONE:
204         keys = new int[] { k };
205         break;
206       case MAJOR:
207         keys = new int[] { k, k + 4, k + 7 };
208         break;
209       case MAJOR7:
210         keys = new int[] { k, k + 4, k + 7, k + 11 };
211         break;
212       case MINOR:
213         keys = new int[] { k, k + 3, k + 7 };
214         break;
215       case MINOR7:
216         keys = new int[] { k, k + 4, k + 7 };
217         break;
```

```
218       default:
219         keys = new int[] { 0 };
220         break;
221       }
222       play(keys);
223     }
224
225     private void tune() {
226       int base = 47;
227       play(base + 14);
228       play(base + 12);
229       play(base + 11);
230       play(base + 7);
231       play(base + 5);
232       play(base + 7);
233       play(base + 2);
234       rest(2);
235     }
236
237     private void tuneChordMajor() {
238       int base = noteNameToMidiKey("C", 5);
239       playChord(base + 14, ARTChord.MAJOR);
240       playChord(base + 12, ARTChord.MAJOR);
241       playChord(base + 11, ARTChord.MAJOR);
242       playChord(base + 7, ARTChord.MAJOR);
243       playChord(base + 5, ARTChord.MAJOR);
244       playChord(base + 7, ARTChord.MAJOR);
245       playChord(base + 2, ARTChord.MAJOR);
246     }
247
248     private void tuneChordMinor() {
249       int base = noteNameToMidiKey("C", 5);
250       playChord(base + 14, ARTChord.MINOR);
251       playChord(base + 12, ARTChord.MINOR);
252       playChord(base + 11, ARTChord.MINOR);
253       playChord(base + 7, ARTChord.MINOR);
254       playChord(base + 5, ARTChord.MINOR);
255       playChord(base + 7, ARTChord.MINOR);
256       playChord(base + 2, ARTChord.MINOR);
257     }
258
259     public void close() {
260       synthesizer.close();
261     }
262
263     public static void main(String[] args) {
264       System.err.println("miniMusicPlayer test routine");
```

```
265       ARTMiniMusicPlayer mp = new ARTMiniMusicPlayer();
266
267        mp.playScale("C", ARTScale.CHROMATIC);
268        mp.rest(2);
269        String note = "C";
270        int octave = 6;
271        mp.play(note, octave);
272        mp.rest(2);
273        mp.playScale("C", ARTScale.MAJOR);
274        mp.rest(2);
275        mp.playScale("C", ARTScale.MINOR_NATURAL);
276        mp.rest(2);
277        mp.playScale("C", ARTScale.MINOR_HARMONIC);
278        mp.rest(2);
279        mp.playScale("C", ARTScale.MINOR_MELODIC_ASCENDING);
280        mp.playScale("C", ARTScale.MINOR_MELODIC_DESCENDING);
281        mp.rest(2);
282        mp.playChord("C", ARTChord.MAJOR);
283        mp.rest(2);
284        mp.playChord("C", ARTChord.MINOR);
285        mp.rest(2);
286        mp.tune();
287        mp.rest(2);
288        mp.tuneChordMajor();
289        mp.rest(2);
290        mp.tuneChordMinor();
291        mp.rest(2);
292 // Tritone scale and scale
293       mp.playSequentially(new int[] { 50, 56, 62 });
294       mp.rest(2);
295       mp.play(new int[] { 50, 56, 62 });
296       mp.rest(2);
297
298       mp.close();
299    }
300 }
```

D.10 `minimusic` – a DSL to access `MiniMusicPlayer`

```
1 melody sanctuary {
2
3 D+M C+M B+ G F G m D m7
4 }
5
6 x = 3;
7 while x > 0 do { print("x is ", x, "\n"); x = x -1; }
```

```

play sanctuary;
```

```
(*****************************************************************************
*
* miniMusic.art – Adrian Johnstone 18 Februrary 2017
*
*****************************************************************************)
prelude { import java.util.HashMap; import uk.ac.rhul.cs.csle.artmusic.*; }

support {
HashMap<String, Integer> variables = new HashMap<String, Integer>();
HashMap<String, ARTGLLRDTHandle> melodies = new HashMap<String, ARTGLLRDTHandle>();
ARTMiniMusicPlayer mp = new ARTMiniMusicPlayer();
}

whitespace &WHITESPACE
whitespace &COMMENT_NEST_ART
whitespace &COMMENT_LINE_C

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)
                          artEvaluate(statement.statement1, statement1);
                      else
                          artEvaluate(statement.elseOpt1, elseOpt1);
                    } |

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

                    'print' '(' printElements ')' ';' | (* print statement *)

                    'melody' ID statement< { melodies.put(ID1.v, statement.statement1); } |
                    'play' ID ';'
                      { if (!melodies.containsKey(ID1.v))
                            artText.println(ARTTextLevel.WARNING, "ignoring request to play undefined melody
                        else
```

```
44                          artEvaluate(melodies.get(ID1.v), null);
45                        } |
46
47                    '{' statements '}' | (* compound statement *)
48
49                    bpm | defaultOctave | note | chord | rest
50
51  elseOpt ::= 'else' statement | #
52
53  bpm ::= 'bpm' INTEGER { mp.setBpm(INTEGER1.v); }
54
55  beatRatio ::= 'beatRatio' REAL { mp.setBeatRatio(REAL1.v); }
56
57  defaultOctave ::= 'defaultOctave' INTEGER
58    { if (INTEGER1.v < 0 || INTEGER1.v > 10)
59         artText.println(ARTTextLevel.WARNING, "ignoring illegal MIDI octave number " + INTEGER1.v);
60        else
61          mp.setDefaultOctave(INTEGER1.v);
62    }
63
64  note ::= simpleNote chordMode { mp.playChord(simpleNote1.v.trim(), chordMode1.v ); } |
65              simpleNote shifters chordMode { mp.playChord(simpleNote1.v.trim(),
66                mp.getDefaultOctave() + shifters1.v, chordMode1.v); } |
67              simpleNote INTEGER chordMode { mp.playChord(simpleNote1.v.trim(), INTEGER1.v, chordMode1.v
68
69  chordMode <v:ARTChord> ::= # { chordMode.v = ARTChord.NONE; } |
70                             'm' { chordMode.v = ARTChord.MINOR; } | 'm7' { chordMode.v = ARTChor
71                             'M' { chordMode.v = ARTChord.MAJOR; } | 'M7' { chordMode.v = ARTCho
72
73  simpleNote<leftExtent:int rightExtent:int v:String> ::=
74     simpleNoteLexeme { simpleNote.v = artLexeme(simpleNote.leftExtent, simpleNote.rightExtent).trim(); }
75
76  simpleNoteLexeme ::= 'A' | 'A#' | 'Bb' | 'B' | 'C' | 'C#' | 'Db' | 'D' | 'D#' | 'Eb' | 'E' | 'F' | 'F#' | 'Gb' | 'G' |
77
78  shifters<v:int> ::= '+' {shifters.v = 1;} | '−' {shifters.v = −1;} |
79                      '+' shifters {shifters.v = shifters1.v + 1; } |
80                      '−' shifters {shifters.v = shifters1.v − 1; }
81
82  chord ::= '[' notes ']'
83
84  notes ::= note | note notes
85
86  rest ::= '.' { mp.rest(1); } | '..' { mp.rest(2); } | '...' { mp.rest(3); } | '....' { mp.rest(4); }
87
88  printElements ::= STRING_DQ { artText.printf("%s", STRING_DQ1.v); } |
89                    STRING_DQ { artText.printf("%s", STRING_DQ1.v); } ',' printElements |
90                    e0 { artText.printf("%d", e01.v); } | e0 { artText.printf("%d", e01.v); } ',' printElements
```

```
91
92  e0 <v:int> ::= e1 { e0.v = e11.v; } |
93                 e1 '>' e1 { e0.v = e11.v > e12.v ? 1 : 0; } | (* Greater than *)
94                 e1 '<' e1 { e0.v = e11.v < e12.v ? 1 : 0; } | (* Less than *)
95                 e1 '>=' e1 { e0.v = e11.v >= e12.v ? 1 : 0; } | (* Greater than or equals*)
96                 e1 '<=' e1 { e0.v = e11.v <= e12.v ? 1 : 0; } | (* Less than or equals *)
97                 e1 '==' e1 { e0.v = e11.v == e12.v ? 1 : 0; } | (* Equal to *)
98                 e1 '!=' e1 { e0.v = e11.v != e12.v ? 1 : 0; } (* Not equal to *)
99
100 e1 <v:int> ::= e2 { e1.v = e21.v; } |
101                e1 '+' e2 { e1.v = e11.v + e21.v; } | (* Add *)
102                e1 '-' e2 { e1.v = e11.v - e21.v; } (* Subtract *)
103
104 e2 <v:int> ::= e3 { e2.v= e31.v; } |
105                e2 '*' e3 { e2.v = e21.v * e31.v; } | (* Multiply *)
106                e2 '/' e3 { e2.v = e21.v / e31.v; } | (* Divide *)
107                e2 '%' e3 { e2.v = e21.v % e31.v; } (* Mod *)
108
109 e3 <v:int> ::= e4 {e3.v = e41.v; } |
110                '+' e3 {e3.v = e41.v; } | (* Posite *)
111                '-' e3 {e3.v = -e41.v; } (* Negate *)
112
113 e4 <v:int> ::= e5 { e4.v = e51.v; } |
114                e5 '**' e4 {e4.v = (int) Math.pow(e51.v, e41.v); } (* exponentiate *)
115
116 e5 <v:int> ::= INTEGER {e5.v = INTEGER1.v; } | (* Integer literal *)
117                ID { e5.v = variables.get(ID1.v); } | (* Variable access *)
118                '(' e1 { e5.v = e11.v; } ')' (* Parenthesised expression *)
119
120 ID <leftExtent:int rightExtent:int lexeme:String v:String> ::=
121   &ID {ID.lexeme = artLexeme(ID.leftExtent, ID.rightExtent); ID.v = artLexemeAsID(ID.leftExtent, ID.right
122
123 INTEGER <leftExtent:int rightExtent:int lexeme:String v:int> ::=
124   &INTEGER {INTEGER.lexeme = artLexeme(INTEGER.leftExtent, INTEGER.rightExtent);
125     INTEGER.v = artLexemeAsInteger(INTEGER.leftExtent, INTEGER.rightExtent); }
126
127 REAL <leftExtent:int rightExtent:int lexeme:String v:double> ::=
128   &REAL {REAL.lexeme = artLexeme(REAL.leftExtent, REAL.rightExtent);
129     REAL.v = artLexemeAsInteger(REAL.leftExtent, REAL.rightExtent); }
130
131 STRING_DQ <leftExtent:int rightExtent:int lexeme:String v:String> ::=
132   &STRING_DQ {STRING_DQ.lexeme = artLexeme(STRING_DQ.leftExtent, STRING_DQ.rightExtent);
133     STRING_DQ.v = artLexemeAsString(STRING_DQ.leftExtent, STRING_DQ.rightExtent); }
```

Appendix E

Image processing operations

Appendix F

3D modelling

Appendix G

Example listings

These listings are available as individual files from

https://github.com/AJohnstone2007/ART

and are included here for completeness.

G.1 From Section 1.5: SLEWAAmbiguityExamples.java

```
1  //@formatter:off Disable formatting in Eclipse so as to retain confusing spacing
2  package slewaExamples;
3
4  public class SLEWAAmbiguityExamples {
5    public static void main(String[] args) {
6      int x, y, z;
7
8      x = 4;
9      y = 6;
10     z = x---y;
11     System.out.println("After x---y:" + " x=" + x + " y=" + y + " z=" + z);
12
13     x = 4;
14     y = 6;
15     z = x-- - y;
16     System.out.println("After x-- - y:" + " x=" + x + " y=" + y + " z=" + z);
17
18     x = 4;
19     y = 6;
20     z = x - --y;
21     System.out.println("After x - --y:" + " x=" + x + " y=" + y + " z=" + z);
22
23     x = 4;
24     y = 6;
25     z = x - - -y;
26     System.out.println("After x - - -y:" + " x=" + x + " y=" + y + " z=" + z);
27
28     System.out.println("5-4-3 is " + (5 - 4 - 3));
29     System.out.println("(5 - 4) - 3 is " + ((5 - 4) - 3));
30     System.out.println("5 - (4 - 3) is " + (5 - (4 - 3)));
```

```
31
32      y = 6; if (x > 3) if (x > 5) y = 1; else y = 0;
33      System.out.println("After y = 6; if (x > 3) if (x > 5) y = 1; else y = 0; the value of y is " + y);
34
35      y = 6;
36      if (x > 3) {
37        if (x > 5) y = 1;
38      } else
39        y = 0;
40      System.out.println("After y = 6; if (x > 3) { if (x > 5) y = 1; } else y = 0; the value of y is " + y);
41
42      y = 6;
43      if (x > 3) {
44        if (x > 5)
45          y = 1;
46        else
47          y = 0;
48      }
49      System.out.println("After y = 6; if (x > 3) { if (x > 5) y = 1; else y = 0; } the value of y is " + y);
50    }
51  }
```

Glossary

abstraction the hiding of unnecessary detail, page 7

actions , page 95

Algebraic Data Types , page 8

assignment model , page 17

attribute grammar , page 12

Attribute Grammar (AG) , page 95

Attribute-Action Grammar (AAG) , page 95

attributes , page 95

back end , page 2

bindings , page 17

compiler , page 2

Context Free Grammar , page 8

control flow , page 16

derivation , page 2

disambiguation rule , page 10

Executable semantics , page 22

external syntax ., page 2

formal semantics , page 6

front end , page 2

grammar , page 2

internal syntax , page 2

interpreter , page 2