

Software Language Engineering with ART

Adrian Johnstone

`a.johnstone@rhul.ac.uk`

October 29, 2024

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

Contents

1	Introduction	1
1.1	Classical compiler phases	1
1.2	Classical front end techniques	3
1.3	Classical approaches to describing languages	3
1.3.1	Informal and formal semantics	5
1.3.2	Specification styles for syntax	6
1.4	Early compilers and modern challenges	7
1.5	Our approach	7
1.6	A running example: the miniGCD language	8
1.7	The fixed-code-and-program-counter interpretation	9
1.8	The problem with assignment	10
1.9	The problem with the program counter	11
1.10	The reduction interpretation	11
1.11	The problem with loops	12
1.12	A reduction evaluation of GCD in MiniGCD	13
1.13	Executable semantics and automation	14
2	Rewriting	17
2.1	Equality of programs	17
2.2	Mathematical objects, their denotations and software implementations	18
2.3	String rewriting	19

2.4	Term rewriting	19
2.5	Internal syntax style	21
2.6	Terms	22
2.6.1	Denoting term symbols	22
2.7	The Value system and plugins	23
3	Context Free Grammars	25
3.1	A languages is a set of strings	25
3.2	A Context Free Grammar generates a language	25
3.3	A derivation records rule applications	25
3.4	A parser constructs derivations	25
3.5	GIFT operators rewrite derivations	25
3.6	Paraterminals specify the lexer-parser interface	25
3.7	Choosers reduce ambiguity	25
4	Reduction semantics	27
4.1	An eSOS specification for MiniGCD	27
5	Attribute interpreters	29
5.1	An attribute-action specification for MiniGCD	29
5.2	Exercises	29
6	Advice on language design	31
A	ART reference documentation	33
A.1	Lexical builtins	33
B	The Royal Holloway course	35
B.1	Aims and motivation	36
B.2	Learning outcomes	36
B.3	Assessment	37

<i>SLE with ART - October 29, 2024</i>	iii
B.4 Protocol for asking questions	38
B.5 Teaching week by week	38
C Music making with Java MIDI	41
D Image processing operations	43
E 3D modelling	45

List of Figures

1.1	Reduction trace for the GCD algorithm with inputs 6,9	15
4.1	An eSOS specification for MiniGCD	28
5.1	An attribute-action specification for MiniGCD	30

List of Tables

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.

Our goal is the construction of *maintainable* SLE tools by writing concise specifications which are amenable to automated checking, and which allow the actual tools to be automatically generated. 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 thus are fragile in the face of attempts to extend or modify them. High level specifications in widely understood notations would make it much easier to extend and modify language specifications, so that other engineers could both maintain and reuse our work into the future.

1.1 Classical compiler phases

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.

The source text is a string of characters. In the Lex phase, this string is parti-

lexemes

tioned into a sequence of substrings called *lexemes*, and the resulting string 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 string.

derivation

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 language, and (b) to build a *derivation* which shows how the lexeme string can be constructed from the language rules, and which shows the nesting relationships in a language. Informally these rules governing the valid ordering of lexemes are called the *syntax* of the language. Programming language syntax is usually specified by rewrite rules which together form a *grammar*. Together, the Lex and Parse phases are often referred to as the *front end*.

front end

symbol table

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.

internal syntax

external syntax

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, we avoid the term AST and instead in this work we refer to internal representations as *internal syntax* as opposed to the textual form of the program which we call the *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 than being recomputed on every loop iteration. This phase is traditionally called an *optimiser* but we avoid that term because in general the resulting code is *not* optimal: indeed different kinds of rework can cancel each other out and in extreme cases actually reduce performance. To be correct, the rework must not change the meaning of a program; it can be very hard to ensure that rework is safe.

compiler

interpreter

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. In an *interpreter* the **Effect** 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.

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.

1.2 Classical front end techniques

The golden age of research into front end techniques ran from roughly 1960 to 1975: Donald Knuth says that ‘*compiler research was certainly intensive, representing roughly one third of all computer science in the 1960s*’ [DEK14, 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.*’ [DEK14, p.42].

This work coalesced around two 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.

1.3 Classical approaches to describing languages

The convergence of theoretical analysis and engineering practice in the **Lex** and **Parse** classical 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 specify the actions of a programming language in the **Rework** and **Effect** phases is a continuing concern.

Most widely used programming have a ‘standard’: a document that describes the effects that should be induced by the phrases of the programming language. How do programming language standards documents specify languages?

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 meanings are the same.

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 device.

state

The written form of a programming language fragment 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

1.3.1 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

formal semantics

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*.

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!

1.3.2 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 rules called a context free rule and that is evident in the extracts above; they all give a context free 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`.

abstraction

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.

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 starts 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.

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

Context Free Grammar

1.4 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. This 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 all identifiers to be declared before use.

Syntax Directed Translation

1.5 Our approach

We have seen that programming language standards documents use context free grammars to specify syntax, and legalistic English to specify semantics.

The same facility can be used to implement a simple `Analyse` phase, yielding a symbol table and some form of internal representation.

The details of the `Analyse`, `Rework` and `Effect` phases vary widely between systems, and are often poorly documented. As a result, it is hard to answer questions concerning the completeness, consistency and correctness of real compilers, and 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 associated translators naturally increases because the users are effectively testers. The corollary is that language processors with few users should perhaps be treated with caution.

**** Java/C front ends**

The techniques we shall use are unashamedly resource hungry: in particular they demand a lot of memory.

The ART tool that we are using here to study software language engineering contains an interpreter for one style of formal semantics called *Structural*

Structural Operational Semantics **Operational Semantics (SOS).**

eSOS

Our particular version is called *eSOS* because it allows certain parts of a specification to be elided away, which saves writing. eSOS is designed to appeal to those with a background in procedural languages: in Chapter 4 we give programs that show how eSOS rules are interpreted so that software engineers can reason about the way in which their specifications will be processed.

Now, some aspects of our presentation would induce a sharp intake of breath from those who prefer a more declarative and mathematically rigorous approach to semantics, but we hope that eSOS will de-mystify SOS for programmers who are wary of mathematical treatments, and perhaps encourage the use of SOS as a design and prototyping medium for domain specific languages where sometimes the only available language definition is the code of an implementation.

semi-formal semantics

Our kind of SOS is perhaps best thought of as a *semi-formal semantics*

attribute-action

ART also supports *attribute-action* systems: a style of semantics that is common in traditional parser generators such as the Bison tool which is distributed with most Unix systems; Chapter 5 describes the approach in detail. These kinds of specifications can usually be executed more efficiently than an eSOS interpretation, but in our experience attribute-action systems are more likely to be incomplete or inconsistent (i.e. buggy) than an eSOS specification. In Chapter 6 we discuss our preferred approach, which is to prototype in eSOS and then, if necessary for performance reasons, to transform to an equivalent but more efficient attribute-action.

1.6 A running example: the miniGCD language

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.

```

1 public class GCD {
2     public static void main(String[] args) {
3         int a = 6;
4         int b = 9;
5
6         while (a != b) {
7             if (a > b)
8                 a = a - b;
9             else
10                b = b - a;
11        }

```

```

12 |   int gcd = a;
13 | }
14 | }

```

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.

```

1 | a := 6;
2 | b := 9;
3 |
4 | while a != b {
5 |     if a > b
6 |         a := a - b;
7 |     else
8 |         b := b - a;
9 | }
10 | gcd := a;

```

We shall call this notation the MiniGCD language. Note that assignment to a variable is denoted by `:=`, not by `=` as it 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).

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

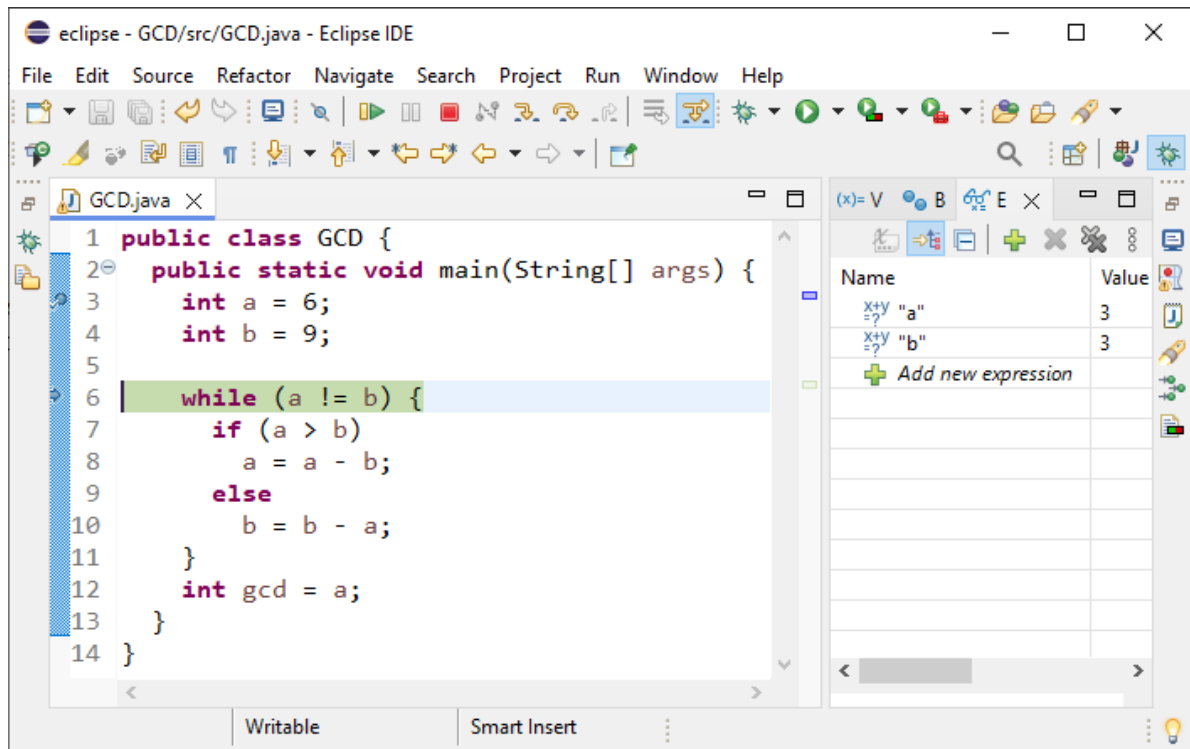
In almost all modern computing devices most programs are lists of instructions that reside in the memory or *store*. The instructions for a particular program do not change as it is being executed. A special register called the *Program Counter* (PC) which points to the next piece of code to be executed, 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 value depending on that outcome; that causes the processor to start execution at some new location.

store

Program Counter

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.

control flow



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.

1.8 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).

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.

bindings

****mutable map vs making a new map**

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.

1.9 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 manageable 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.

1.10 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.

1.11 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.

1.12 A reduction evaluation of GCD in MiniGCD

Figure 1.1 on page 15 presents a reduction semantics trace for the GCD algorithm written in MiniGCD program shown on page 9.

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 9, 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 initialisations 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 following 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.

1.13 Executable semantics and automation

Manually constructing the execution trace shown in Figure 1.1 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 semantics* to distinguish it from an efficient production-quality compiler.

Executable semantics

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 6.

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 ↦ 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 ↦ 6, b ↦ 9}
Evaluate a ≠ b with store {a ↦ 6, b ↦ 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 ↦ 6, b ↦ 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 ↦ 6, b ↦ 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 ↦ 6, b ↦ 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 ↦ 6, b ↦ 9}
Evaluate a > b with store {a ↦ 6, b ↦ 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 ↦ 6, b ↦ 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 ↦ 6, b ↦ 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 ↦ 6, b ↦ 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 ↦ 6, b ↦ 9}
Evaluate b - a with store {a ↦ 6, b ↦ 9}
b:=b-a; while a!=b if a>b a:=a-b; else b:=b-a; gcd=a; {a ↦ 6, b ↦ 9}
b:=b-6; while a!=b if a>b a:=a-b; else b:=b-a; gcd=a; {a ↦ 6, b ↦ 9}
b:=9-6; while a!=b if a>b a:=a-b; else b:=b-a; gcd=a; {a ↦ 6, b ↦ 9}
b:=3; while a!=b if a>b a:=a-b; else b:=b-a; gcd=a; {a ↦ 6, b ↦ 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 ↦ 6, b ↦ 3}
Evaluate a ≠ b with store {a ↦ 6, b ↦ 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 ↦ 6, b ↦ 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 ↦ 6, b ↦ 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 ↦ 6, b ↦ 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 ↦ 6, b ↦ 3}
Evaluate a > b with store {a ↦ 6, b ↦ 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 ↦ 6, b ↦ 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 ↦ 6, b ↦ 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 ↦ 6, b ↦ 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 ↦ 6, b ↦ 3}
Evaluate a - b with store {a ↦ 6, b ↦ 3}
a:=a-b; while a!=b if a>b a:=a-b; else b:=b-a; gcd=a; {a ↦ 6, b ↦ 3}
a:=a-3; while a!=b if a>b a:=a-b; else b:=b-a; gcd=a; {a ↦ 6, b ↦ 3}
a:=6-3; while a!=b if a>b a:=a-b; else b:=b-a; gcd=a; {a ↦ 6, b ↦ 3}
a:=3; while a!=b if a>b a:=a-b; else b:=b-a; gcd=a; {a ↦ 6, b ↦ 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 ↦ 3, b ↦ 3}
Evaluate a ≠ b with store {a ↦ 3, b ↦ 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 ↦ 3, b ↦ 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 ↦ 3, b ↦ 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 ↦ 3, b ↦ 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 ↦ 3, b ↦ 3}
Assign result
gcd=a; {a ↦ 3, b ↦ 3}
, {a ↦ 3, b ↦ 3 gcd ↦ 3}

```

Figure 1.1 Reduction trace for the GCD algorithm with inputs 6,9

Chapter 2

Rewriting

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

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

An *equation* is two expressions separated by the *equality symbol* $=$. At a fundamental level, this tells us that the two expressions either side are interchangeable 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

$$a \wedge b \dots$$

Now we know a few facts about Boolean algebra.

2.1 Equality of programs

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

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

generates the same output as

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

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

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

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

Code improvement

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

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

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

2.2 Mathematical objects, their denotations and software implementations

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

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

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

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

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

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

2.3 String rewriting

2.4 Term rewriting

Programs often contain *expressions* such as

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

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

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

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

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

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

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

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

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

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

All three of these forms are formally equivalent in that we can unambiguously convert between them 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 ??.

2.5 Internal syntax style

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

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

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

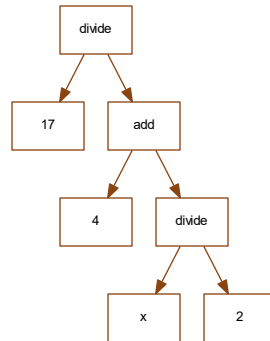
might be written

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

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

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

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



2.6 Terms

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

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

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

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

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

2.6.1 Denoting term symbols

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

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

2.7 The Value system and plugins

Chapter 3

Context Free Grammars

- 3.1 A language is a set of strings**
- 3.2 A Context Free Grammar generates a language**
- 3.3 A derivation records rule applications**
- 3.4 A parser constructs derivations**
- 3.5 GIFT operators rewrite derivations**
- 3.6 Paraterminals specify the lexer-parser interface**
- 3.7 Choosers reduce ambiguity**

Chapter 4

Reduction semantics

4.1 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 4.1 An eSOS specification for MiniGCD

Chapter 5

Attribute interpreters

5.1 An attribute-action specification for MiniGCD

5.2 Exercises

```

!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 5.1 An attribute-action specification for MiniGCD

Chapter 6

Advice on language design

Appendix A

ART reference documentation

A.1 Lexical builtins

&CHAR_BQ	`C
&ID	AlphanumericIdentifier
&INTEGER	123
&REAL	12.3
&STRING_BRACE	{ A string delimited by braces }
&STRING_BRACE_NEST	{ A string { with nested instances } delimited by braces }
&STRING_DOLLAR	\$ A string delimited by dollar signs \$
&STRING_DQ	" A string delimited by double quotes "
&STRING_PLAIN_SQ	' A string delimited by single quotes with no escapes '
&STRING_SQ	' A string delimited by single quotes '

These builtins can only appear as an argument to the whitespace directive

```
&SIMPLE_WHITESPACE
&COMMENT_BLOCK_C      /* a C style block comment */
&COMMENT_LINE_C // a C style line comment
&COMMENT_NEST_ART (* An ART style comment (* which nests *) *)
```


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 6.

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 and should probably skip straight to Chapter 1.

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

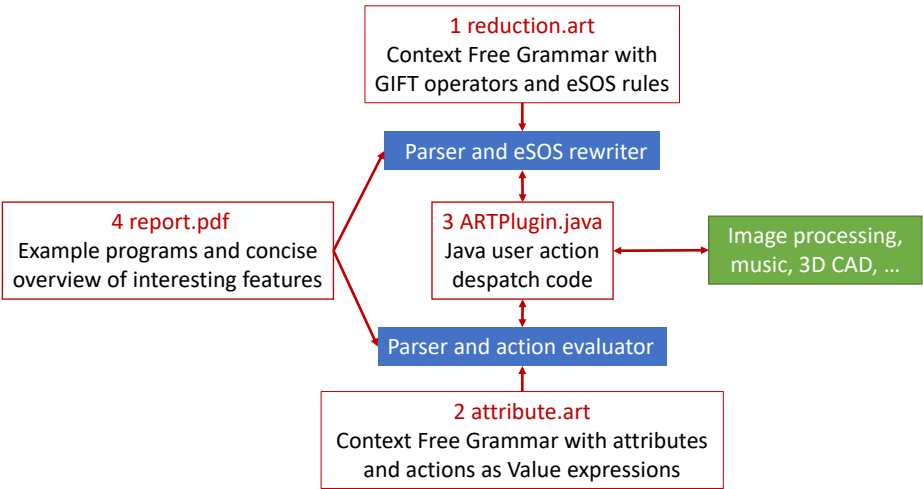
1. music generation *via* the Java MIDI interface;
2. image processing using the two-dimensional (2D) features of Java FX; and
3. Computer Aided Design for 3D printing using an extended form of the three-dimensional (3D) features of Java FX.

You must commit to one of these topics by the end of the week 2 lab. To help you decide, please read the background material on each of these in Appendices [C](#), [D](#) and [E](#).

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
4. **report.md** - a concise report highlighting interesting and novel features, along with example programs that work with both the reduction and attribute interpreters.

Here is a diagram showing how these 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`. You demonstrate the utility of your language using example programs listed in `report.pdf`

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.4 Protocol for asking questions

When asking for help, I 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 these attachments `artLog.txt`, `reduction.art`, `attribute.art` and `ARTPlugin.java`. Do not send screenshots; only text file attachments of these standard files are acceptable.

B.5 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 topic	Laboratory session
1	1	Two interpreters for GCD	Alero 3D design
2	3	Terms and pattern matching	termTool
3	4	Structural Operational Semantics	eSOS for expressions
4	3	The ART Value system	eSOS for control flow
5	2	Context Free Grammars and choosers	OSBRD parsing

6	2	GIFT operators	Generation of eSOS terms
7	5	Attribute Grammars and Attribute Theories	Eliminating tags
8	2	Paraterminals, lexicalisation and lexical choice	custom token patterns

Appendix C

Music making with Java MIDI

Appendix D

Image processing operations

Appendix E

3D modelling

Bibliography

- [DEK14] Edgar G. Daylight Donald. E. Knuth. *Algorithmic Barriers Falling*. Lonely Scholar, 2014.