Software language engineering with ART

Adrian Johnstone
a.johnstone@rhul.ac.uk

June 12, 2024

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

Contents

List of Figures

List of Tables

Chapter 0

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.

0.1 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 formal 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 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 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.

0.2 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 eSOS interpreter uses to execute programs;

4. be able to write *attribute-action rules* that the attribute evaluator uses to execute programs;

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*.

0.3 Assessment

Your command of these learning outcomes will be assessed *via* a substantial personal project and an invigilated examination, each component being worth 50% of the final mark.

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

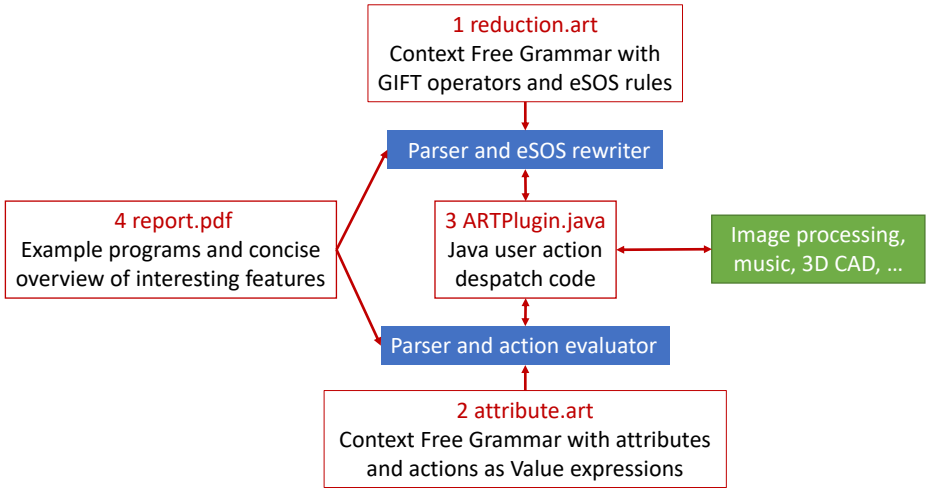
1. music generation *via* the Java MIDI interface;
2. image processing using the two-dimensional (2D) features of Java FX; and
3. Computer Aided Design for 3D printing using an extended form of the three-dimensional (3D) features of Java FX.

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 A, B and C.

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 interpeters to your chosen domain specific actions
4. `report.pdf` - a concise report highlighting interesting and novel features, along with example programs that work with both interpreters.

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



The blue boxes above represent language execution 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.

0.4 Protocol for asking questions

There is only one rule: if you ask for help then you must send me one email containing everything that I need to reproduce your observations on my machine.

Depending on which stage you are at, this will be:

0. A text cut-and-paste of the ALL of the output that you are seeing, including the command line that you are using to initiate the run.

Never send an screenshot image because I cannot easily get the text back out of that.

1. Your informal specification

2. Your rules file, often called something like esos_MyLanguage.art

3. Your complete plugin source code, that is ValueUserPlugin.java

4. Your 'hatted' external-to-internal syntax grammar

5. Your attribute-action grammar

6. !try directives and example programs that exercise your deliverables and show the problem

These deliverables should be in text files that are attached to your email.

Please don't mess with my script files: a few students couldn't get things to compile and that turned out to be because they've changed some of my setup without realising the implications. I strongly advise that you work in SLELabs/ProjectWork and use my standard scripts.

So to summarise: if you need help then send me an email outlining your problem which has text attachments showing the log of your run and the ART deliverables. I'll then try to reproduce on my machine and give you advice.

0.5 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	eSOS	eSOS for expressions
4	3	The ART Value system	eSOS for control flow
5	2	Context Free Grammars	OSBRD parsing
6	2	GIFT operators	Generation of eSOS terms
7	5	Attribute-action interpreters	mini expression languages
8	5	Delayed attributes	mini control flow languages
9	2	Paraterminals and lexicalisation	mini with custom token patterns
10	2	Ambiguity management	lexical and phrase level choice

Chapter 1

First examples

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:

- ◇ 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.
- ◇ 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.

state

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.

We call the written form of a programming language fragment its *syntax*. *syntax*

We call the meaning of a fragment its *semantics* (that is the effect on the state of the computer). *semantics*

1.1 Specification styles for 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 not ideal 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 ART tool that we are using here to study software language engineering contains an interpreter for one style of formal semantics called *Structural Operational Semantics* (SOS). *Structural Operational Semantics*

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. *eSOS*

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

semi-formal semantics

attribute-action

ART also suports *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.

1.2 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 *Context Free Grammar* (CFG), and that is evident in the extracts above; they all give a context free grammar 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`.

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 CFG rule, then, is to give a template for one feature of the language: the rule tells is that a conditional statement starts with the keyword `if` which must be followed by an expression over the booleans, 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. A complete set of such rules with no missing definitions is called a Context Free Grammar. ART contains tools for processing and visualising grammars in various ways that we shall discuss in Chapter 2.

1.3 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 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. MiniGCD only contains the features used here: it does not even provide addition (though we will extend it later).

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

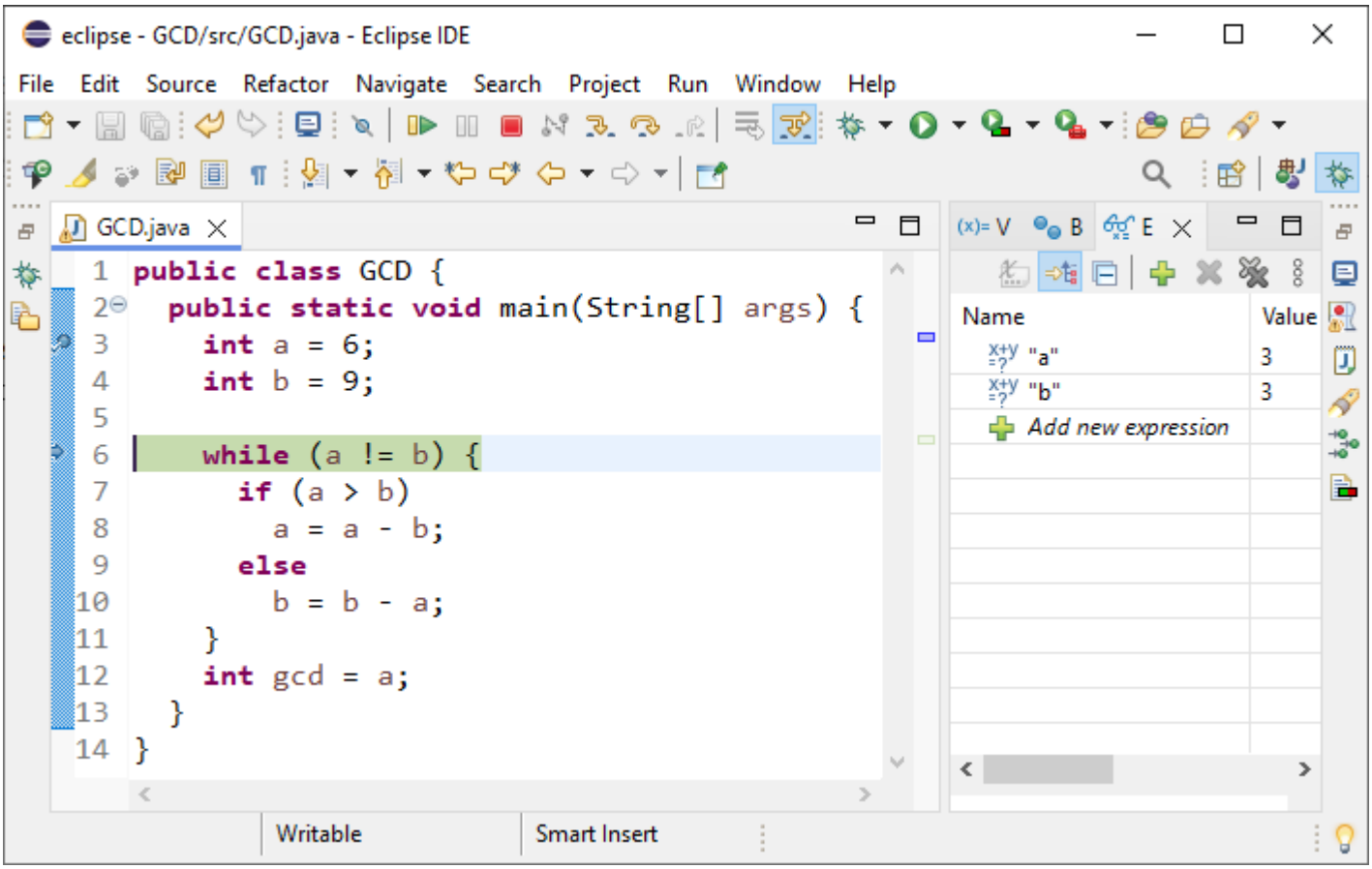
store

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.

1.5 The problem with assignment

substitution model

assignment model

The *substitution model* for variables that is used in most mathematical reasoning 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 history 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.

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

1.6 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.

1.7 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 be done', 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.

1.8 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 have taken here is to make use of a *program identity*, that is a program transformation that does not change the semantics of a program term, but does change the syntax, and thus the reduction trace. If we have a loop of the form

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.

1.9 A reduction evaluation of GCD in MiniGCD

Figure 1.1 on page 11 presents the reduction semantics trace for the GCD algorithm written in MiniGCD program shown on page 5.

There are a large number of steps in this trace, which make for intimidating reading, but bear in mind that a step (very roughly) corresponds to a machine operation such as fetching an operand or adding two numbers. Useful programs entail the execution of a *lot* of operations: some of the programs we run on modern processors take an appreciable amount of time to execute even

though a 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 shall write the entire program term on a single line: rather than the nicely laid out version shown on page 5, 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, 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 tuple for 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, and the raising of exceptions.

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.

1.10 An eSOS specification for MiniGCD

Constructing the execution trace shown in Figure 1.1 would be very time consuming, although it does have the benefit of showing very clearly and concisely

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

```
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 1.2 An eSOS specification for MiniGCD

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

1.11 An attribute-action specification for MiniGCD

1.12 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 1.3 An attribute-action specification for MiniGCD

Chapter 2

Context Free Grammars

2.1 A languages is a set of strings

2.2 A Context Free Grammar generates a language

2.3 A derivation records rule applications

2.4 A parser constructs derivations

2.5 GIFT operators rewrite derivations

2.6 Paraterminals specify the lexer-parser interface

2.7 Choosers reduce ambiguity

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++;}
```

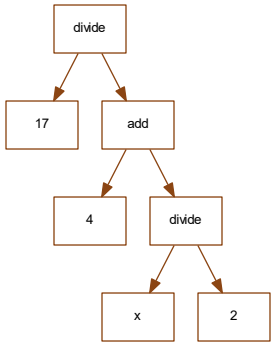
might be written

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

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

3.6.2 Typed terms

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

3.7 Terms and their implementation in Java

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

We can move from

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

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

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

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

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

Arrays should just be a vector of children

3.8 The Value system and plugins

Chapter 4

Reduction semantics with eSOS

Chapter 5

Attribute action interpreters

Chapter 6

Advice on language design

Appendix A

Music making with Java MIDI

Appendix B

Image processing operations

Appendix C

3D modelling

Appendix D

ART reference documentation

D.1 Relation symbols

```
-ss   --- x -> y
-ssKleene   --- x ->* y
-ssRepeated   --- x ->> y
-bs   --- x => y
-bsKleene   --- x =>* y
-bsRepeated   --- x =>> y
-user1   --- x -\ y
-user1Kleene   --- x -\* y
-user1Repeated   --- x -\> y
-user2   --- x -/ y
-user2Kleene   --- x -/* y
-user2Repeated   --- x -/> y
-anywhereRewrite   --- x ~> y
-anywhereRewriteKleene   --- x ~>* y
-anywhereRewriteRepeated   --- x ~>> y
```

D.2 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 delimted 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 arument 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 *) *)
```

Glossary

abstraction the hiding of unnecessary detail, page 4

assignment model , page 7

attribute-action , page 4

bindings , page 7

control flow , page 6

eSOS , page 3

formal semantics , page 3

normal forms ., page 10

Program Counter , page 6

redex , page 10

reduction step , page 9

reduction trace , page 9

semantics the meaning of a language, page 3

semi-formal semantics , page 4

state the set of values maintained by a program, page 2

store , page 6

Structural Operational Semantics , page 3

substitution model , page 7

syntax the written form of a language, page 3