ART Tutorial

Software Language Engineering with ART

Adrian Johnstone

a.johnstone@rhul.ac.uk

August 22, 2025

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

Contents

Chapter 1

Introduction

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

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

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

In practice we mostly write software in a *hopeful* way, and then use testing to try and find the gaps in our understanding. Unfortunately, programming languages are inherently difficult to test since they are designed to be flexible notations with very many combinations of interacting features and in any case, as Edsger Dikstra famously noted, *Program testing can be used to show the presence of bugs, but never to show their absence.* [DDH72, p6].

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

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

1.1 The classical approach to translation

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

lexemes

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

derivation

grammar

front end

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

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, in our work we avoid the term AST and instead refer to internal representations as *internal syntax* as opposed to the original form of the program which we call the *external syntax*

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

compiler

interpreter

back end

middle end

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

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

Classical front end techniques

The years 1960–75 represent a golden age of research into front end techniques: Donald Knuth noted that *'compiler research was certainly intensive, representing roughly one third of all computer science in the 1960s'*[KD14, 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.'*[KD14, p.42].

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

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

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

Classical approaches to describing languages

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

How are programming languages defined in current practice? Most widely used programming languages have a 'standard': a document that describes the effects that should be induced by the phrases of the programming language. For instance, here is an extract from the Pascal report

> **9.2.2.1. If statements.**
>
> The if statement specifies that the statement following the symbol then be executed only if the Boolean expression yields true.
>
> IfStatement = "if" BooleanExpression "then" Statement

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

> **14.9. The if Statement**
>
> The if statement allows conditional execution of a statement.
>
> *IfThenStatement:*
> if (*Expression*) *Statement*
>
> The Expression must have type boolean or Boolean, or a compile-time error occurs.
>
> **14.9.1. The if-then Statement**
>
> An if-then statement is executed by first evaluating the Expression.
>
> If evaluation of the Expression completes abruptly for some reason, the if-then statement completes abruptly for the same reason.
>
> Otherwise, execution continues by making a choice based on the resulting value:
>
> (a) If the value is true, then the contained Statement is executed; the if-then statement completes normally if and only if execution of the Statement completes normally.
>
> (b) If the value is false, no further action is taken and the if-then statement completes normally.

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

> **6.8.4 Selection statements**
>
> Syntax
>
> *selection-statement:*
> if (*expression*) *statement*
>
> Semantics
>
> A selection statement selects among a set of statements depending on the value of a controlling expression.
>
> A selection statement is a block whose scope is a strict subset of the scope of its enclosing block. Each associated substatement is also a block whose scope is a strict subset of the scope of the selection statement.
>
> **6.8.4.1 The if statement**
>
> Constraints
>
> The controlling expression of an if statement shall have scalar type.
>
> Semantics
>
> The substatement is executed if the expression compares unequal to 0.

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

If we write in C or Java the program fragment

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

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

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

From these examples we can deduce the following.

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

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

state

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

syntax

semantics

Informal and formal semantics

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

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

formal semantics

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

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

1.2 Early compilers and modern challenges

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

single-pass compiler

In multi-pass compilers the details of the Analyse, Rework and Effect phases vary widely between systems, and are often poorly documented, so it can be hard to establish the completeness, consistency and correctness of real compilers. Compiler errors are not rare, at least in the early stages of a language's development: as the user base for a language grows, confidence in the associated translators naturally increases because the users are effectively testers. Language processors with few users should perhaps be treated with caution.

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

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

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

1.3 Specification styles for syntax

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

For Pascal we have

IfStatement = “if” BooleanExpression “then” Statement

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

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

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

Context Free Grammar

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

1.4 Specification styles for semantics

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

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

It seems that non-trivial languages always have 'dark corners' where the interaction of useful features can cause surprising effects. Java's design benefited greatly from previous generations of programming languages, but the *Java Puzzlers* book [BG05] show a wide variety of small programs with surprising effects – and that book was published in 2005 at the time of Java 5. The language has been significantly extended since then and further quirks will have resulted. Of course, these are confusions that arise at the level of the *user* of a language: the understanding of the language by *implementers* must (if their implementations are to be correct) subsume all of these details and more. A specification written in English is unlikely to answer all implementers' questions.

Algebraic Data Types

The most significant attempt to define a general purpose language using formal rules (rather then English-language descriptions) is Standard ML. SML began as a scripting language for a theorem prover. A key design goal was to provide an external syntax that was comfortable for mathematicians via the use of type inference which (almost) eliminates the need for type declarations and pattern matching on *Algebraic Data Types* to directly support the kinds of case analysis that naturally arise when writing down proofs. The formal definition of the language was published in 1997 [MHMT97] and *in principle* allows automatic construction of an interpreter from the rules given in the book. We are going to use a related specification style, coupled to new algorithms which remove the weaknesses of classical front end tools, and intepreters which can directly execute formal semantic specifications.

1.5 Ambiguity

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

Ambiguity is a fertile source of jokes:

> *How do you make a sausage roll? Release the sausage at the top of a ramp.*
>
> *How do you make a professor fast? Take their lunch away.*

And so on.

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

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

1. In C and Java, does the entirely valid phrase $z = x---y$ mean the same as $z = (x--) - y$ or $z = x - (--y)$ or $z = x - (-(-y))$?

2. In everyday arithmetic (never mind programming languages) does $5-4-3$ evaluate to -2 or to 4, that is, should we interpret the expression as $((5-4)-3)$ or $(5-(4-3))$?

3. In this Java program fragment `y = 6; if (x > 3) if (x > 5) y = 1; else y = 0;` what should the final value of `y` be when `x` is 4? If we think that the fragment has the same meaning as

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

```
    y = 0;
```

then the the final value of `y` is 0.

If we use this interpretation

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

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

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

disambiguation rule

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

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

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

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

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

Consider the Java declaration `Set<Set<Integer>> setOfSets;` The intrinsic arguments are bracketed using `<...>` and so the nesting finishes with `>>`. But those two characters together also represent the right-shift operator in Java, so how does the lex phase know whether to partition `>>` as two closing bracket lexemes `>`, `>` or as a single operator `>>`? Once we have a phrase level analysis from the parser we can resolve this ambiguity, but in the classical pipeline the lexer runs first, and classical lexers can only return a single sequence of lexemes. Users of classical tools have to adopt a variety of complex mechanisms to overcome these difficulties, and that makes the resulting translators hard to understand and hard to completeness, consistency and correctness. We shall show how to use *general lexer* and a *general multiparser* which allows all

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

1.6 Our approach – Ambiguity Retained Translation

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

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

The ART front end In ART, the classical LR and LL style deterministic parsers are replaced by a *multiparser* and a *multilexer*. As the name suggests, multiparsers are capable of returning multiple derivations (interpretations) of a string, and in fact our MGLL algorithm will return *all* derivations, even when there are infinitely many of them [SJW23]. MGLL achieves this in worst case cubic time and cubic space; in practice the worst case bound is elusive and the algorithms only require linear time for typical current programming language phrases, degrading gracefully towards the cubic bound when processing difficult parts of the grammar.

multiparser

multilexer

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

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

In ART we do something much simpler: take the derivation tree that emerges from the front end and progressively transform it under the control of *term rewriting* rules until the program has all been rewritten away, accumulating the program's side effects as we go. The particular form of rewrite rules that we use is called a *Structural Operational Semantics* (*SOS*); this approach was introduced by Gordon Plotkin in 1980 [Plo04].

term rewriting

SOS

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

attribute grammar

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

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

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

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

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

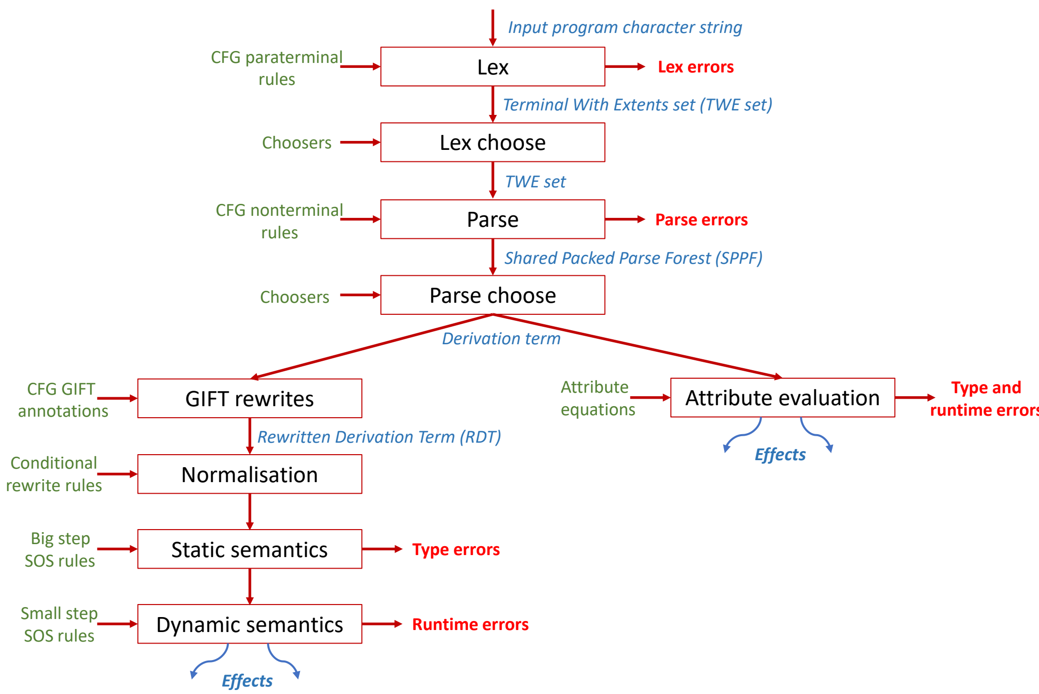


Figure 1.1 The ART pipeline

Chapter 2

Models of program execution

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

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

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

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

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

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

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

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

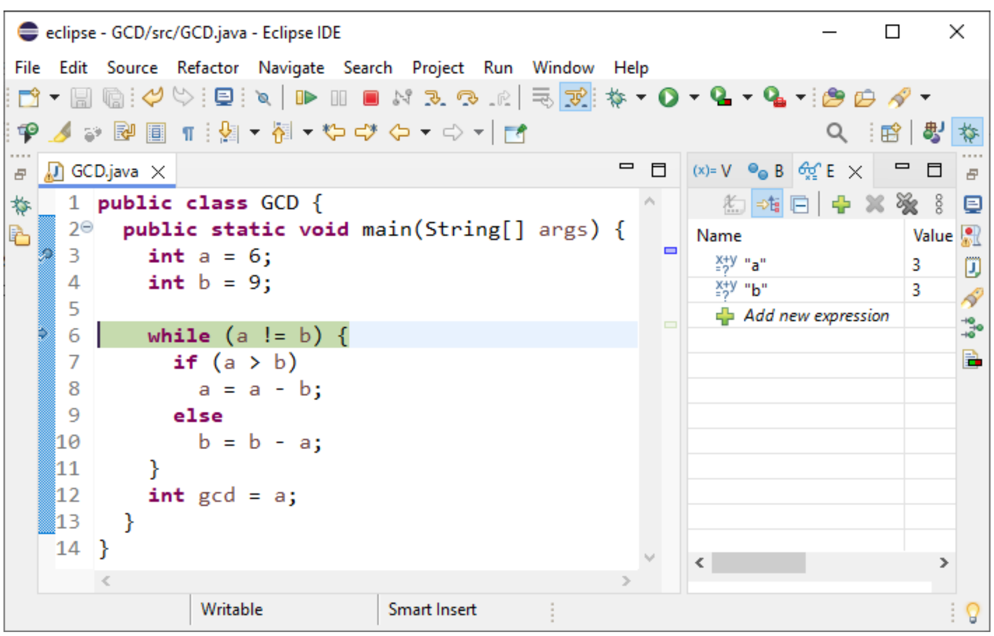
store

Program Counter

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

control flow

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



We can see the program's Java code on the left and the state of the store with variables `a` and `b`, both presently mapped to the value 3. The program counter value is represented by the small arrow in the margin at line 6.

2.2 The problem with assignment

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, x and 3 will be 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).

substitution model

assignment model

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

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

2.3 The problem with the program counter

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

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

2.4 The reduction interpretation

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

Consider this program

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

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

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

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

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

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

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

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

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

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

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

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

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

reduction trace

reduction step

We call this kind of display of machine states the *reduction trace* for our program, and each line represents a *reduction step*–so called because usually the program fragment reduces in size at each step (though not always, as we shall see in the next section). The steps match up rather well with the individual machine level instructions that would be executed by a real computer, and at every point we have a complete record of the state of the machine as well as being able to see what else we have to do.

2.5 The problem with loops

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

while booleanExpression **do** statement;

then we can *always* transform it into

if booleanExpression { statement; **while** booleanExpression **do** statement; }

We have effectively unpacked the first iteration of the loop and are handling it directly with an **if** statement followed by a new copy of the **while** loop which will compute any further iterations. When we have completed all of the iterations we shall encounter a term like

if false { statement; **while** booleanExpression **do** statement; }

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

2.6 A reduction evaluation of GCD in MiniGCD

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

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

We shall use a slightly more compact form to display the steps. First, we write the entire program term on a single line: rather than the nicely laid out version shown on page 16, we say a=6; b=9; **while** a!=b **if** a>b a:=a−b; **else** b:=b−a; gcd=a;

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

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

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

redex

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

semantic entities

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

normal forms

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

2.7 Executable semantics and automation

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

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

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

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

Executable semantics

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

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

2.8 Specifying reduction semantics: a preview

As a preview — all of which will make sense by the time you have read all of the subsequent chapter — let us now look at an executable formal *reduction* semantics for the GCD language. It come in three parts (i) a context free grammar that specifies the external, human friendly, syntax of the GCD language; (ii) a set of reduction rules and (iii) some test inputs.

Here are the context free rules, written in ART's notation. We shall explain these in detail in Chapter 5.

```
seq ::= statement^^ | statement seq
statement ::= assign^^ | while^^ | if^^
assign ::= &ID ':='^ expression ';'^
while ::= 'while'^ expression 'do'^ statement
if ::= 'if'^ expression statement | 'if'^ expression 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
```

Here are the reduction rules, which are explained in Chapter 4.

```
!configuration ->, _sig

-sequenceDone --- seq(__done, _C), _sig -> _C, _sig
-sequence _C1, _sig -> _C1P, _sigP --- seq(_C1, _C2), _sig -> seq(_C1P, _C2), _sigP

-ifTrue --- if(true, _C1, _C2),_sig -> _C1, _sig
-ifFalse --- if(false, _C1, _C2),_sig -> _C2,_sig
-ifResolve _E, _sig ->_EP, _sigP --- if(_E,_C1,_C2),_sig -> if(_EP, _C1, _C2), _sigP
```

```

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

-assign _n |> __int32(_) --- assign(_X, _n), _sig -> __done, __put(_sig, _X, _n)
-assignR _E, _sig -> _I, _sigP --- assign(_X, _E), _sig -> assign(_X, _I), _sigP

-gt _n1 |> __int32(_) _n2 |> __int32(_) --- gt(_n1, _n2),_sig -> __gt(_n1, _n2),_sig
-gtR _n |> __int32(_) _E2, _sig -> _I2,_sigP --- gt(_n, _E2),_sig -> gt(_n, _I2), _sigP
-gtL _E1, _sig -> _I1, _sigP --- gt(_E1, _E2),_sig -> gt(_I1, _E2), _sigP

-ne _n1 |> __int32(_) _n2 |> __int32(_) --- ne(_n1, _n2),_sig -> __ne(_n1, _n2),_sig
-neR _n |> __int32(_) _E2, _sig -> _I2,_sigP --- ne(_n, _E2) ,_sig -> ne(_n, _I2), _sigP
-ne _E1, _sig -> _I1, _sigP --- ne(_E1, _E2),_sig -> ne(_I1, _E2), _sigP

-sub _n1 |> __int32(_) _n2 |> __int32(_) --- sub(_n1, _n2),_sig -> __sub(_n1, _n2),_sig
-subR _n |> __int32(_) _E2, _sig -> _I2,_sigP --- sub(_n, _E2),_sig -> sub(_n, _I2), _sigP
-subL _E1,_sig -> _I1,_sigP --- sub(_E1, _E2),_sig -> sub(_I1, _E2), _sigP

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

Finally, here are the tests: they start with the directive **!try** which is followed by the whole program to be executed, written as a string. (You can also give the name of a file for large input programs).

It turns out that the context free rules act so as to convert this string form of the program into a *term* which is a tree-like structure. It is possible,and actually recommended, to write the reduction rules before deciding on the external human-friendly syntax, and so ART allows you to execute a term directly, bypassing the parsing stage. The second **!try** here contains the term that the parser generates when running on the string in the first try, so these two generate the same execution trace.

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

!try seq(assign(a, 6), seq(assign(b, 9), while(ne(deref(a), deref(b)), if(gt(deref(a), deref(b)),
          assign(a, sub(deref(a), deref(b))), assign(b, sub(deref(b), deref(a))))))), __map
```

2.9 Specifying attribute-action semantics: a preview

A reduction semantics is ideal for prototyping: it separates out the syntax rules from the reduction rules, and the reduction rules are amenable to further formal analysis. However, there is a cost: the repeated rewriting of the tree is expensive, both in terms of space and computation time. There is an alternative approach, variants of which are widely used in production translators. The idea is to take the tree that the parser produces and associate data fields called attributes with each node, and also to associate actions with each tree node which are allowed to look at nearby attributes, and compute new values. We then walk the tree executing actions without changing the tree itself. We call these kinds of specifications action-attribute systems.

One form of an attribute-action system embeds into the Context Free Grammar rules actions written as Java fragments. Here is such a specification: the actions are delimited by !! ... !! pairs. We shall examine this approach in Chapter 6.1.

```
!prelude !! import java.util.Map; import java.util.HashMap; !!
!support !! Map<String, Integer> variables = new HashMap<>(); !!
statements ::= statement statements
| statement !! System.out.println("Final variable map " + variables); !!

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

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

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

e2 ::= INTEGER !!e2.v = INTEGER1.v; !! // Integer literal
| ID !! e2.v = variables.get(ID1.v); !! // Variable access
| '(' e1 !! e2.v = e11.v; !! ')' // Parenthesised expression

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

Our main goal is conciseness, and both reduction and attribute-action specifications achieve that. The reduction specification contains only 15 syntax rules, and 18 rewrite rules. The attribute-action specification contains only eight context free rules and 15 actions. Each specification gives us variables, expressions, assignment, selection and loops (albeit with only a few operators, one type and no procedural abstraction).

Now let us explain how to read these rules, and how to design with them.

Chapter 3

Rewriting

Sometimes things look different but mean the same thing. For instance the program fragment 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 the string representation of the 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 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 program translation just by using equations.

3.1 Semantic equivalence 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.print(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 to get the same output effect we could just have written

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

These three fragments are all 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()`. Usually, our notion of semantic equivalence does *not* include performance, but only the values computed by a program.

Establishing semantic equivalence of fragments may be very easy, or very hard indeed. For instance, there is a simple recipe which will convert **for** loops to the form in our second fragment:

for(I ; P ; U) S $\quad\rightarrow\quad$ I ; **while**(P){ S ; U }

In this recipe, I stands for any valid initialisation expression, P for a predicate (an expression yielding a boolean), U is an update expression and S stands for the statement which is controlled by **for** construct.

In this example, I is **int** i = 1, P is i<10, U is i++ and S is System.out.print(i + "␣")

rewrite rule

rewrite schema

This recipe is our first example of a *rewrite rule*. Strictly speaking, this is a *rewrite schema* because the placeholders I, P, U, S are *variables* that can stand for any expression of arbitrary complexity and so the schema represents an infinite set of individual rewrite rules, each involving explicit expressions that do not contain placeholder variables.

Note how the structure of the Java **for** construct neatly brings together all four of the elements that we need to perform the transformation to a **while** construct.

If we reverse the direction of the arrow, the resulting rule is not so useful:

I ; **while**(P){ S ; U } $\quad\rightarrow\quad$ **for**(I ; P ; U) S

We could use this reverse-rule to undo our particular transformation from **for** to **while**, but for most **while** loops it would be quite hard to figure out where the four placeholder expressions are – for instance the initialisation expression might be a long way before the **while** keyword. The initialisation expression might even be inside an **if** statement, and so the start value for the loop might be data-dependent. To correctly rework general **while** statements to **for** would require a detailed analysis of the whole program.

refactoring

Now, there are many useful transformations that can be implemented: and they for the basis of the *refactoring* transformations that many Integrated Development Environments provide, as shown in Figure 3.1 for Eclipse.

The recipes for performing refactorings can be much more complex than the single rewrite rule above, and some aspects of programming languages (such as unrestricted **void*** pointers in ANSI-C) make it very hard to be sure that refactorings are *safe*, by which we mean that a refactoring transformation has not changed the meaning of the program.

3.2 Static and dynamic properties

static property

dynamic property

A *static property* of a program is one that depends only on the program text. A *dynamic property* depends both on the program text and some particular input;

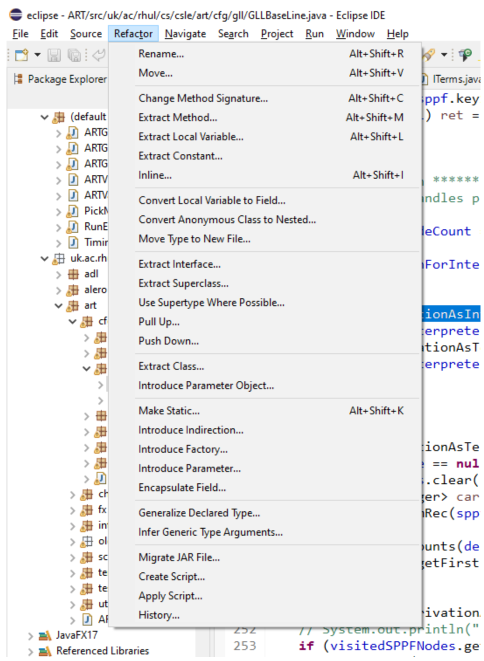


Figure 3.1 Refactoring in Eclipse

hence if we want to evaluate a dynamic property of a program we shall need to run the program, and that run will only tell us about the behaviour in response to a particular input. Dynamic properties demand that we use a testing-style strategy to check our assumptions.

Static properties are powerful, in the sense that they tell us things that will be true for *any* run of the program. The rewrite rule above exploits a statically visible relationship between the **for** and **while** constructs in Java to establish semantic equivalence between the first and second versions.

Establishing the equivalence to our third version (the print statement that simply outputs the result of the program) is *much* harder. Although the equivalence is strictly speaking, statically visible, we would need considerable reasoning to establish the equivalence and in practice we would probably just execute the fragments and compare the results: there are no useful structural relationships between the fragments that we can exploit to give a general rewrite rule.

3.3 Mechanised rewriting

We want to *mechanise* our rewriting mechanisms so that a computer can perform rewrites without human intervention. We want to write a program which will take a set of rewrite rules and some input, and then transform the input for us according to those rules. Such a program is called a *rewriter*. An IDE *rewriter*
with refactorings (such as Eclipse)is a specialised rewriter that offers a fixed set

of transformations (though there may be extension mechanisms).

ART is a rewriter that provides two kinds of rewriting: a specialised *string rewriter* which parses input strings — usually user programs written in some human-friendly external syntax — against a set of context-free string rewrite rules to produce a derivation tree, and a *term rewriter* which reworks that tree, perhaps reducing the whole tree to a value with some side effects (reduction semantics) or leaving the tree intact and rewriting data attributes associated with each node (attribute semantics).

The major challenge in implementing a rewriter is in locating a rule whose constants and variables line up with the structure that we are rewriting. The major challenge in using such a rewriter is to write rules which are well-behaved in ways that we shall discuss: primarily we want rules which yield the same outcome regardless of the order in which they are applied.

3.4 String rewriting

Anybody who has used a text editor is familiar with the principle of string rewriting which is really just cut and paste. Let us consider a simple sub-problem, and think about it formally.

3.4.1 Rules for upper casing

Upper-casing a piece of text involves locating all instances of lower case letters in the range $\boxed{\text{a}}$ to $\boxed{\text{z}}$ and rewriting them into their corresponding upper case character in the range $\boxed{\text{A}}$ to $\boxed{\text{Z}}$. We could capture the uppercasing operation as 26 separate rules:

$$a \to A, \quad b \to B, \quad \dots, \quad z \to Z$$

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

We cannot reasonably be expected to write out a separate rewrite rule for every possible case: that would entail writing something of the form $x \to y$ for every string containing lower case alphabetic characters. Even if we limited ourselves to strings of length 5, there would be 26^5 such rewrite rules, which is 11,881,376 separate rewrite rules.

$\boxed{\textit{rule schema}}$

A *rule schema* is a shorthand for a set of rewrite rules in which *variables* are used as placeholders for arbitrary bits of string. We shall use variables α and β to represent substrings of any length, including the empty string which has length zero.

This allows us to write out the more useful relation $\to$ as follows

$$\alpha a\beta \to \alpha A\beta, \quad \alpha b\beta \to \alpha B\beta, \quad \dots, \quad \alpha z\beta \to \alpha Z\beta$$

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

3.4.2 Confluence

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

$$\alpha a\beta \to \beta A\alpha, \quad \alpha b\beta \to \beta B\alpha, \quad \ldots, \quad \alpha z\beta \to \beta Z\alpha$$

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

This is a rather tricky rewrite system to reason about. Consider the string $AxyZ$. This has two lower case letters, and so two rewrites will be needed to achieve a normal form. Depending on the order in which we perform the rewrites, we get different results:

$$AxyZ \to yZXA \to ZXAY$$

$$AxyZ \to ZYAx \to XZYA$$

We say that these rules are *non-confluent*. The term comes from the study of rivers: sometimes rivers split into multiple channels called distributaries might spread out into a delta, or might come back together into a single channel at a confluence. *confluent*

We want programs to give same results whenever we run them with particular inputs, so we need our reduction semantics rules to be confluent, but that can be quite challenging. The core approach is to attach conditions to each rule such that at most one rule can be applied at each rewrite step. We might also attach a priority to each rule, and if more than one is applicable at some step, we choose the highest priority one. We only use this prioritisation as a last resort because ordered prioritised rules are harder to reason about, and this more error prone. On the other hand, ordering the rules can allow more concise specifications.

3.4.3 Avoiding implicit equality tests

This innocuous looking rewrite scheme has a hidden constraint:

$$\alpha a\alpha \to \alpha A\alpha$$

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

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

3.4.4 Avoiding contiguous variables

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

$$\alpha\beta a \rightarrow \alpha A \beta$$

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

α	β	Rewrite
xy	ϵ	xyA
x	y	aAy
ϵ	xy	Axy

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

If were to restrict our variables to only matching a single character then this problem would go away, but that is quite a strict constraint. In ART, we distinguish between simple variables that match a single input element and 'star' variables that can match sequences of elements. We only allow a single star variable on the left hand size of a rule.

3.4.5 Matching, binding and substituting

open expression

substitution

closed

pattern

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

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

match-pattern $\rightarrow$ substitute-pattern

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

3.5 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.

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.

3.6 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++;}
```

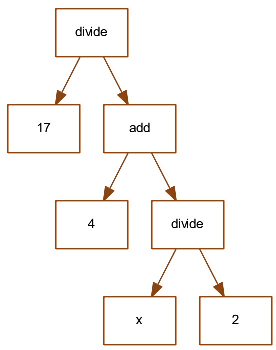
might be written

```
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



3.7 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)`.

3.7.1 Denoting term symbols

In term rewriting, we are very permissive about what constitutes a symbol: theyare not just limited to the kinds of alphanumeric identifiers that we use in conventional programming languages.

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 \(\) \,. 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 \\.

3.8 Rewriting in ART

ART supports both limited string rewriting and general conditional term rewriting. String rewriting is limited in the sense that we only define rules for a parser which can convert external human-friendly syntax into prefix form that we need for our rewrite rules. ART does not actually do any string rewriting; instead the parser works out the ways in which a particular input string couldbe constructed from the rules. If we want to experiment with rewriting itself, we use term rewrite rules.

3.8.1 String rewriting in ART

String rewrite rules must be context free, and are specified using rules that look like this:

```
X ::= 'a' Y 'b'

Y ::= 'y'

Y ::= #
```

nonterminal

context-free terminal

Each rule comprises: a *nonterminal* symbol (X and Y above) followed by the ::= symbol (which you should read as ‘expands-to’) followed a sequence of one or more symbols, each of which may be (a) a nonterminal or (b) a stropped 'string' representing a *context-free terminal*.

As a special case, we may also have rules whose right hand side is a single # character. This specifies that a nonterminal may be rewritten to nothing, i.e. it disappears when we rewrite it. (We usually use ϵ for the empty string when writing in a mathematical style, but few keyboards provide that character so we use # instead.

We shall explore the use of these rules to specify syntax in detail in Chapter 5. As a preview, the way to read these rules is to start with a string comprising the LHS of the first rule: we shall call that the working string. Now select the leftmost nonterminal in the working string, look at the the rule(s) that have on their left hand side and rewrite the working string, replacing the leftmost nonterminal with one of its right hand sides. When the working string has no nonterminals left in it, the rewriting process is complete.

If we apply that process to these rules we get these two derivations

$$X \Rightarrow aYb \Rightarrow ayb$$

and

$$X \Rightarrow aYb \Rightarrow ab$$

Note how in the second derivation, Y has been rewritten to # which does not appear in the actual string: # is a meta-symbol representing 'nothing'.

As a notational convenience, we are allowed to group rules which have the same left hand side together, separating them with the | character which you should read as 'or'. Thus we can write these rules more compactly as

```
X ::= 'a' Y 'b'

Y ::= 'y' | #
```

3.8.2 Term rewriting in ART

Term rewrite rules look like this:

```
– label
optionalCondition1 optionalCondition2 ...
–––
conclusion
```

The label is optional, and serves merely to give the rule a name which can be reported as a rewrite rule is being applied. If absent, thesystem will generate a unique name of the form Rn .

Conditions are also optional. They can be used to check, for instance, the the of values and also to invoke recursive calls to the term rewriter (a technique we shall cover in detail in Chapter 4) but plain unconditional rewrites will not have conditions.

The separating bar --- is required.

transition

relation symbol

The conclusion is written as a *transition* which has a left hand term, a *relation symbol* (usually some sort of arrow) and a right hand term. At any given point the rewriter has a 'current term' which we call Θ (read as capital Theta). The rewriter looks through its rules to find one where the left hand side of the conclusion matches Θ. It then evaluates the conditions above the line, and if they are fulfilled then Θ is rewritten to the right hand side of the conclusion.

Importantly, these terms can have placeholder variables in them, which allow subterms to be carried over unchanged. Variable names always start with a *single* underscore.

The ART script language is free-format so we can space things out vertically, or for simple rules write them all on one line.

So, for example, if we represent the external syntax

'while'expression 'do'statement

using the internal syntax term

while(expression, statement)

then our **while** loop conversion trick can be implemented as

```
--- while(_E, _S) -> if(_E, seq(_S, while(_E,_S)), __done)
```

Informally, when we match the rule to Θ, the variables _E and _S are bound to the expression and statement terms respectively. They are then substituted in to the right hand side term as part of the rewrite.

Chapter 4

Reduction semantics

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

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

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

4.1 The basic idea

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

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

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

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

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

4.2 Execution via substitution

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

Program execution as a set of states with transitions between them representing the execution steps of a program. More formally, we developed the idea of a program execution trace as a series of steps that walk a *transition relation* over *configurations*. A configuration represents the state of a computer, and configuration Γ_1 is related to Γ_2 if and only if Γ_2 can appear immediately after Γ_1 in *some* execution of *some* program. Configurations always contain a program term, and in addition we add entities that represent whatever side effects of program execution we need to record.

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

4.2.1 Configurations

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

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

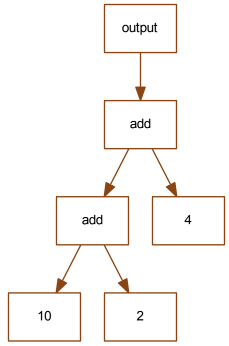
For example, this program fragment when interpreted will eventually result in the value 16 being output.

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

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

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

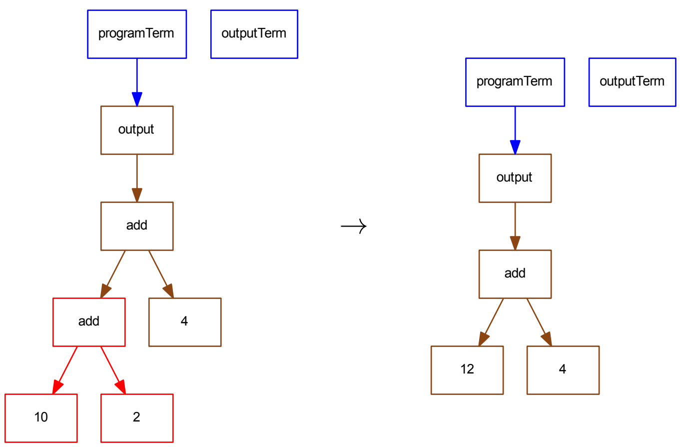
This sort of prefix functional term corresponds directly to a tree if we think of the written term as being the textual trace of a pre-order tree traversal, with parentheses being added to show when we pass down or up a tree edge:



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

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

which we can show graphically as



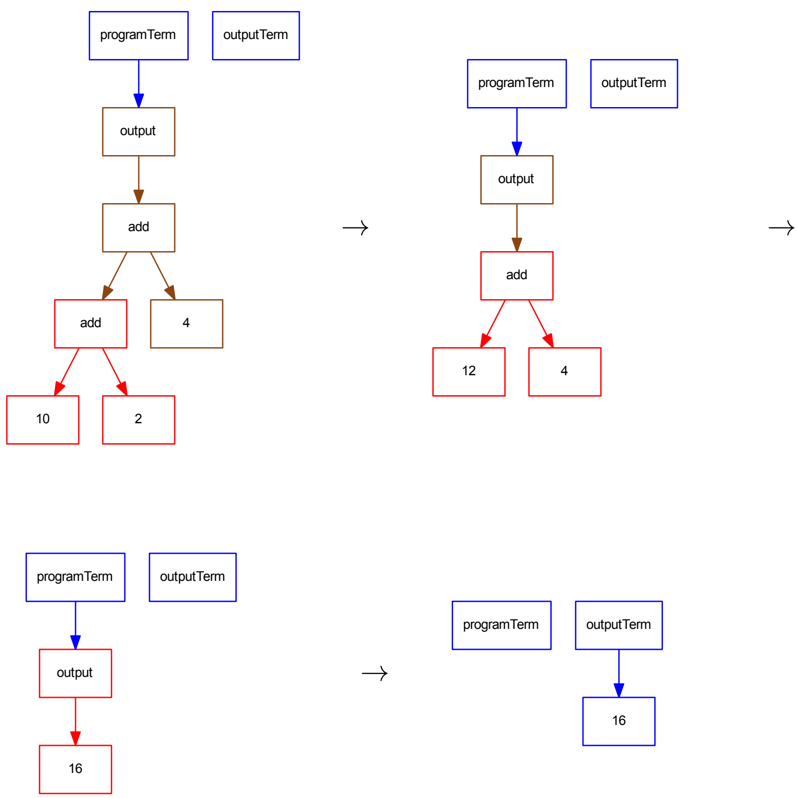
We have highlighted the program part that is about to be rewritten in red. We call this part the *reducible expression* or *redex* for short. We also have blue nodes representing the various state components of a running program: in this case limited to the program term and a presently empty term intended to represent the list of outputs made by a program.

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

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

$\langle$output(add(12,4)), [ ]$\rangle$
$\langle$output(16), [ ]$\rangle$
$\langle$, [16]$\rangle$

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



4.3 Avoiding empty terms – the special value __done

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

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

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

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

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

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

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

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

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

4.4 Term variables are metavariables

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

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

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

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

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

seq(__done, X)$\rightarrow X$

which as an ART specification would be written as

```
--- seq(__done, _X) -> _X
```

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

4.5 Pretty-printed rules

Notice how the red machine-friendly version of the above rule only uses characters that are in the ASCII character set and available on a keyboard; the more human friendly typeset black version uses maths characters like $\rightarrow$ and is less verbose. In ART, we always specify things in the red style, but ART can output pretty-printed typeset versions of the rules for use in LaTeX documents.

4.6 Pattern matching of terms

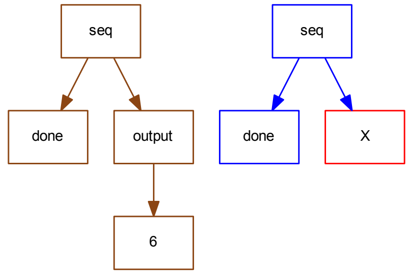
Since we are trying to build a formal game which will execute a program without human intervention, we need some syntax in which to write the rules of the game. As we have already seen, for this reduction of a sequence with __done as its left hand argument we write

seq(__done, X) $\rightarrow X$

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

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

We can represent this process visually as follows:



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

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

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

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

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

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

We shall write

$$\theta \triangleright \pi$$

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

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

returns

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

and

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

returns

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

whereas

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

returns

$$\bot$$

because $\mathsf{output}(6)$ does not match $\mathsf{\_\_done}$.

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

4.7 Pattern substitution

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

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

We shall write

$$\pi \triangleleft \rho$$

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

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

returns

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

Here is a recursive function to perform substitution

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

4.8 Rules and rule schemas

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

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

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

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

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

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

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

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

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

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

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

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

In pseudo code, we might say something like this:

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

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

$$\text{if}\ \ \langle \rho_1 = (\theta \triangleright \mathsf{output(plusOp(}X\,\mathtt{,}Y\mathtt{)))}, \alpha\rangle$$

$$\text{and}\ \ \mathtt{is012(}X\mathtt{)} \triangleright \textit{true}$$

$$\text{and}\ \ \mathtt{is012(}Y\mathtt{)} \triangleright \textit{true}$$

$$\text{and}\ \ \rho_2 = ((\mathtt{\_\_add(}X\,\mathtt{,}Y\mathtt{)} \triangleleft \rho_1) \triangleright Z)$$

$$\text{then}\ \ \theta \rightarrow \theta' = \langle \mathsf{\_\_done}, [Z \triangleleft \rho_2]\rangle$$

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

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

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

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

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

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

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

The inference rule representation of our schema is

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

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

This allows us to abbreviate our inference rule to:

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

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

4.9 The interpreting function F_{SOS}

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

4.9.1 Managing the local environment

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

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

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

4.9.2 Procedural pseudo-code for F_{SOS}

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

```
let R be the set of rule schemas

Fsos(C: configuration) returns configuration OR bottom
  for r in R
    if C |> r.conclusion.lhs then
      let E be an empty set of bindings
      for c in R.conditions
        if isSideCondition(c)
          let res be (c.lhs <| E) |> c.rhs
```

```
10             if res = bottom then next r else add res to E
11           else
12             let T be c.lhs <| E
13             if isvalue(T) then return T
14             let res be Fsos(T) |> c.rhs
15             if res = bottom then next r else add res to E
16         return r.conclusion.rhs <| E
17   return bottom
```

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

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

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

Conditions can be either side conditions or transitions.

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

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

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

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

4.9.3 Program term rewrites - the outer interpreter

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

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

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

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

  return C
```

4.10 Structural Operational Semantics and execution traces

It is clear that in some sense the structure of the term to be executed specifies the execution order.

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

We shall now look in detail at some rules, and use ART's trace facility to see exactly what happens as the rewriter attempts to reduce a program to a value.

4.10.1 SOS rules for a subtraction language with output

As a first example, let us generalise the 0, 1, 2 addition language of the previous section to allow expressions involving arbitrarily nested additions of 32-bit integers with output.

We begin with a rule that performs addition for expressions such as $3+4$, that is where each operand is a single integer. The rule uses our style of abstract syntax, which would encode this example as add(3, 4).

It is convenient to name rules for reference purposes by giving a label at the start of the rule compising a – sign and an alphanumeric name. These names have no meaning in themselves but we often use names that indicate the purpose of the rule. If we do not use a label then ART gnerates a name of the form Rn when n is a rule number. The ART rewriter !trace facility will announce rules as they are used, and this is very helpful when debugging.

Configurations

We begin by deciding on a configuration for our language. Since we are only adding literal numbers we do not have program variables, and thus do not need a store σ or an environment ρ to act as a symbol table. However we do want to model the output, so we need a an output list which we shall call α.

In some specifications it is useful to have more than one relation, represented by different kinds of arrows each of which has its own configuration so we need to specify which relation we mean when we set up a configuration. When we specify a configuration for a particular relation, we write something of the form !configuration –> x:y, ... where x:y specifies a semantic entity with x is the root name of the variables we shall use to manipulate that entity and y is its type. This is a suitable directive for our language:

```
!configuration -> _alpha:__list
!trace 5
```

Here we have also set the trace level to 5 so as to show the variable bindings developed during a rule test.

Adding literals

Now we add a rule for addition which can only handle two literal numbers, and test it using a !try. Notice how the !try and both sides of the transition have exactly the same shape as the configuration: a program term followed by a literal empty __list for the !try and a program term followed by a variable for the rule.

```
-add
_n1 |> __int32(_) _n2 |> __int32(_)
---
add(_n1, _n2), _alpha -> __add(_n1,_n2), _alpha

!try add(5,3), __list
```

The match expressions above the line (`_n1 |> __int32(_) _n2 |> __int32(_)`) are checking that both operands really are of type`__int32`. Without them, we might end up calling the `__add` function with invalid arguments, and that would give a fatal error that would stop the rewriter.

When we run this example at trace level 5 we get a blow-by-blow description of the rewriter's actions as it works its way round the rule, and eventually calls the `__add` Value function:

```
Step 1
  Rewrite attempt 1: <add(5, 3), []> ->
  -add _n1 |> _ _n2 |> _ --- <add(_n1, _n2), _alpha>-><__add(_n1, _n2), _alpha>
  -add bindings after Theta match { _n1=5, _n2=3, _alpha=[] }
  -add premise 1 _n1 |> _
  -add bindings after premise 1 { _n1=5, _n2=3, _alpha=[] }
  -add premise 2 _n2 |> _
  -add bindings after premise 2 { _n1=5, _n2=3, _alpha=[] }
  -add rewrites to <8, []>
Normal termination on <8, []> after 1 step and 1 rewrite attempt
```

Nested expressions

In general, we would like to handle nested expressions such as `add(3, add(4,5))`. However, our simple `-add` rule will not rewrite such expressions:

```
!try add(3, add(4,5)), __list
```

yields

```
1 Step 1
2   Rewrite attempt 1: <add(3, add(4, 5)), []> ->
3   -add _n1 |> _ _n2 |> _ --- <add(_n1, _n2), _alpha>-><__add(_n1, _n2), _alpha>
4   -add bindings after Theta match { _n1=3, _n2=add(4, 5), _alpha=[] }
5   -add premise 1 _n1 |> _
6   -add bindings after premise 1 { _n1=3, _n2=add(4, 5), _alpha=[] }
7   -add premise 2 _n2 |> _
8   -add premise 2 failed: seek another rule
9 Stuck on <add(3, add(4, 5)), []> after 1 step and 1 rewrite attempt
```

At line 5, we test the first *premise* (that is the leftmost match or transition expression). That succeeds, since `_n1` is bound to `__int32(3)` and that *does* match __int32(_). However, at line 7 we test the second premise and that fails because _n2 is bound to add(4, 5) and that does *not* match __int32(_). As a result, the whole rule fails. There are no other rules, so the rewriter reports that it is stuck.

What we need are *resolution rules* which can take the inner expression add(3,4) and rewrite it to a simpler form. Here is a suitable rule which will recursively call the rewriter on the right operand instead of just trying to match it against a type:

```
-addResolveRight
_n |> __int32(_) _E, _alpha -> _EP, _alphaP
---
add(_n, _E), _alpha -> add(_n,_EP), _alphaP
```

Note how the first premise _n |> __int32(_) simply requires the left argument to be a 32-bit integer. The second premise _E, _alpha -> _EP, _alphaP requires the rewriterto take the second argument _E and separately *transition* it to some new _EP (expression primed). Note also that both sides of the transition have the same shape as the configuration: the original output list _alpha appears on the left hand side of the transition, but the resulting list is called _alphaP, and it is _alphaP that is used on the right hand side of the conclusion. We do this to ensure that any updates to the list that occur as side effects of the resolution are propagated to the final rewrite.

When we run !try add(3, add(4,5)), __list with rules -add and -addResolveRight we get quite a long trace. It is very important to work through this trace and ensure that everything makes sense: this is the first time we have seen the 'transition above the line' idiom: it is perhaps the most important idea in this chapter.

```
Step 1
  Rewrite attempt 1: <add(3, add(4, 5)), []> ->
  -add _1 |> _ _2 |> _ --- <add(_1, _2), _3>-><__add(_1, _2), _3>
  -add bindings after Theta match { _1=3, _2=add(4, 5), _3=[] }
  -add premise 1 _1 |> _
  -add bindings after premise 1 { _1=3, _2=add(4, 5), _3=[] }
  -add premise 2 _2 |> _
  -add premise 2 failed: seek another rule
  -addResolveRight _n |> _ <_E, _alpha>-><_EP, _alphaP> ---
                   <add(_n, _E), _alpha>-><add(_n, _EP), _alphaP>
  -addResolveRight bindings after Theta match { _n=3, _E=add(4, 5), _alpha=[] }
  -addResolveRight premise 1 _n |> _
  -addResolveRight bindings after premise 1 { _n=3, _E=add(4, 5), _alpha=[] }
  -addResolveRight premise 2 <_E, _alpha>-><_EP, _alphaP>
    Rewrite attempt 2: <add(4, 5), []> ->
    -add _1 |> _ _2 |> _ --- <add(_1, _2), _3>-><__add(_1, _2), _3>
    -add bindings after Theta match { _1=4, _2=5, _3=[] }
    -add premise 1 _1 |> _
```

```
19     -add bindings after premise 1 { _1=4, _2=5, _3=[] }
20     -add premise 2 _2 |> _
21     -add bindings after premise 2 { _1=4, _2=5, _3=[] }
22     -add rewrites to <9, []>
23   -addResolveRight bindings after premise 2 { _1=3, _2=add(4, 5), _3=[], _4=9, _5=[] }
24   -addResolveRight rewrites to <add(3, 9), []>
25 Step 2
26   Rewrite attempt 3: <add(3, 9), []> ->
27   -add _1 |> _ _2 |> _ --- <add(_1, _2), _3>-><__add(_1, _2), _3>
28   -add bindings after Theta match { _1=3, _2=9, _3=[] }
29   -add premise 1 _1 |> _
30   -add bindings after premise 1 { _1=3, _2=9, _3=[] }
31   -add premise 2 _2 |> _
32   -add bindings after premise 2 { _1=3, _2=9, _3=[] }
33   -add rewrites to <12, []>
34 Normal termination on <12, []> after 2 steps and 3 rewrite attempts
```

This trace breaks down into two steps. In Step 1 we rewrite the program term add(3, add(4, 5)) to add(3, 9) by performing an inner rewrite on add(4, 5) to9, and in the second step we rewrite add(3, 9) to 12. The second step is essentially the same as that for our earlier example on page 53 so we do not need to go through that again.

The first step is more interesting. In lines 2–8 we see the rewriter trying the −add rule first, and rejecting it because premise 2 fails. The rewriter then moves to the −addResolveRight rule. At lines 12–13, premise 1 is checked successfully. Line 15–22 show the rewriter recursing — at line 15 the rewriter announces that is is starting rewrite attempt 2, and lines 15–22 are indented to show that this is an *inner* rewrite. We see that the variable_E has effectively chopped the subexpression add(4, 5) out of the original term, and the rewriter can now process that in isolation,starting with the −add rule which will accept this subexpression since it only has integer literal arguments. Once it has rewritten to 9, we can restart the outer rewrite attempt.

This example shows that our rewrite rules form an inductive specification, which we can process in a natural way using recursion. We shall use this idea over and over again.

The standard triad of rules for arity-2 (diadic) operations

Now, we are not quite finished with our rules for addition! We need a third rule which will handle compound expressions in the *left* argument. Here is the complete set of three rules that we need:

```
-add
_n1 |> __int32(_) _n2 |> __int32(_)
---
add(_n1, _n2), _alpha -> __add(_n1,_n2), _alpha

-addResolveRight
_n |> __int32(_) _E, _alpha -> _EP, _alphaP
---
add(_n, _E), _alpha -> add(_n,_EP), _alphaP

-addResolveLeft
_E1, _alpha -> _E1P, _alphaP
---
add(_E1, _E2), _alpha -> add(_E1P,_E2), _alphaP
```

The new -addResolveLeft rule uses the same mechanism of a transition above the line to resolve the left argument. The right hand argument is simply carried across to the rewritten term. Note that only the first rule -add actually calls the addition function __add. The other two rules both rewrite add(,) terms to add(,) terms.

Here is a more compact trace (at level 3) showing all three rules working together to process !try add(add(2,3), add(4,5)), __list

```
Step 1
  Rewrite attempt 1: <add(add(2, 3), add(4, 5)), []> ->
  -add _n1 |> _ _n2 |> _ --- <add(_n1, _n2), _alpha>-><__add(_n1, _n2), _alpha>
  -addResolveRight _n |> _ <_E, _alpha>-><_EP, _alphaP> ---
                    <add(_n, _E), _alpha>-><add(_n, _EP), _alphaP>
  -addResolveLeft <_E1, _alpha>-><_E1P, _alphaP> ---
                   <add(_E1, _E2), _alpha>-><add(_E1P, _E2), _alphaP>
    Rewrite attempt 2: <add(2, 3), []> ->
    -add _n1 |> _ _n2 |> _ --- <add(_n1, _n2), _alpha>-><__add(_n1, _n2), _alpha>
    -add rewrites to <5, []>
  -addResolveLeft rewrites to <add(5, add(4, 5)), []>
Step 2
  Rewrite attempt 3: <add(5, add(4, 5)), []> ->
  -add _n1 |> _ _n2 |> _ --- <add(_n1, _n2), _alpha>-><__add(_n1, _n2), _alpha>
  -addResolveRight _n |> _ <_E, _alpha>-><_EP, _alphaP> ---
                    <add(_n, _E), _alpha>-><add(_n, _EP), _alphaP>
    Rewrite attempt 4: <add(4, 5), []> ->
    -add _n1 |> _ _n2 |> _ --- <add(_n1, _n2), _alpha>-><__add(_n1, _n2), _alpha>
    -add rewrites to <9, []>
```

```
  -addResolveRight rewrites to <add(5, 9), []>
Step 3
  Rewrite attempt 5: <add(5, 9), []> ->
  -add _n1 |> _ _n2 |> _ --- <add(_n1, _n2), _alpha>-><__add(_n1, _n2), _alpha>
  -add rewrites to <14, []>
Normal termination on <14, []> after 3 steps and 5 rewrite attempts
```

Step 1 rewrites add(2,3) to 5, step 2 rewrites add(4,5) to 9 and step 3 rewrites add(5,9) to 14.

A triad of rules in this particular order is the standard way in ART to manage arity-2 operations such as arithmetic, logic and relation functions.

It is important that the rules are listed in order from the most specific (rule -add which demands literal integer operands) to the most general. If we put -addResolveLeft first, then the rewriter will use it recurse down into the left hand operands until it finds a terminal, then rewrite to the same terminal. The recursion unwinds with each transition simply replicating its left hand side until we get back to the top, at which point the rewriter notices that nothing has changed and announces that it is stuck.

Handling output

The rules above make provision for an output list called α by having a configuration comprising a program term and a list, and by ensuring that any updates to the list are propogated across transitions. However, we have not yet done any output; here is the only rule we need to add output to our language:

```
-output
_E, _alpha -> _EP, _alphaP
---
output(_E), _alpha -> __done, __put(_alphaP, 0, _EP)
```

Rule -output resolves its single argument in the same way as the addition resolution rules, by recursively reducing the argument to a value _EP. We also keep track of changes to the output with _alphaP just in case evaluating our argument itself makes changes to the output.

```
Step 1
  Rewrite attempt 1: <output(add(3, 4)), []> ->
  -output <_E, _alpha>-><_EP, _alphaP> --- <output(_E), _alpha>-><__done, __put(_alphaP, 0, _EP)>
    Rewrite attempt 2: <add(3, 4), []> ->
    -add _n1 |> _ _n2 |> _ --- <add(_n1, _n2), _alpha>-><__add(_n1, _n2), _alpha>
    -add rewrites to <7, []>
  -output rewrites to <__done, [7]>
Normal termination on <__done, [7]> after 1 step and 2 rewrite attempts
```

Sequencing

We would like to be able to execute a sequence of output statements so as to build an output list: at the moment we can only reduce a single output() and so our output list can never be more than one element long.

Now, some programming languages are *expression based* in which every phrase returns a value, but most (including Java and C) distinguish between *statements* and *procedures* which do not return a value and *expressions* and *functions* which do.

Our internal syntax is inherently expression based, but we use the special value __done to represent an expression which is really a statement, just as we use the pseudo-type void in Java to show that a method is a procedure that returns nothing rather than a function that returns a value.

We represent sequencing using the arity-2 constructor seq, so that a sequence of two output command would be written as seq(output(...), output(...)) and a sequence of three outputs as seq(output(...), seq(output(...), output(...))) The seq expressions nest in the same was as arithmetic expressions, and a one arity-2 constructor is enough to describe sequences of any length.

The whole point of a sequence is to perform the leftmost element before the rightmost. We impose this ordering in the same way that we controlled the evaluation order of the add arguments. The idea is that we have a resolution rule -sequence that reduces its left argument to __done and base rule -sequenceDone that only matches sequences that have already been reduced, and which rewrites itself to its own right argument:

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

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

Here is result of a level 2 trace using these rules (and the output rule) to reduce a sequence of two output statements:

```
!try seq(output(3),output(4)),__list
```

```
Step 1
  -sequence rewrites to <seq(__done, output(4)), [3]>
Step 2
  -sequenceDone rewrites to <output(4), [3]>
Step 3
```

```
  -output rewrites to <__done, [4, 3]>
Normal termination on <__done, [4, 3]> after 3 steps and 6 rewrite attempts
```

We can see that the first step rewrites the left argument to __done, updating the output to [3] as a side effect. The second step then rewrites away the __done and the third step puts 4 onto the output list.

Here is a more detailed level 3 trace of the same !try:

```
Step 1
  Rewrite attempt 1: <seq(output(3), output(4)), []> ->
  -sequenceDone --- <seq(__done, _C), _alpha>-><_C, _alpha>
  -sequenceDone Theta match failed: seek another rule
  -sequence <_C1, _alpha>-><_C1P, _alphaP> --- <seq(_C1, _C2), _alpha>-><seq(_C1P, _C2), _alphaP>
    Rewrite attempt 2: <output(3), []> ->
    -output <_E, _alpha>-><_EP, _alphaP> --- <output(_E), _alpha>-><__done, __put(_alphaP, 0, _EP
      Rewrite attempt 3: <3, []> ->
        Terminal <3, []>
    -output rewrites to <__done, [3]>
  -sequence rewrites to <seq(__done, output(4)), [3]>
Step 2
  Rewrite attempt 4: <seq(__done, output(4)), [3]> ->
  -sequenceDone --- <seq(__done, _C), _alpha>-><_C, _alpha>
  -sequenceDone rewrites to <output(4), [3]>
Step 3
  Rewrite attempt 5: <output(4), [3]> ->
  -output <_E, _alpha>-><_EP, _alphaP> --- <output(_E), _alpha>-><__done, __put(_alphaP, 0, _EP)>
    Rewrite attempt 6: <4, [3]> ->
      Terminal <4, [3]>
  -output rewrites to <__done, [4, 3]>
Normal termination on <__done, [4, 3]> after 3 steps and 6 rewrite attempts
```

4.10.2 Building rules for a full language

We now switch to building rules for a language that has assignment, but no output. We will have a single scope region, so we do not need separate symbol tables (represented by environments ρ) and can thus make do with just a store, so the configuration is:

```
!configuration -> _sig:__map
```

Assignment and dereferencing

Assignment is the process of creating (or updating) a binding in the store between an identifier and a value. Dereferencing is the process of replacing an identifier with its bound value in the store. We use the __map operations __put and __get to manipulate _sig.

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

-assignR
_E,_sig -> _I,_sigP
---
assign(_X,_E),_sig -> assign(_X,_I),_sigP

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

Control flow

For the GCD language, we need sequencing, conditionals and a basic loop:

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

-sequence
_C1,_sig -> _C1P,_sigP
---
seq(_C1,_C2),_sig -> seq(_C1P,_C2),_sigP

-ifTrue
--- if(true,_C1,_C2),_sig -> _C1,_sig

-ifFalse
--- if(false,_C1,_C2),_sig -> _C2,_sig

-ifResolve
_E,_sig ->_EP,_sigP
---
if(_E,_C1,_C2),_sig -> if(_EP,_C1,_C2),_sigP

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

Note how the sequencing rules are deliberately slightly different to those on page pagerefseq: now we have a store _sig instead of an output list alpha and the labels and variable names (which have no semantic significance) are modified.

Arithmetic and relational operations

Our example only required greater-then, not-equal and subtraction so that is all we have implemented here, using the standard triad of rules. We show the gt rules in vertically-spaced style, and then a more compressed form for the other operations.

```
-gt
_n1 |> __int32(_) _n2 |> __int32(_)
---
gt(_n1,_n2),_sig -> __gt(_n1,_n2),_sig

-gtR
_n |> __int32(_) _E2,_sig -> _I2,_sigP
---
gt(_n,_E2),_sig -> gt(_n,_I2),_sigP

-gtL
_E1,_sig -> _I1,_sigP
---
gt(_E1,_E2),_sig -> gt(_I1,_E2),_sigP

-ne _n1 |> __int32(_) _n2 |> __int32(_) --- ne(_n1,_n2),_sig -> __ne(_n1,_n2),_sig
-neR _n |> __int32(_) _E2,_sig -> _I2,_sigP --- ne(_n,_E2) ,_sig -> ne(_n,_I2),_sigP
-neL _E1,_sig -> _I1,_sigP --- ne(_E1,_E2),_sig -> ne(_I1,_E2),_sigP

-sub _n1 |> __int32(_) _n2 |> __int32(_) --- sub(_n1,_n2),_sig -> __sub(_n1,_n2),_sig
-subR _n |> __int32(_) _E2,_sig -> _I2,_sigP --- sub(_n,_E2),_sig -> sub(_n,_I2),_sigP
-subL _E1,_sig -> _I1,_sigP --- sub(_E1,_E2),_sig -> sub(_I1,_E2),_sigP
```

4.11 An eSOS specification for the GCD language

Finally, here is a complete set of rules for the GCD language, covering only those features required for the example on page 15, written in a compressed style.

```
-assign _n |> __int32(_) --- assign(_X,_n),_sig -> __done,__put(_sig,_X,_n)
-assignR _E,_sig -> _I,_sigP --- assign(_X,_E),_sig -> assign(_X,_I),_sigP

-deref --- deref(_R),_sig -> __get(_sig,_R),_sig

-sequenceDone --- seq(__done, _C),_sig -> _C,_sig
-sequence _C1,_sig -> _C1P,_sigP --- seq(_C1,_C2),_sig -> seq(_C1P,_C2),_sigP

-ifTrue --- if(true,_C1,_C2),_sig -> _C1,_sig
-ifFalse --- if(false,_C1,_C2),_sig -> _C2,_sig
```

```
-ifResolve _E,_sig ->_EP,_sigP --- if(_E,_C1,_C2),_sig -> if(_EP,_C1,_C2),_sigP

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

-gt _n1 |> __int32(_) _n2 |> __int32(_) --- gt(_n1,_n2),_sig -> __gt(_n1,_n2),_sig
-gtR _n |> __int32(_) _E2,_sig -> _I2,_sigP --- gt(_n,_E2),_sig -> gt(_n,_I2),_sigP
-gtL _E1,_sig -> _I1,_sigP --- gt(_E1,_E2),_sig -> gt(_I1,_E2),_sigP

-ne _n1 |> __int32(_) _n2 |> __int32(_) --- ne(_n1,_n2),_sig -> __ne(_n1,_n2),_sig
-neR _n |> __int32(_) _E2,_sig -> _I2,_sigP --- ne(_n,_E2) ,_sig -> ne(_n,_I2),_sigP
-neL _E1,_sig -> _I1,_sigP --- ne(_E1,_E2),_sig -> ne(_I1,_E2),_sigP

-sub _n1 |> __int32(_) _n2 |> __int32(_) --- sub(_n1,_n2),_sig -> __sub(_n1,_n2),_sig
-subR _n |> __int32(_) _E2,_sig -> _I2,_sigP --- sub(_n,_E2),_sig -> sub(_n,_I2),_sigP
-subL _E1,_sig -> _I1,_sigP --- sub(_E1,_E2),_sig -> sub(_I1,_E2),_sigP
```

Here is a GIFT-annotated grammar which translates from our human-friendly syntax to terms suitable for use with these rules.

```
seq ::= statement^^ | statement seq
statement ::= assign^^ | while^^ | if^^
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
```

And here are two !trys that firstly exercise the rules using an explicit term, and secondly run a human-friendly string through the parser with the resulting term passed to the rewriter.

```
!try seq(assign(a,6), seq(assign(b,9), while(ne(deref(a), deref(b)),
    if(gt(deref(a), deref(b)), assign(a, sub(deref(a), deref(b))), assign(b, sub(deref(b), deref(a))))))),
    __map
!try "a := 6; b := 9; while a != b do if a > b then a := a - b; else b := b - a;"
```

The two tries give identical results: in fact the term above was generated with the parser.

```
Normal termination on <__done, {a=3, b=3}> after 37 steps and 90 rewrite attempts
Normal termination on <__done, {a=3, b=3}> after 37 steps and 90 rewrite attempts
```

Chapter 5

Context Free Grammars and parsing

A context free grammar (CFG) is a string rewriting system which gives rules for composing sentences in a language. Each sentence is a string of characters or of words separated by whitespace. The complete collection of valid sentences specified by a grammar is called the *language* of that grammar.

Nearly all programming language standards documents use CFG rules to specify the syntactically valid forms of programs. In this chapter we look at the basic idea of a CFG; show how to specify common language idioms using CFG rules; and show how to add so-called GIFT annotations to CFG rules so as to generate terms suitable for the reduction-semantics rewriter we shall discuss in the next chapter.

There are three different ways that we might use a CFG:

1. As a language *generator* which takes a CFG Γ and lists valid sentences from $L(\Gamma)$ (the language of Γ) that could be used to test our language processors.

2. As a language *recogniser* which takes a CFG Γ and a string σ and returns boolean TRUE is $\sigma$is a sentence in $L(\Gamma)$ and FALSE otherwise.

3. As a language *parser* which takes a CFG Γ and a string σ and returns *derivations* of σ in Γ.

A derivation is a sequence of rewrite steps showing how σ can be constructed using the rules in Γ.

A *derivation tree* composes those rewrite steps into a tree or term,

There maybe more than one derivation tree for some σ in $L(\Gamma)$ in which case we say that Γ is *ambiguous*.

Γ may have cycles, in which case there will be an infinite set of derivation trees for some $\sigma \in L(\Gamma)$.

A *general CFG parser* is one which can

1. process any CFG;

2. return a representation of *all* of the derivation trees of any $\sigma \in L(\Gamma)$.

ART includes a general parser which uses the MGLL algorithm

5.1 Outer and inner syntax

Language processing tools usually maintain internal representations that are optimised for the task in hand. In the context of language based software tools, we often call this internal form the *abstract* or *inner syntax* of a language. The equivalent technical terms *intermediate form*, *internal representation* and *model* also appear in the literature. The term abstract syntax suggests some sort of relationship to the 'real' syntax of the user language, which is then called the *outer* or *concrete syntax*.

Transformation from outer to inner syntax typically involves (a) throwing away of redundant detail in the outer syntax (such as the parentheses that surround the predicate following an if keyword in Java, (b) 'normalisation' of forms such as converting for and do...while loops in Java to while loops so that only one style of looping has to be treated internally, and (c) rendering the result in the chosen internal syntax style.

Real systems are often layered, with the internal 'abstract' syntax of one layer becoming the starting point for an inner translation. As a result, we find the terms abstract and concrete unhelpful and prefer to use the terms *inner* and *outer* syntax for each layer; the inner syntax for a layer is the outer syntax for its enclosed layer. So for Java, for instance, the outermost syntax is the human-friendly form documented in the Java Language Specification. An implementation of `javac` (the Java compiler) will have its own internal form which is suitable for analysing and checking Java programs, and then some sort of back end which can be thought of as translating from that compiler-specific internal form to Java Virtual Machine Byte code which is then written into the `.class` files. This code then forms the external syntax for the Java Virtual Machine interpreter `java`.

5.1.1 Inner syntax for reduction semantics

The reduction interpreters that we shall describe in the next chapter use term rewriting to successively rewrite program terms into simpler forms, accumulating side effects during the process. For theses kinds of language processors, we have an external syntax which is comfortable for human programmer such as

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

```
    b:=b−a;

gcd := a
```

and a corresponding *inner* syntax which might be

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

This internal syntax is a nested expression over terms such as seq(_X, _Y) which sequences operations together, assign(_N, _v) which binds a value to a name and sub(_v, _w) which evaluates $v - w$.

5.1.2 Syntactic sugar, redundancy and syntactic 'noise' in human-friendly external syntax

An outer syntax designed for humans often contains elements which protect against common error patterns without adding any semantics. For instance, we could design a concise Java conditional expression which allowed us to write expressions such as

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

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

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

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

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

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

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

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

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

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

5.1.3 The legacy of non-general parsing

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

5.2 Chapter overview

The main goal of this chapter is to show how to write CFGs that both describe the valid sentences in your programming language *and* have derivations that are in the right form to be fed directly into an executable reduction semantics based on rewrite rules in the style described Chapter 4.

By writing the CFG carefully we can use a CFG parser to automatically translate from external to internal syntax. In this we are aided by so-called GIFT annotations which specify local folding of nodes in the derivation tree. A complete reduction interpreter is then specified as a set of CFG rewrite rules to convert a program written in external syntax to an initial term written in internal syntax, and a set of rewrite rules that successively reduces that term to a value, accumulating side-effects along the way.

A secondary goal is to give an overview of the parsing process using the simplest algorithm I know of which can do at least *some* useful work: a non-general algorithm called RDSOB (known elsewhere as OSBRD!). The intention is to give insights into the process of parsing, without discussing details of the general MGLL algorithm. We shall then see how to enhance the basic parser with attributes and actions so as to make a rudimentary translation tool. This acts as a preview of the action-attribute technique described in detail in Chapter 6.1.

5.3 CFG definitions and examples

Context Free Grammars have a finite set of literals or *terminal symbols* T, a finite set of *nonterminals* N and a set or rewrite rules or *productions* written, for instance, as X ::= Y 't' Z where X, Y, Z are nonterminals and 't' is a terminal. One of the nonterminals is called the *start* symbol.

More formally, we write:

a CFG Γ is a 4-tuple (N, T, S, P) with $N \cap T = \emptyset, S \in N, P \subseteq N \times (N \cup T)^*$

which we read as

> A Context Free Grammar is a set of nonterminal symbols N, a set of terminal symbols T (N and T disjoint), a special start nonterminal symbol S and a set of productions the left hand side of which is a nonterminal, and the right hand side of which is a (possibly empty) sequence of terminals and nonterminals.

5.3.1 CFGs in ART

In ART, terminal symbols are always stropped 'like' 'this' and nonterminals never are, which ensures that we can immediately see if a symbol is a terminal or a nonterminal. As noted earlier, we prefer to have an explicit symbol for 'nothing', so we represent the empty righthand side of a rules with a hash symbol #. Unless otherwise specified,the first start symbol is the left hand side of the first rule in the specification.

Here is a first example.

```
S ::= 'a' R T
R ::= 'r'
T ::= 't'
```

This CFG has three nonterminals and three terminals, with $N = \{$S, R, T$\}$ and $T = \{$'a', 'r', 't'$\}$. The start symbol S is just S because the first rule is a production for S. Our conventions are useful because in reality all we need to do is list the rewrite rules in P — we can infer N, T and S from that list.

5.3.2 Sentential forms and derivations

A sentence is a (possibly empty) string of terminals. A *sentential form* is a (possibly empty) string of terminals and nonterminals. All sentences are sentential forms (but, of course, not all sentential forms are sentences!).

A derivation step rewrites a sentential form Σ into another sentential form Σ' by (a) picking one of the nonterminals X in Σ and then (b) replacing it by the right hand side of one of the productions of X. Sentential forms that contain no nonterminals cannot be rewritten: they are sentences.

Where a sentential form has more than one nonterminal, which should we pick for rewriting? To use the language of rewriting, how do we select a redex? It turns out that it does not matter since CFG rewriting is always confluent. Hence, by convention we always choose the left-most nonterminal to expand. When we are being careful, we call this the left-derivation.

5.3.3 Generating a language

We generate strings by maintaining a current sentential form which is initialised with just the start symbol. We then repeatedly rewrite the left-most nonterminal, generating new sentential forms and sentences.

We can ask ART to do this for us but watch out because interesting grammars have infinite languages, so we need to specify an upper bound on the number of sentential forms we want or the process will not terminate! Using the example above

```
S ::= 'a' R T
R ::= 'r'
T ::= 't'
!generate 10 sententialforms
```

gives

```
 1 _S
 2 a _R _T
 3 a r _T
"4 a r t
```

ART has listed four sentential forms. In each, nonterminals are preceded with an underscore to show that they are really *variables* which will in due course be replaced. Any sentences (sentential forms without nonterminals) are preceded with a " character.

We have asked for 10 sentential forms, but in fact this very simple grammar only has four sentential forms, so the output stops quickly.

The first sentential form is just the start symbol, as it always is. We then expand the rightmost nonterminal in the only way we can by replacing S with its only right hand side 'a' R T. Next we must rewrite R as it is now the rightmost symbol, and then we rewrite T at which point the sentential form is a sentence, and we are finished.

We sometimes write these derivations in a more compact form as

$$S \Rightarrow aRT \Rightarrow arT \Rightarrow art$$

Here, the convention is that lower case letters represent terminals and upper case nonterminals.

Sometimes we want to miss out intermediate steps, so we write $\stackrel{*}{\Rightarrow}$ to mean 'zero or more derivation steps'. So, for instance, we could say

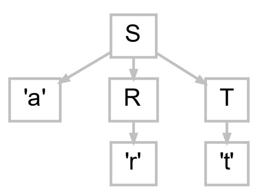
$$S \stackrel{*}{\Rightarrow} arT \Rightarrow art$$

or just

$$S \stackrel{*}{\Rightarrow} art$$

We can now define the language of Γ as $L(\Gamma) = \{u|S \stackrel{*}{\Rightarrow} u, u \in T^*\}$

We can also display the derivation as a *derivation tree*:



The root of the derivation tree is always labelled with the start symbol: it corresponds to the first sentential form in the list above. The leaves of the tree are the terminals in the resulting sentence, and terminals can only appear on the leaves. The internal nodes are labelled with nonterminals, and the children of an internal node are labelled with the elements of the production used to expand that nonterminal instance in the derivation.

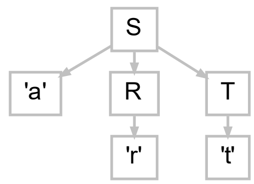
As a special case, the production X ::= # is represented as a node labelled X with a single leaf child labelled #. The empty string # is neither a terminal nor a nonterminal but a denotation for the absence of either; when writing out a sentence from a derivation tree we list the leaf node terminal labels in order, omitting any # labels.

As usual, we can represent the derivation tree textually as a prefix form term: for this tree the corresponding term is S(a, R(r), T(t))

5.3.4 Four representations of derivations

We have seen four representations of the same derivation:

```
 1 _S
 2 a _R _T
 3 a r _T
”4 a r t
```



$\boxed{S \Rightarrow aRT \Rightarrow arT \Rightarrow art}$

S(a, R(r), T(t))

These representations are all equivalent in the sense that they contain the same information, and that given one we can immediately write out the other three. They are in fact three different external syntaxes for the same internal object — a derivation. All but the bottom left one are outputs from ART; we shall usually use the term style at bottom right.

5.3.5 Adding a rule

Let us add another rewrite rule for nonterminal R

```
S ::= 'a' R T
R ::= 'r' 'r'
R ::= 'r'
T ::= 't'
!generate 10 sententialforms
```

This yields

```
 1 _S
 2 a _R _T
 3 a r _T
 4 a r r _T
"5 a r t
"6 a r r t
```

Sentential forms 3 and 4 show the R in form 2 being replaced by two different productions, and their further expansion to two different sentences at 5 and 6.

ART allows us to group together CFG rules that have the same left hand side, writing the alternate right hand sides out separated by | characters (which we read as 'OR'), so we can rewrite the rule for R as R ::= 'r' 'r' | 'r' giving

```
S ::= 'a' R T
R ::= 'r' 'r' | 'r'
T ::= 't'
!generate 10 sententialforms
```

As you would expect, running this script gives the same output as above.

5.3.6 The empty string

At first glance, one might assume that sentential forms always get longer, starting with the single element sentential form containing just the start symbol and building up to a sentence. However, a nonterminal can have an *empty production*, or to put it another way a nonterminal can 'expand' to the empty string (and thus just be erased from a sentential form).

We need to take a little care when discussing the empty string: in fact we need a clear way to denote it. Just as the symbol 0 means 'nothing' in arithmetic and $\emptyset$ means an empty set, we need to agree how to denote a CFG rule whose right hand side is empty.

We *could* use this notation

```
S ::= 'a' R T
R ::=
T ::= 't'
!generate 10 sententialforms
```

that is, just have nothing on the right hand side (and indeed some CFG tools work that way), but we prefer an explicit symbol which in ART is the hash character #.

If you try the form above, you will get a syntax error from ART

```
31,2 GLLBL syntax error
   30: R ::=
   31: T ::= 't'
---------^
   32: !generate 10 sententialforms
*** Fatal: Syntax error in script
```

The correct way to write the rule is:

```
S ::= 'a' R T
R ::= #
```

```
T ::= 't'
!generate 10 sententialforms
```

which generates the following output: in sentential form 2 the R nonterminal is rewritten to the empty string and so disappears; the eventual sentence is then just at.

```
 1 _S
 2 a _R _T
 3 a _T
"4 a t
```

A hash symbol # will never appear in a sentential form (and by extension, never in a sentence either) since it is neither a terminal nor a nonterminal — it is a metasymbol representing 'nothing'.

Similarly, # can never appear in a CFG right hand side except when it is the entire right hand side. This rule also generates a syntax error because it has a # embedded within a sequence of grammar symbols.

```
R ::= 'r' # A
```

gives this error

```
45,10 GLLBL syntax error
   44:
   45: R ::= 'r' # A
-----------------^
   46:
*** Fatal: Syntax error in script
```

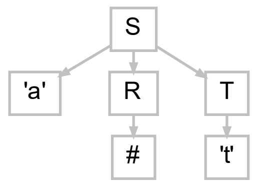
We *are* still allowed to combine right hand sides with the | symbol, so this is a legal CFG:

```
S ::= 'a' R T
R ::= 'r' | #
T ::= 't'
!generate 10 sententialforms
```

which yields

```
 1 _S
 2 a _R _T
 3 a r _T
 4 a _T
"5 a r t
"6 a t
```

Perhaps counter-intuitively, we *do* put an explicit # node into the tree representation of a derivation. The derivation of at above is $S \Rightarrow aRT \Rightarrow aT \Rightarrow at$ and the corresponding tree is



This is because it is important for some operations that nonterminals only label internal nodes of the tree, and that terminals only label leaves of the tree. Without this #-labelled node we would have a nonterminal-labelled leaf. When reading the sentence off of a derivation tree, we list the leaves left to right and simply omit any # nodes.

5.3.7 Using recursion to specify infinite languages

There are only two sentences in our language so far. How would we specify the language $\{art, arrt, arrrt, arrrrt, \ldots\}$ that is, the set of strings which start with an a, finish with a t and have one or more r characters between? The trick is to use an inductive (recursive) specification:

```
S ::= 'a' R T
R ::= 'r' | 'r' R
T ::= 't'
!generate 10 sententialforms
```

The rule for R now expands an instance of R either to the terminal 'r' or to 'r' R. In the second case, we effectively just insert an r into the sentential form, followed by a new instance of R so we can go round again over and over until we finally rewrite the R to just 'r'.

ARTs output shows this in action:

```
1 _S
 2 a _R _T
 3 a r _T
 4 a r _R _T
"5 a r t
 6 a r r _T
 7 a r r _R _T
"8 a r r t
 9 a r r r _T
 10 a r r r _R _T
```

ART is performing *breadth first expansion* here in which all of the possible left most expansions of the current sentential form are loaded onto the end of queue after which the start of the queue is unloaded to the current sentential form.

If we just want to see sentences, we can request them with !generate 10 sentences which gives

```
"1 a r t
"2 a r r t
"3 a r r r t
"4 a r r r r t
"5 a r r r r r t
"6 a r r r r r r t
"7 a r r r r r r r t
"8 a r r r r r r r r t
"9 a r r r r r r r r r t
"10 a r r r r r r r r r r t
```

5.3.8 Left and right recursion

The previous rule R ::= 'r' | 'r' R is an example of *right* recursion because the recursive self-reference is at the right hand end. We can generate the same language with the left recursive rule R ::= 'r' | R 'r'

```
S ::= 'a' R T
R ::= 'r' | 'r' R
T ::= 't'
!generate 10 sententialforms
```

yields

```
 1 _S
 2 a _R _T
 3 a r _T
 4 a _R r _T
"5 a r t
 6 a r r _T
 7 a _R r r _T
"8 a r r t
 9 a r r r _T
 10 a _R r r r _T
```

In this case we again insert an r into the sentential form, but this time preceded by a new instance of R.

In future examples we shall be interested in exploiting these variant growth patterns,both of which are useful. However some parsing techniques, including the RDSOB algorithm described in this chapter, will go into an infinite loop when processing a left recursive grammar rule. ARTs default MGLL parser does not have this defect and will handle any CFG specification.

5.3.9 Recursion and the empty string

What if we want to generate the language $\{at, art, arrt, arrrt, \ldots\}$, that is, strings that start with an `a`, end with a `t` and have *zero* or more internal `r` characters (as opposed to one or more in the preceding examples).

We simply change the base case of the recursive rule to match the empty string to give:

```
S ::= 'a' R T
R ::= # | 'r' R
T ::= 't'
!generate 10 sentences
```

which yields

```
"1 a t
"2 a r t
"3 a r r t
"4 a r r r t
"5 a r r r r t
"6 a r r r r r t
"7 a r r r r r r t
"8 a r r r r r r r t
"9 a r r r r r r r r t
"10 a r r r r r r r r r t
```

5.3.10 Ambiguity

There may more than one way to generate a sentence; that is there may be some sentence σ which can be derived in more than one way. A CFG with this property is called *ambiguous*. Here is an example of an ambiguous CFG which generates only one sentence, but in two different ways.

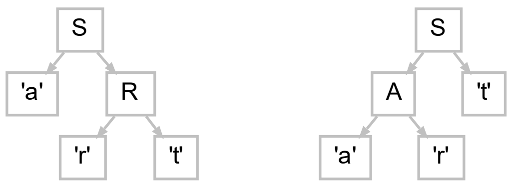
```
S ::= 'a' R | A 't'
A ::= 'a' 'r'
R ::= 'r' 't'
!generate 10 sententialforms
```

yields

```
 1 _S
 2 a _R
 3 _A t
"4 a r t
"5 a r t
```

There are two derivations here: $S \Rightarrow aR \Rightarrow art$ and $S \Rightarrow At \Rightarrow art$ and as a result the sentence *art* appears twice in the list at positions 4 and 5.

The two derivation trees and their terms are



S(a, R(r, t)) S(A(a, r), t)

When parsing, ART will find both derivations. *Usually* we want to select only one, and we can do that with a *chooser* relation which we shall discuss in section 5.5. If no choosers are specified, ART parsers use a default derivation selection strategy called *longest match*: wherever there is a choice of derivation steps we select the one that matches the longer part of the input. In this case, ART would select $S \Rightarrow At \Rightarrow art$ correspoding toS(A(a, r), t) since the A child of S in the right hand derivation matches two elements of the input, whereas the'a' child of S in the left hand derivation only matches one element.

5.3.11 Cycles

A particularly unpleasant situation arises when a CFG has a nonterminal that can derive just itself. Such a rule is called *cyclic* because it generate sentential forms that derive themselves, and so the generator will go round and round repeatedly processing the same sentential form.

```
S ::= 'a' R T
R ::= 'r'| R // cycle!
T ::= 't'
!generate 10 sententialforms
```

yields

```
1 _S
 2 a _R _T
 3 a r _T
```

```
 4 a _R _T
"5 a r t
 6 a r _T
 7 a _R _T
"8 a r t
 9 a r _T
 10 a _R _T
```

The problem here is that every time we use the rule R ::= R we are simply replacing an instance of R with an instance of R, so the sentential form does not change and we shall keep going forever.

This particular example is easy to spot, as it is *directly* cyclic, but sometimes cycles arise within a chain of productions.

Although they occasionally appear in programming language standards, cyclic grammars are best thought of as laboratory curiosities to be avoided in practical applications of CFGs. ART will happily handle cycles, returning an encoding of the derivations which uses a loop to show where the cycles are, but derivation selection is challenging in the presence of cycles.

5.3.12 A useful letter generator

We round off this section with a useful (maybe) specification that generates thank you letters:

```
letter ::= salutation person ',' 'thank' 'you' 'for' 'my' gift '.' 'It' 'is' reaction '.' 'I' 'will' action 'it' '.'
gift ::= occasion 'present'
occasion ::= 'birthday' | 'house' 'warming'
salutation ::= 'Dearest' | 'Hey'
person ::= 'you' | 'Madam' | 'Sir'
reaction ::= 'incredible' | 'in' 'the' 'bin'
action ::= 'love' | 'hide' | 'try' 'to' 'forget'
```

ART tells us that this grammar has exactly 72 sentences. We could work that out without actually generating them by noting that

1. There are no recursive rules, so the language is finite.
2. Rules occasion, salutation and reaction have two alternate productions each
3. Rules person and action each have three alternates.
4. Rule gift has one instance of occasion which has two productions, so there only two ways to rewrite gift
5. The start rule has only one production which has one instance of each of salutation, person, gift, reaction and action. To get the total number

of sentences that can be generated from letter, we multiply together the number of sentences in each of their sublanguages to get

$$2 \times 3 \times 2 \times 2 \times 3 = 72$$

My favourite sentence is number 12:

```
"12 Dearest you, thank you for my house warming present. It is in the bin. I will try to forget it.
```

5.4 Parsing

Language generation is helpful way of understanding CFGs and can be useful if we want to construct a random set of test sentences for our language processors, but by far the most common application of CFGs is to *parsing*: given a CFG Γ and a putative sentence σ, a general parser should either return the full set of derivations if $\sigma \in L(\Gamma)$ or a helpful error message if $\sigma \notin L(\Gamma)$.

In ART, we activate the parser with a !try directive: for instance

```
S ::= 'a' R T
R ::= 'r'
T ::= 't'

!try "art"
!print outcome
!print derivation
!show derivation
```

which yields

```
Accept
current derivation term: [1480]
S(a, R(r), T(t))
```

The !try "art" directive sets the current input to art and runs the parser.

!print outcome will print `Accept` or `Reject` depending on whether the string can be generated by those rules, and !print derivation will print out the derivation term. The !show derivation directive produces a *visualisation* of the derivation tree, which maybe a pop up window in the IDE or a file `derivation.dot` which may be displayed using the GraphViz tools.

If we ask ART to parse a string that is not in the language of the rules, we get an error message:

```
!try "ara"
!print outcome
!print derivation
```

yields

```
1,2 GLLBL syntax error
    1: ara
---------^
Reject
current derivation term: [0]
null term
```

The message `1,2 GLLBL syntax error` means that ART was unable to find a derivation, and the furthest it got whilst parsing was line 1,column 2. ART will then print out the line in question with a pointer to the column position. The string is rejected, and the resulting derivation term is the null term.

We shall have much more to say about parsers and parsing later, but for now all we need to know is that a !try directive will parse an input string if it can, and then automatically pass the resulting term on to an interpreter if interpreter rules have been defined.

5.5 Derivation selection using choosers

Not in 2025 curriculum

5.6 GIFT annotations

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

- the suppression of recursion-scaffolding nodes,
- the construction of expression trees made up solely of nodes labeled with terminals,
- the renaming of nonterminals by production-specific labels
- the suppression of entire sub-trees,
- the local reordering of sub-trees,
- the insertion of new pieces of tree.

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

GIFT stands for Gather-Insert-Fold-Tear. The ART tool presently only implements the two Fold operations, but we shall discuss applications of the other operators. Collectively, the GIFT operations may be thought of as abbreviations for certain tree-rewrite operations. The best way to think about the GIFT operators is that they are annotations that are loaded into the derivation tree, and that a GIFT rewriting phase then rewrites the derivation tree under the control of those operators into a Rewritten Derivation Tree (RDT).

5.6.1 Fold operators

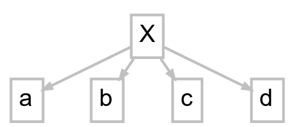
The fold operators can only be applied to a node which has a parent: that is the root node may not be folded.

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

This script

```
X ::= 'a' 'b' 'c' 'd'
!try "abcd"
```

yields this derivation tree

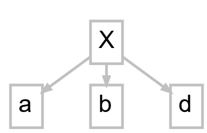


The idea of the fold operators is that the edge joining the annotated node to its parent is folded in half so that the child node and the parent node are coincident. If we fold under (ˆ) then the child goes under the parent; if we fold over then the child goes over the parent. Alternatively, you can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

For fold-under, then, we have

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

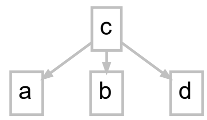
yields



Fold-over replaces the parent with the folded node, so

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

gives



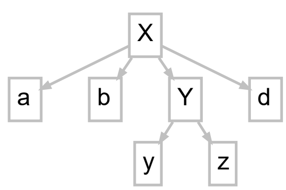
Note that this allows us to build trees which have *terminals* as internal nodes.

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

Consider these rules:

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

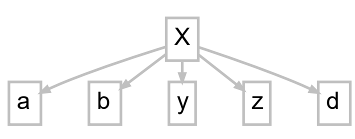
The sole generated string is `abyzd` with this derivation tree:



Annotating the Y node, for fold-under we have

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

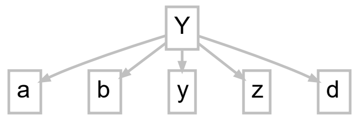
which yields



For fold-over we have

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

which yields



5.7 Context Free Grammar rules for expressions

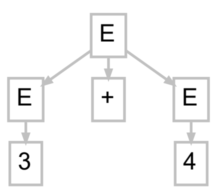
Expressions over diadic (arity-2) operators have the form

$$v_1 o_1 v_2 o_2 \ldots o_n v_{n+1}$$

where the v_i are values (or parenthesized subexpressions) and the o_i are operators symbols.

This grammar certainly accepts such strings, and works well for $n = 1$

```
E ::= &INTEGER
E ::= '(' E ')'
E ::= E '+' E
E ::= E '*' E
!try "3 + 4"
```



However, this grammar is *highlyambiguous* and will generate lots of different derivations for more complex inputs. In practice, we want one particular derivation which reflects the priorities and associativities for our language.

We can encode operator priority and associativity using these informal rules.

(i) have one nonterminal for each priority level.

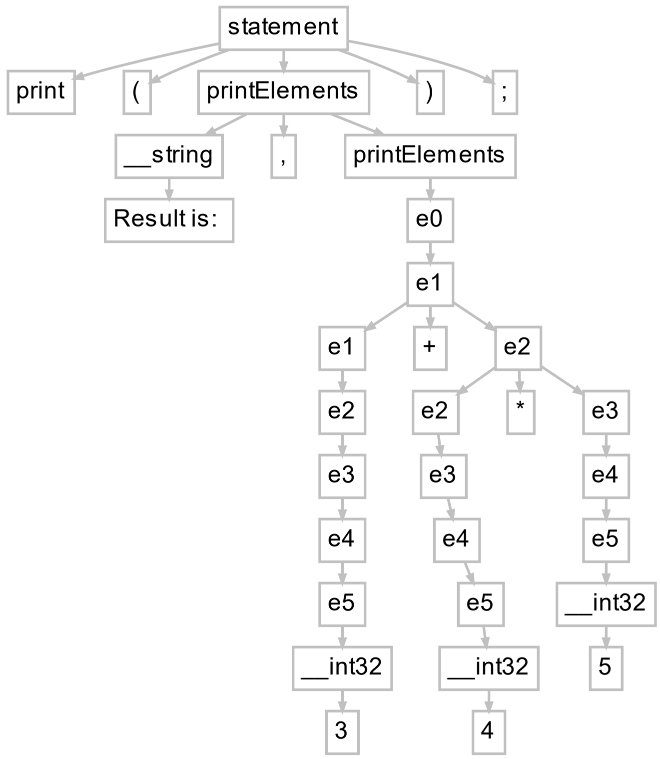
(ii) list these nonterminals in ascending order of priority, linking in a loop via a parenthesized call to the lowest priorityrule

(iii) For left associative operators, write left recursive rules; for right associative operators write right recursive rules; for non-associative operators (such as relationals) write non-recursive rules.

Here is an example for the common integer arithmetic operations:

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

printElements ::= __string |
                  __string ',' printElements |
                  e0 | e0 ',' printElements

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

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

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

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

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

e5 ::= __int32 | (* Integer literal *)
        '(' e1 ')' (* Parenthesised expression *)

__string ::= &STRING_SQ
__int32 ::= &INTEGER

!try "print('Result is: ', 3+4*5);"
```

The input string 3+4+5 yields this rather large, but anambiguous, derivation:

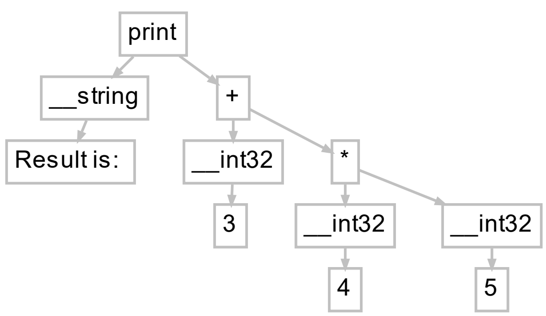


5.7.1 Using GIFT operators to tame expression trees

Careful use of the promotion operators will do away with the many redundant unit rule steps, and also remove the parethesis and semicolon symbols:

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

printElements ::= __string |
                  __string ','^ printElements^ |
                  e0 | e0 ','^ printElements^

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

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

```

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

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

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

e5 ::= __int32^^ | (* Integer literal *)
          '(' e1^^ ')' (* Parenthesised expression *)

__string ::= &STRING_SQ

__int32 ::= &INTEGER

!try "print('Result is: ', 3+4*5);"
```



5.7.2 Replacing operator symbols by alphanumeric identifiers

We are almost finished, but in a reduction semantics, we prefer to use words rather than symbols for our operations and types.

For nonterminals with a single production this is straightforward - we simply used the desired word as then nonterminal and we have already been handled above.

For instance, a 32-bit integers and 64-bit reals can be specified as

```
__int32 ::= &INTEGER
__real64 ::= &REAL
```

The elements beginning with a & are called *builtins*. They are pre-packaged recognisers for common idioms, such as string and numeric literals. You will find a full list of the available builtins in section **??**.

For rules like

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

we need to work a little harder by defining *unit rules* that effectively rename our symbolic operators to their alphanumeric equivalents and delete away the symbolic terminal:

```
e1 ::= e2^^ |
         e1 add^^ e2 | (* Add *)
         e1 'sub^^ e2 ; (* Subtract *)

add ::= '+'^
sub ::= '-'^
```

5.7.3 Renaming

There is one further (very useful) trick that we can use when working with GIFT operators: the *unit-epsilon* rule, that is a nonterminal whose only expansion is the empty string #

Now, adding an instance of such a rule can never change the language generated by the grammar; but we can use it to change the name of a grammar node:

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

add ::= #^
sub ::= #^
```

Chapter 6

Attribute interpreters

6.1 Inline actions and attributes

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

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

attributes

actions

The general ideas behind attribute evaluators can be seen in several early compilers which implement semantics by traversing derivations and accumulating information in an *ad hoc* way. in 1968, Donald Knuth described a formalisation of this approach called an *Attribute Grammar (AG)* [Knu68, Knu71, Knu90]. The values of attributes in an Attribute Grammar are specified using a purely equational approach, and the equations for attributes at a particular derivation tree node must only depend on other attributes which are locally visible. For a subtree comprising some parent node and its children, attributes of the parent node whose equations depend only on attributes of the child nodes are called *synthesized* attributes. Attributes of child nodes which are updated by equations are called *inherited* attributes.

Attribute Grammar (AG)

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

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

Attribute-Action Grammar (AAG)

6.2 Derivation traversers

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

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

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

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

6.3 Attribute Grammars

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

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

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

6.3.1 Formal definitions

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

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

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

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

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

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

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

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

6.3.2 Attribute grammars in practice

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

Consider a rule of the form

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

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

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

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

6.3.3 Attribute grammar subclasses

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

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

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

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

6.4 RDSOB: generating parsers using a weak algorithm

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

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

Parser generators, then, can be thought of as processors for a DSL (BNF) which translates to, say, Java. Embedded within each parser will be a *parsing algorithm*. We shall illustrate this process using a parsing algorithm called Recursive Descent with Singleton Ordered Backtracking (RDSOB). It is a rather limited algorithm: it's advantage is that it is easy to understand and easy to generate. This means that it is possible to fully explain the internals not just of the parsers but of parser generators: that is programs that write out the parsers.

6.5 Recursive Descent with Singleton Ordered Backtrack

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

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

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

6.5.1 The RDSOB algorithm

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

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

This seemingly innocuous change has a big impact on the languages that can be successfully parsed by RDSOB compared to a truly general technique such as GLL. We are highlighting this difference here because occasionally one encounters parsing tools which use algorithms based on ordered grammars, and in my experience the authors often do not adequately explain the limitations of the technique.

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

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

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

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

6.5.2 An RDSOB example in Java

Here is a small grammar.

```
S ::= 'b' | 'a' X '@'
X ::= 'x' X | #
!generate rdsobOracle
```

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

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

```
boolean parse_S() {
  int iiAtEntry = inputIndex, oiAtEntry = oracleIndex;

  /* Nonterminal S, alternate 1 */
  inputIndex = iiAtEntry; oracleIndex = oiAtEntry; oracleSet(1);
  if (match("b")) { return true; }

  /* Nonterminal S, alternate 2 */
  inputIndex = iiAtEntry; oracleIndex = oiAtEntry; oracleSet(2);
  if (match("a")) {
  if (parse_X()) {
  if (match("@")) { return true; }}}

  return false;
}

boolean parse_X() {
  int iiAtEntry = inputIndex, oiAtEntry = oracleIndex;

  /* Nonterminal X, alternate 1 */
  inputIndex = iiAtEntry; oracleIndex = oiAtEntry; oracleSet(1);
  if (match("x")) {
```

```
  if (parse_X()) { return true; }}

  /* Nonterminal X, alternate 2 */
  inputIndex = iiAtEntry; oracleIndex = oiAtEntry; oracleSet(2);
  /* epsilon */ return true;
}
```

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

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

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

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

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

The initial call to `S()` with `cc=0` tries alternates 1 and 2, before a calling `X()` which tries alternate 1 and then calls `X()` again which itself then calls `X()`. Alternate 1 cannot match `@` to `x`, so alternate 2 is tried instead which must succeed because it is an ϵ-production.

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

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

6.6 Engineering a complete Java parser

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

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

1. uk.ac.rhul.cs.csle.art.cfg.rdsob.RDSOBAbstract which contains auxiliary meth-

ods.

2. ARTGeneratedRDSOBOracle which extends class ARTRDSOBAbstract with the parse functions and a main() function which processes command line arguments.

Here is the contents of ARTGeneratedRDSOBOracle.java for our example grammar — in later sections we shall add more functions to support semantics processing and tree construction.

```
import java.io.FileNotFoundException;import java.util.LinkedList;

class ARTGeneratedRDSOBOracle extends uk.ac.rhul.cs.csle.art.cfg.rdsob.RDSOBAbstract {

boolean parse_S() {
  int iiAtEntry = inputIndex, oiAtEntry = oracleIndex;

  /* Nonterminal S, alternate 1 */
  inputIndex = iiAtEntry; oracleIndex = oiAtEntry; oracleSet(1);
  if (match("b")) { return true; }

  /* Nonterminal S, alternate 2 */
  inputIndex = iiAtEntry; oracleIndex = oiAtEntry; oracleSet(2);
  if (match("a")) {
  if (parse_X()) {
  if (match("@")) { return true; }}}

  return false;
}

boolean parse_X() {
  int iiAtEntry = inputIndex, oiAtEntry = oracleIndex;

  /* Nonterminal X, alternate 1 */
  inputIndex = iiAtEntry; oracleIndex = oiAtEntry; oracleSet(1);
  if (match("x")) {
  if (parse_X()) { return true; }}

  /* Nonterminal X, alternate 2 */
  inputIndex = iiAtEntry; oracleIndex = oiAtEntry; oracleSet(2);
  /* epsilon */ return true;
}

void parse(String filename) throws FileNotFoundException {
  input = readInput(filename);

  System.out.println("Input: " + input); inputIndex = oracleIndex = 0; builtIn_WHITESPACE();
```

```
    if (!(parse_S() && input.charAt(inputIndex) == '\0')) { System.out.println("Reject"); return; }

    System.out.println("Accept");
    System.out.print("Oracle:");
    for (int i = 0; i < oracleIndex; i++)
      System.out.print(" " + oracle[i]);
      System.out.println();
  }

public static void main(String[] args) throws FileNotFoundException{
      if (args.length < 1)
        System.out.println("Usage: java ARTGeneratedRDSOBOracle <input file name>");
      else new ARTGeneratedRDSOBOracle().parse(args[0]);

    }
}
```

The constructor sets the initial oracle length 1000, creates the array and zeroes the inputIndex and oracleIndex variables. Loading the input string is the responsibility of the parse() method;it makes use of function readInput() which automatically appends an end-of-string marker `'\0'`.

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

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

The main() function simply instantiates the enclosing class, passing the first argument as an input filename.

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

Here is the source of the support methods and variables in RDSOBAbstract which is the superclass for the generated parser classes.

```
package uk.ac.rhul.cs.csle.art.cfg.rdsob;

import java.io.File;
import java.io.FileNotFoundException;
import java.util.Arrays;
import java.util.Scanner;
```

```

public class RDSOBAbstract {
  protected String input;
  protected int inputIndex;
  protected int oracleIndex;
  protected int lexemeL;
  protected int lexemeR;
  protected int oracle[];

  protected RDSOBAbstract() {
    oracle = new int[1000];
    inputIndex = oracleIndex = 0;
  }

  protected String lexeme() {
    return input.substring(lexemeL, lexemeR);
  }

  protected String readInput(String filename) throws FileNotFoundException {
    return new Scanner(new File(filename)).useDelimiter("\\Z").next() + "\0";
  }

  protected void oracleSet(int i) {
    if (oracleIndex == oracle.length) oracle = Arrays.copyOf(oracle, oracle.length + oracle.length / 2);
    oracle[oracleIndex++] = i;
  }

  protected boolean match(String s) {
    if (input.regionMatches(inputIndex, s, 0, s.length())) {
      inputIndex += s.length();
      builtIn_WHITESPACE();
      return true;
    }
    return false;
  }
}
```

6.7 Builtin recognisers

The RDSOBAbstract base class also provides a set of built in matchers which can be used to efficiently parse identifiers, numeric literals, strings and so on.

The parser generator translates identifiers preceded by an ampersand & such as &ID into calls to the corresponding built in matcher. In detail, &XYZ is translated to a call to builtin_XYZ(). The generator does not check the name of the built in, so just by adding a new built in member function to class uk.ac.rhul.cs.csle.artRDSOB.ARTRDSOBAbstract we can extend the repertoire of builtin matchers.

Here is a slightly modified version of our test grammar.

```
S ::= 'b' | 'a' X '@'
X ::= &ID X | #
!generate rdsobOracle
```

It generates this parse function for X

```
boolean parse_X() {
  int iiAtEntry = inputIndex, oiAtEntry = oracleIndex;

  /* Nonterminal X, alternate 1 */
  inputIndex = iiAtEntry; oracleIndex = oiAtEntry; oracleSet(1);
  if (builtIn_ID()) {
  if (parse_X()) { return true; }}

  /* Nonterminal X, alternate 2 */
  inputIndex = iiAtEntry; oracleIndex = oiAtEntry; oracleSet(2);
  /* epsilon */ return true;
}
```

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

With this modification, our grammar can now accept strings of Java-style identifiers. For instance, input `a name1 name2 @` yields

```
Input: a name1 name2 @
Accepted
Oracle: 2 1 1 2
```

6.7.1 Terminals, patterns, lexemes and tokens

Now that we have CFG rules containing builtins, we need to define some new technical terms.

pattern

Both 'x' and &ID specify terminals in our CFG rules. Each terminal has a *pattern* which is the (possibly infinite) set of strings that the terminal matches. For stropped literal terminals like x the pattern will have exactly one string in it made of the characters between the strops. We sometimes call these kinds of terminals *singleton pattern* terminals.

The builtins are as a shorthand for commonly occurring elements of programming languages such as numeric literals, strings and identifiers. We *could* write CFG rules that specify the pattern of an identifier. When writing translators, it is not enough to know that, say, &ID has matched an identifier: we shall want to know which particular identifier was matched. The substring of the input matched by a terminal is called the *lexeme* and will of course be one element drawn from the pattern.

lexeme

In our production parsers based on the GLL algorithm, we think of there being a separate phase called *lexicalisation* which segments the input string into a string of lexemes, and then replaces each lexeme with a *token* which is simply a natural number which uniquely identifies the token. That way the parser only 'sees' a string of integers, and does not have to continually re-read the input string,which can significantly improve performance. The component that performs this segmentation is called a *lexer*.

lexicalisation

token

lexer

In RDSOB we do not have a separate lexer: our match and builtin_xyz() methods reread the string each time they are called, and with backtracking that means that each part of the string may be read many times. We can tolerate this because RDSOB is just a pedgogic tool, and is not intended to be used for production tasks.

6.7.2 Source code for builtins

Each builtin has an associated *recogniser* which is simple piece of code which attempts to advance inputIndex over a substring that conforms to the builtin's pattern, returning true if a match is found, and otherwise false.

The start and end indices of the matched substring is remembered in global variables lexemeL and lexemeR. There is a method lexeme() which will extract the corresponding substring from the input: this will be important when we add semantics processing to the RDSOB parser in the next section.

Note how each of them calls builtin_WHITESPACE() after matching so to silently consume white space between lexemes.

Therest of this section shows the complete source code for the builtin recognisers.

```
protected boolean builtIn_ID() {
  if (!Character.isJavaIdentifierStart(input.charAt(inputIndex))) return false;
  lexemeL = inputIndex++;
  while (Character.isJavaIdentifierPart(input.charAt(inputIndex)))
    inputIndex++;
  lexemeR = inputIndex;
  builtIn_WHITESPACE();
  return true;
}

protected boolean builtIn_WHITESPACE() {
```

```
    while (Character.isWhitespace(input.charAt(inputIndex)))
      inputIndex++;
    return true;
  }

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

  protected boolean builtIn_INTEGER() {
    if (!Character.isDigit(input.charAt(inputIndex))) return false;
    lexemeL = inputIndex;

    boolean hex = (input.charAt(inputIndex) == '0' && (input.charAt(inputIndex + 1) == 'x' ||
                   input.charAt(inputIndex + 1) == 'X'));
    if (hex) inputIndex += 2; // Skip over hex introducer

    while (hex ? isxdigit(input.charAt(inputIndex)) : Character.isDigit(input.charAt(inputIndex)))
      inputIndex++;
    lexemeR = inputIndex;
    builtIn_WHITESPACE();
    return true;
  }

  protected boolean builtIn_REAL() {
    if (!Character.isDigit(input.charAt(inputIndex))) return false;
    lexemeL = inputIndex;
    while (Character.isDigit(input.charAt(inputIndex)))
      inputIndex++;
    if (input.charAt(inputIndex) != '.') return true;
    inputIndex++; // skip over the .
    while (Character.isDigit(input.charAt(inputIndex)))
      inputIndex++;
    if (input.charAt(inputIndex) == 'e' || input.charAt(inputIndex) == 'E') {
      inputIndex++;
      while (Character.isDigit(input.charAt(inputIndex)))
        inputIndex++;
    }
    lexemeR = inputIndex;
    builtIn_WHITESPACE();
    return true;
  }

  protected boolean builtIn_CHAR_SQ() {
```

```
    if (input.charAt(inputIndex) != '\'') return false;
    inputIndex++; // skip over the '
    lexemeL = inputIndex;
    if (input.charAt(inputIndex) == '\\') inputIndex++;
    inputIndex++;
    if (input.charAt(inputIndex) != '\'') return false;
    lexemeR = inputIndex;
    inputIndex++; // skip past final delimiter
    builtIn_WHITESPACE();
    return true;
  }

  protected boolean builtIn_STRING_SQ() {
    if (input.charAt(inputIndex) != '\'') return false;
    lexemeL = inputIndex + 1;
    do {
      if (input.charAt(inputIndex) == '\\') inputIndex++;
      inputIndex++;
    } while (input.charAt(inputIndex) != '\'');
    lexemeR = inputIndex;
    inputIndex++; // skip past final delimiter
    builtIn_WHITESPACE();
    return true;
  }

  protected boolean builtIn_STRING_DQ() {
    if (input.charAt(inputIndex) != '"') return false;
    lexemeL = inputIndex + 1;
    do {
      if (input.charAt(inputIndex) == '\\') inputIndex++;
      inputIndex++;
    } while (input.charAt(inputIndex) != '"');
    lexemeR = inputIndex;
    inputIndex++; // skip past final delimiter
    builtIn_WHITESPACE();
    return true;
  }

  protected boolean builtIn_ACTION() {
    if (!(input.charAt(inputIndex) == '!' && input.charAt(inputIndex + 1) == '!')) return false;
    inputIndex += 2;
    lexemeL = inputIndex;
    while (true) {
      if (input.charAt(inputIndex) == 0) break;
      if (input.charAt(inputIndex) == '!' && input.charAt(inputIndex) == '!') {
        inputIndex += 2;
        break;
```

```
      }
      inputIndex++;
    }
    lexemeR = inputIndex - 2;
    builtIn_WHITESPACE();
    return true;
```

6.8 Using inline native actions

native action

A *native action* is delimited by !! ... !! brackets and must be written in the implementation language for the generated parser (in our case Java). ART itself has no understanding of what is between the !! brackets and simply copies the text into the generated methods. This can be quite tiresome since syntax errors within the actions are only revealed when the generated Java methods are compiled.

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

```
S ::= 'b' | 'a' X '@'
         X ::= 'x' !! System.out.println("Matched an x at location " + inputIndex); !! X | #
!generate rdsobOracle
```

The generated parser running on the string `axx@` yields:

```
Input: axx@
Accepted
Oracle: 2 1 1 2
Semantics:
Matched an x at location 2
Matched an x at location 3
```

The parse is as before. After parsing is completed, a second pass is made during which the semantics are executed. We shall look at the implementation details below.

6.8.1 Accessing the lexeme matched by a builtin

The terminal 'x' can, of course, only match one thing: an `x` on the input. Our builtins, though, can match many different things. Recall that the recognisers keep track of the most recent lexeme in global variables. The abstract superclass contains a method lexeme() which return the most recent substring matched by a builtin — it does not remember whitespace or singleton terminal lexemes. This enables us to print out the specific identifier matched by &ID.

```
S ::= 'b' | 'a' X '@'
X ::= &ID !! System.out.println("Matched " + lexeme() + " at location " + inputIndex); !! X | #
!generate rdsobOracle
```

The generated parser running on the string `axx@` yields:

```
Input:  a name1 name2 @
Accepted
Oracle: 2 1 1 2
Semantics:
Matched name1 at location 9
Matched name2 at location 15
```

6.8.2 Handling numeric literals

The lexeme() method returns a substring. We can use methods from the standard Java API to convert these strings to numbers, and then use those numbers in actions.

In this specification, the CFG rule for nonterminal X has been changed to match a single &INTEGER followed by a single ID. The rule then prints out the matched identifier as many times as the matched number specifies.

```
1 S ::= 'b' | 'a' X '@'
2
3 X ::=
4   &INTEGER !! int count = Integer.parseInt(lexeme()); !!
5   &ID !! String id = lexeme(); !!
6
7   !! for (int i = 0; i < count; i++) System.out.println(i + " " + id); !!
8
9 !generate rdsobOracle
```

At line 4, the lexeme from the &INTEGER match is converted to a numeric value and assigned to count. At line 7 we have an action which uses a for loop to print the &ID lexeme outcount times.

When run on the input `a 3 name @`, this translator yields

```
Input: a 3 name @
Accepted
Oracle: 2 1
Semantics:
0 name
1 name
2 name
```

Similarly, we can use Double.parseDouble to convert a lexeme matched by the &REAL builtin:

```
X ::= &REAL !! double count = Double.parseDouble(lexeme()); !!
```

6.9 Attributes and instances

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

In ART, user attributes are specified on the left hand side of a CFG rule. For instance

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

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

Now, a rule might have multiple instances of a nonterminal. Within the actions, we need to be able to specify which one we want and we do that by numbering them from the left.

In the specification below, we have a rule A with an attribute v of type int. This rule can match either the letter a, in which case the attribute is set to 1, or the letter b in which case the attribute is set to 2.

Rule X calls A twice, and adds together the resulting synthesized attribute values. The two instances are callled A1 and A2.

```
X ::= A A !! System.out.println(A1.v + A2.v); !!

A<v:int> ::=
  'a' !! A.v = 1; !!
| 'b' !! A.v = 2; !!

!generate rdsobOracle
```

6.9.1 Counting

Here is an example that uses attributes to add up the number of 1's seen in a binary string, and then pass that result back to the start symbol S.

```
S ::= 'b' | 'a' X !! System.out.println("Result is: " + X1.v); !! '@'
```

```

X<v:int> ::=
   '1' X !! X.v = 1 + X1.v; !!
| '0' X !! X.v = X1.v; !!
| !! X.v = 0; !! #

!generate rdsobOracle
```

The language of this grammar is

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

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

The result of running this parser on the string `a1110011@` is

```
Input: a1110011@
Accepted
Oracle: 2 1 1 1 2 2 1 1 3
Semantics:
Result is: 5
```

6.9.2 A four function calculator

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

```
(* A calculator that does not respect left-associative operators *)
S ::= e1 !! System.out.println("Result: " + e11.v); !!

e1<v:int> ::=
   e2 '+' e1 !! e1.v = e21.v + e11.v; !!
| e2 '-' e1 !! e1.v = e21.v - e11.v; !!
| e2 !! e1.v = e21.v; !!

e2<v:int> ::=
   e3 '*' e2 !! e2.v = e31.v * e21.v; !!
| e3 '/' e2 !! e2.v = e31.v / e21.v; !!
| e3 !! e2.v = e31.v; !!

```

```
e3<v:int> ::=
  '(' e1 ')' !! e3.v = e11.v; !!
| integer !! e3.v = integer1.v; !!

integer<v: int> ::=
  digit !! integer1.v = integer.v * 10 + digit1.v; !! integer !! integer.v = integer1.v; !!
| digit !! integer.v = integer.v * 10 + digit1.v; !!

digit<v:int> ::=
  '0' !! digit.v = 0; !!
| '1' !! digit.v = 1; !!
| '2' !! digit.v = 2; !!
| '3' !! digit.v = 3; !!
| '4' !! digit.v = 4; !!
| '5' !! digit.v = 5; !!
| '6' !! digit.v = 6; !!
| '7' !! digit.v = 7; !!
| '8' !! digit.v = 8; !!
| '9' !! digit.v = 9; !!
```

When the generated parser is run on the string `(3 + 4) * 5` we get the following output

```
Input: (3 + 4) * 5
Accepted
Oracle: 1 3 1 1 1 3 2 2 4 3 3 2 2 5 3 2 2 6
Semantics:
Result: 35
```

6.10 Implementing inline semantics

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

As a result of this retry behaviour, we cannot simply execute inline semantics during the parse, even though when we design grammars and their semantic actions we tend to think of the action being executed as a side-effect of parsing. Instead, we need to complete the searching associated with parsing and only then run through the grammar the 'correct' way to execute the actions. This is the purpose of the oracle: during parsing we construct the oracle as we go, adjusting it as necessary when we backtrack. By the end of the parse we have a map of where the parser *should* have gone. We call the control data structure an oracle because it is as if we had a parser which instead of guessing where to go could simply ask an all-powerful oracle for advice.

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

We also create an *attribute class* for each nonterminal that encapsulates the attributes that are declared for that nonterminal. These classes only include data elements and are called Attribute_xyz where xyz is the name of the nonterminal.

In a rule of the form

```
X ::= Y
```

The semantics function for X (called semantics_X()) initially creates an attribute block for each right hand side nonterminal instance. Actions in semantics_X() can then write values into those blocks, implementing inherited attributes.

The semantics function for Y takes an attribute block: semantics_Y(Attribute_Y Y). Each instance of Y is supplied with the appropriate block as it is called, and the actions in semantics_Y can then update the values in that attribute block. Since blocks are passed by reference, after semantics_Y returns, those changed values will be accessible in semantics_X, thus implementing synthesized attributes.

Here is a complete example. Recall the attributed grammar that adds up the 1's in a string:

```
S ::= 'b' | 'a' X !! System.out.println("Result is: " + X1.v); !! '@'
X<v:int> ::=
  '1' X !! X.v = 1 + X1.v; !!
| '0' X !! X.v = X1.v; !!
| !! X.v = 0; !! #
```

These are the semantics functions which are written out into the generated parser, along with the definition of class Attributes_X. There is no attribute class for nonterminal S because no attributes have been declared for S.

```
void semantics_S() {
  Attributes_X X1 = new Attributes_X();
  switch(oracle[oracleIndex++]) {
    case 1:
    match("b");
    break;

    case 2:
    match("a");
    semantics_X(X1);
    System.out.println("Result is: " + X1.v);
    match("@");
    break;
  }
}
```

```
class Attributes_X { int v; }

void semantics_X(Attributes_X X) {
  Attributes_X X1 = new Attributes_X();
  switch(oracle[oracleIndex++]) {
    case 1:
    match("1");
    semantics_X(X1);
    X.v = 1 + X1.v;
    break;

    case 2:
    match("0");
    semantics_X(X1);
    X.v = X1.v;
    break;

    case 3:
    X.v = 0;
    /* epsilon */
    break;
  }
 }
```

The control flow within semantics functions is *via* switch statements selecting on the current oracle index; as each element of the oracle is consumed, the index is incremented by one. The use of switch statements is faster then the sequential testing required in the parse functions, and more importantly the contents of the oracle tell the semantics functions exactly which production to use. There is no backtracking, and thus it is safe to execute the actions as they are encountered.

6.11 Attributes and actions within ART general parsers

The RDSOB algorithm is intended to be purely illustrative — we have used it to create an intellectual framework for understanding attribute evaluation, but the algorithm is too slow and too restricted for production use. It cannot handle left recursion — critically important when writing expression grammars — and it requires productions to be tested in reverse order of the matching substring. This second constraint is not always possible to achieve.

We switch now to using fully general parsers. The specification syntax does not change, but the directives needed to implement the specification do.

Firstly, we do not need to generate the parser source code because ART has built-in general parsers that directly access the internal form of the grammar.

However, we do still need to handle native Java-language actions. This is because the ART tool does not in any sense understand the Java language: the actions between the !! ... !! brackets are just treated as text strings. We need to write them out to a Java file, use the `javac` compiler to generate a class file and then load it on a subsequent ART run.

Essentially we are using the same mechanism as for ARTValuePlugin, except that instead of your writing your own plugin (and then compiling it and allowing ART to find it and attach to it) you are writing actions which ART writes out to a file called ARTGeneratedActions.java (which you then compile, and allow ART to find and attach to).

The actions generator is activated in response to this directive:

```
!generate actions
```

The generator will announce that a new ARTGeneratedActions.java has been written out, and give a timestamp which is also written into the Java file. This gives each generated set of actions a unique identity which can be helpful when debugging.

```
Writing new ARTGeneratedActions: 2025-03-17 14:54:07
```

As with ARTValuePlugin.java we need to now compile the generated actions, and make sure that the resulting ARTGeneratedActions.class file is located on the class path for the next run of ART.

ART supports a variety of interpretation mechanisms. By default, ART runs the eSOS reduction interpreter, but we now want to use the attribute-action interpreter. We specify that by issuing this directive *before* any !try directives that need attribute-action interpretation.

```
!interpreter attributeAction
```

When that directive is processed, ART looks for an ARTGeneratedActions.class file, and if it finds one then it will attach to it; announcing the timestamp as it does so.

```
*** Attached to ARTGeneratedActions 2025-03-17 14:54:07
```

On windows systems, this batch file is a convenient way to test attribute-action systems:

```
javac −cp .;art.jar ARTValuePlugin.java
java −cp .;art.jar uk.ac.rhul.cs.csle.art.ART %1 !generate actions
javac −cp .;art.jar ARTGeneratedActions.java
java −cp .;art.jar uk.ac.rhul.cs.csle.art.ART !interpreter attributeAction %1
```

Line 1 compiles your ARTValuePlugin. (ifyouhave one); line 2 runs ART to write out the generated action file ARTGeneratedActions.java; line 3 compiles the generated action file and line 4 runs ART again with the attribute-action interpreter which picks up the compiled ARTGeneratedActions.class.

A Un*x equivalent is below. Note the use of $1 instead of %1 as command line argument place holders and the use of : instead of ; as directory separators in the class path arguments.

```
#!/bin/sh
javac -cp .:art.jar ARTValuePlugin.java
java -cp .:art.jar uk.ac.rhul.cs.csle.art.ART $1 !generate actions
javac -cp .:art.jar ARTGeneratedActions.java
java -cp .:art.jar uk.ac.rhul.cs.csle.art.ART !interpreter attributeAction $1
```

6.12 Control flow in attribute-action interpreters

Data flowing up and down trees enables us to implement calculations and even conditional IF-THEN-ELSE if there are no side-effects: we could evaluate both the THEN and the ELSE branches and throw away the result we do not need. However, in the presence of side effects such an approach would not be safe, as any side effects from the discarded branch would persist.

More significantly, it is unclear how to handle loops and procedure call just with value attributes. What we need is a way of allowing actions to directly manipulate the interpretation order.

Now, the attribute-action interpreter works the same general way as the RDSOB semantics actions: we call the interpreter in the start symbol, and then the derivation (in the form of the oracle) guides us as to which other nonterminals need to be called. This all happens automatically. However, if we could tell the interpeter *not* to call a particular nonterminal, and instead allow user actions to initiate the call then we could choose which of the THEN and ELSE branches to call, and completely ignore the other one, ensuring that it's side effects never occur. We could even use the same mechanism to *repeatedly* call a nonterminal, thus implementing loops. Finally, if we have some way of storing part of a derivation in variable, we could use that to implement procedure call. Attibutes which store references to part of the derivation are called *higher order* attributes, and they give access to the internals of the derivation in the same way that Java's reflection capabilities give access to Java's runtime internals.

6.12.1 Delayed attributes

We distinguish two classes of higher order attributes: attributes which capture part of an existing derivation tree, and attributes which contain new pieces of tree which can be used to extend a derivation tree from the parser. In ART, we support the former, but not the latter. This means that the shape and labelling of a derivation tree in ART *cannot* be modified by an attribute grammar: only the attribute values associated with tree nodes can be modified.

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

We achieve (a) by adding a *delay* annotation !< to any right hand side instance of a nonterminal in a grammar rule. The generator adds a field called term to the attribute block for the delayed nonterminal which holds the subtree term for that instance.

For (b), we provide a method interpret(X) which takes an attribute block as an argument, and creates a recursive instance of the attribute-action interpreter acting on the subtree term held in that block's term field. This is reminiscent of the way that a transition above the line in a reduction rules triggers a recursive instance of the eSOS interpreter.

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

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

```
ifStatement ::= 'if' e0 'then' statement!<
          !! if (e01.v != 0) interpret(statement1); !!
```

The evaluator will automatically descend into the subtree for e0 but will not descend into the subtree for statement.

In the action, we look at the result that was computed within e01, and if it is not zero (signifying false) we call the evaluator on the the subtree root node held in the term attribute for statement1. This effectively emulates what would have happened automatically if we had left off the !< annotation, but under the control of the result of e01.v. Hence the evaluation order of the tree is being dictated by the attributes and semantic actions themselves! This is exactly the sense in which our attributes are higher order.

However, we can only traverse bits of tree that were built by the parser: we cannot make new tree elements and call the evaluator on them. Fully general higher order attributes do allow that. We call our restricted form *delayed* attributes so as to distinguish them from the more general technique.

We can use these delayed attributes to build interpreters for languages with conditionals, loops and function calls.

6.13 The `minicall` language

We round off this discussion of attribute-action interpreters with a small specification which shows how to implement an integer-only arithmetic language with conditional, loop and procedure call features. The language is called *Mini*. This particular variant is called **`minicall.art`**, and the full specification may be retrieved from

https://github.com/AJohnstone2007/ART/blob/main/doc/examples/mini/minicall.art.

Here is a slimmed-down version called minicallNoActions.art which contains just the CFG rules with no attributes or actions, so as to show the syntax of the language.

```
text ::= statements
statements ::= statement | statement statements

statement ::=
   ID '=' e0 ';'
| 'if' e0 'then' statement elseOpt
| 'while' e0 'do' statement
| 'print' '(' printElements ')' ';'
| 'procedure' ID statement!<
| 'call' ID ';'
| '{' statements '}'

elseOpt ::= 'else' statement | #

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

e0 ::= e1 | e1 '>' e1 | e1 '<' e1 | e1 '>=' e1 | e1 '<=' e1 | e1 '==' e1 | e1 '!=' e1
e1 ::= e2 | e1 '+' e2 | e1 '-' e2
e2 ::= e3 | e2 '*' e3 | e2 '/' e3 | e2 '%' e3
e3 ::= e4 | '+' e3 | '-' e3
e4 ::= e5 | e5 '**' e4
e5 ::= INTEGER | ID | '(' e1 ')'

ID ::= &ID
STRING_SQ ::= &STRING_SQ
INTEGER ::= &INTEGER
```

The expression grammar used the conventional mechanisms for ensuring priority and associativity of the operators. Statements are terminated by ; and procedure call is triggered with the call keyword. Procedures do not have parameters.

Here is a small Mini program embedded in a !try

```
!try "
procedure sub { print('Hello from a procedure\n'); }

x = 3;
while x > 0 do { print('x is ', x, '\n'); x = x -1; }

call sub;
"
```

Our hoped for output from this program is:

```
Accept
x is 3
x is 2
x is 1
Hello from a procedure
Final variable map {x=0}
```

This full version of `minicall.art` uses exactly the same CFG rules, but decorated with attribute declarations and actions. It also includes delay annotations on lines 12, 16 and 25 which handle IF-THEN=ELSE, WHILE-DO and PROCEDURE respectively.

```
1  !support
2  !! import java.util.Map; import java.util.HashMap; !!
3  !! Map<String, Integer> variables = new HashMap<>();
4      Map<String, AbstractAttributeBlock> procedures = new HashMap<>(); !!
5  
6  text ::= statements !! System.out.println("Final variable map " + variables); !!
7  statements ::= statement | statement statements
8  
9  statement ::=
10    ID '=' e0 ';' !! variables.put(ID1.v, e01.v); !! (* assignment *)
11 
12 | 'if' e0 'then' statement!< elseOpt!< (* if statement *)
13    !! if (e01.v != 0) interpret(statement1);
14         else interpret(elseOpt1); !!
15 
16 | 'while' e0!< 'do' statement!< (* while statement *)
17    !! interpret(e01);
18         while (e01.v != 0) {
19             interpret(statement1);
```

```
            interpret(e01);
          } !!

| 'print' '(' printElements ')' ';' (* print statement *)

| 'procedure' ID statement!< !! procedures.put(ID1.v, statement1); !!
| 'call' ID ';' !! interpret(procedures.get(ID1.v)); !!

| '{' statements '}' (* compound statement *)

elseOpt ::= 'else' statement | #

printElements ::=
  STRING_SQ !! System.out.print(STRING_SQ1.v); !!
| STRING_SQ !! System.out.print(STRING_SQ1.v); !! ',' printElements
| e0 !! System.out.print(e01.v); !!
| e0 !! System.out.print(e01.v); !! ',' printElements

e0<v:int> ::=
  e1 !! e0.v = e11.v; !!
| e1 '>' e1 !! e0.v = e11.v > e12.v ? 1 : 0; !! (* Greater than *)
| e1 '<' e1 !! e0.v = e11.v < e12.v ? 1 : 0; !! (* Less than *)
| e1 '>=' e1 !! e0.v = e11.v >= e12.v ? 1 : 0; !! (* Greater than or equals*)
| e1 '<=' e1 !! e0.v = e11.v <= e12.v ? 1 : 0; !! (* Less than or equals *)
| e1 '==' e1 !! e0.v = e11.v == e12.v ? 1 : 0; !! (* Equal to *)
| e1 '!=' e1 !! e0.v = e11.v != e12.v ? 1 : 0; !! (* Not equal to *)

e1<v:int> ::=
  e2 !! e1.v = e21.v; !!
| e1 '+' e2 !! e1.v = e11.v + e21.v; !! (* Add *)
| e1 '-' e2 !! e1.v = e11.v - e21.v; !! (* Subtract *)

e2<v:int> ::=
  e3 !! e2.v= e31.v; !!
| e2 '*' e3 !! e2.v = e21.v * e31.v; !! (* Multiply *)
| e2 '/' e3 !! e2.v = e21.v / e31.v; !! (* Divide *)
| e2 '%' e3 !! e2.v = e21.v % e31.v; !! (* Mod *)

e3<v:int> ::=
  e4 !!e3.v = e41.v; !!
| '+' e3 !!e3.v = e31.v; !! (* Posite *)
| '-' e3 !!e3.v = -e31.v; !! (* Negate *)

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

```
67 e5<v:int> ::=
68   INTEGER !!e5.v = INTEGER1.v; !! (* Integer literal *)
69 | ID !! e5.v = variables.get(ID1.v); !! (* Variable access *)
70 | '(' e1 !! e5.v = e11.v; !! ')' (* Parenthesised expression *)
71
72 (* Lexical productions *)
73 ID <v:String> ::= &ID !! ID.v = lexeme(); !!
74 STRING_SQ <v:String> ::= &STRING_SQ !! STRING_SQ.v = lexemeCore().translateEscapes(); !!
75 INTEGER<v:int> ::= &INTEGER !! INTEGER.v = Integer.parseInt(lexeme()); !!
```

6.13.1 The `!support` directive

The type of an ART attribute can be any Java class, but as is usual with Java programs we shall need to import classes that are not part of the system package, so we need a way to add import statements to the ARTGeneratedActions.java file. We often also want to add methods and data fields to the top level class in ARTGeneratedActions.java so that they are globally available within actions.

The !support directive takes two actions: the first is written into the start of ARTGeneratedActions.java and thus can be used for import statements, and the second is written into the start of the top level class, and can be used to declare global data structures and supporting methods.

Lines 1–4 below uses the !support directive to create two HashMaps, one of which holds variables which are updated via the assignment action at line 10, and the other of which holds procedure names created using the action at line 25. The use of separate tables means that variable names and procedure names are in separate name spaces, and it also handles the typing issues elegantly: variables hold integer results whereas procedures are represented by their associated attribute block.

6.13.2 Program variables

We distinguish three kinds of actions associated with program variables: *declaration*, *definition* and *usage*.

A declaration reserves storage for a variable and gives that space a name. A definition binds that name to a value by loading the value into the storage created by the declaration. A usage extracts the value from the storage associated with variable name and returns it for use in an expression.

In the Mini language, there is no separate declaration and an assignment handles both declaration and definition. At line 10 we match an identifier and an expression and then load them as a key-value pair into the global variables map. Variable usage is specified at line 69: when an identifier appears in an expression context, its value is retrieved from the variables map for use in the enclosing expression.

6.13.3 Control flow

An IF-THEN-ELSE statement need only evaluate its predicate once, so at lines 12–14 we can use the 'automatic' evaluation and do not need to delay the predicate nonterminal. During execution, either branch could be taken, so we delay both branches

The WHILE-DO statememt at lines 16–21 has a similar form to IF-THEN but its evaluation is more complex because we need to re-evaluate the predicate on every loop iteration; in fact we expect the predicate to be evaluated $n+1$ times is the loop body is executed n times. Hence both the predicate expression and the controlled statement are delayed and we begin by interpreting the predicate to decide whether to execute the loop body at all. We then loop over the body, re-evaluating the predicate each time.

6.13.4 Procedure declaration and call

Procedure declarations are surprisingly simple. At line 25 we recognise the keyword procedure, an identifier and a statement. We then bind the identifier to the statements attribute block in the procedures map. A call then simply needs to retrieve the attribute block bound to a procdure's name and interpret it.

It is fair to say that this mechanism is very limited. All variables are global; there are no parameters and there is no way to return early from a procedure. Adding scopes and parameters is quite challenging; adding named parameters even more so. Even deciding which parameter passing mechanism to use requires very careful thought.

We leave these as exercises for the reader.

Bibliography

[BG05] Joshua Bloch and Neal Gafter. *Java Puzzlers*. Addison Wesley, 2005.

[DDH72] O. J. Dahl, E. W. Dijkstra, and C. A. R. Hoare, editors. *Structured programming*. Academic Press Ltd., GBR, 1972.

[KD14] Donald. E. Knuth and Edgar G. Daylight. *Algorithmic Barriers Falling*. Lonely Scholar, 2014.

[Knu68] Donald E. Knuth. Semantics of Context-Free Languages. *Mathematical Systems Theory*, 2(2):127–145, 1968.

[Knu71] Donald E. Knuth. Semantics of Context-Free Languages: Correction. *Mathematical Systems Theory*, 5(2):95–96, 1971.

[Knu90] Donald E. Knuth. The genesis of attribute grammars. In P. Deransart and M. Jourdan, editors, *Attribute Grammars and their Applications*, pages 1–12, Berlin, Heidelberg, 1990. Springer Berlin Heidelberg.

[MHMT97] Robin Milner, Robert Harper, David MacQueen, and Mads Tofte. *The Definition of Standard ML (Revised Edition)*. The MIT Press, 1997.

[Plo04] Gordon D. Plotkin. A structural approach to operational semantics. *J. Log. Algebraic Methods Program.*, 60-61:17–139, 2004.

[SJW23] Elizabeth Scott, Adrian Johnstone, and Robert Walsh. Multiple input parsing and lexical analysis. *ACM Trans. Program. Lang. Syst.*, 45(3), July 2023.