Course welcome

Welcome to the Software Language Engineering course.

You have been engineering *with* software languages for at least two years. This course, though, is about the engineering *of* software languages: I shall show you how to build languages using concise formal notations from which the implementation could be automatically generated.

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

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

In practice we mostly write software in a *hopeful* way, and then use testing to try and 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.

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

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

Course structure

This course runs over eleven weeks, with four sessions per week and will be assessed as 50% examination and 50% project.

Each week we shall have two hours of presentations by me, using a mixture of pre-recorded videos, live discussion and quizzes.

One more hour a week will be dedicated to a scripted laboratory in which you will work through a set of examples and then attempt some follow-on exercises in your own time. Nearly all of the labs use a toolkit called ART which was developed here at Royal Holloway.

The fourth hourly session supports your individual project work. Within the project you will develop your own Domain Specific Programming language in stages, and each week I will check in with you individually to monitor your progress.

The required resources for this course are my book *Software Language Engineering* and the software tool *ART* which provides implementations of the techniques discussed in the book.

There are also many short videos with accompanying slides and captions which I will be using in lectures. Now, if I am honest, I find watching videos rather a slow way to learn since I can read at least three times faster than I speak, so if you'll excuse the recursive nature of this comment, I'd like to suggest that you might try reading the transcripts for yourself rather than passively watching me read them out to camera.

Formalising language

You need a language just to engage with this course: human language or *natural language* (specifically, early 21st century English). The process by which my words induce meaning in your mind is of great interest but still very mysterious, and we shall not consider it further. Our focus will be purely on *formal* languages: that is synthetic notations used mainly with computing systems, and also in mathematics and engineering. Programming languages are the pre-eminent example of formal languages.

When you learn to program, you are often shown rules which explain the syntactically valid forms of a language construct. For instance, in C and Java the keyword `if` must always be followed by a left parenthesis and the compiler will reject your program if you breach that rule. If done well, a collection of such rules can be used to automatically write a special program called a *syntax analyser* which will check your program for correctness.

In this context formal means more than just synthetic: it means that we work within a set of consistent rules that are susceptible to mechanisation. Formal systems typically have *configurations* which we move between according to formal rules. Now, mechanisation does not necessarily mean that judgement becomes superfluous. Board games such as Chess, Go and Draughts are all formal systems which present a variety of possible moves at each stage. The art of winning is to choose the best move.

Analysing syntax is a start, but we would also like to formalise the semantics of a program: to turn it into some sort of mechanisable game. That turns out to be harder, and there are several competing approaches. Later, we shall look at two such techniques: Structural Operational Semantics and attribute grammars. First though, let's look in more detail at formalisation itself.

Rewriting systems, types and values

Rewriting is at the heart of everything we do on this course. At the end of the previous section we saw a rewrite interpretation of a simple program: we picked a part of the program to execute, then rewrote the program so that it contained only the ‘whats left to do’ part, whilst at the same time recording any side effects (such as variable updates) in auxiliary semantic entities.

We can also use rewriting to perform syntax analysis, by decomposing the program into its component syntactic parts. *String* rewriting systems enable us to show how to *generate* a program in, say, Java from a single symbol, or conversely how to *parse* a program back to a single symbol. The sequence of rule applications during such a *derivation* naturally generates a tree like structure which itself can form the basis of the semantics phase using *tree* or *term* rewriting.

We shall look at these processes in detail in later sections. In this section we are going to develop the terminology of rewriting systems and look at efficient ways to implement them. So this section is about building infrastructure for rewriting systems that we shall use in later sections to build language interpreters.

Semantics specification using Structural Operational Semantics

In this section we are going to formalise and mechanise the rewrite execution model that we saw at the end of the first section. There are two main tasks: we have to have rules that rewrite fragments of the program, and we have to have a process by which we locate the next appropriate rewrite: that is we need the rules themselves and we need a *strategy* for applying them.

In a Structural Operational Semantics we use the strategy of only rewriting at the root: that is each step of the program corresponds to a rewrite of the root of the program term. The initial program term is the whole program written as a prefix expression; in general the program gets smaller as the interpreter rewrites it (though there are exceptions!) This strategy works well for programming language which have compositional semantics (which is most of them).

There is an important initial task which is to decide what shape the program *configurations* should have. In a pure functional language, a configuration would only need the program term itself. In a procedural language we would also need a store, representing memory, and probably an environment representing the collection of identifiers that the user program has created. We might also need representations of input, output and even exceptions. But of course we shall begin with simpler examples.

Syntax specification and analysis

The *internal* syntax of terms in eSOS on is rather uncomfortable for humans to read. The regularity of a prefix syntax is just what we need when when we are writing rules that are activated by pattern matching, but people clearly like more visual cues and fewer parentheses. We learn in primary school to use symbols such as the upright cross for addition, the dash for subtraction and two parallel lines to denote equality. Since these symbols are so broadly understood it is easy to imagine that they are somehow natural and inevitable, but in fact + and - were only introduced in the middle of the sixteenth century: prior to that most arithmetic and geometric texts used a prose style. In the early days some languages, notably COBOL, also used prose phrases to describe arithmetic using phrases such as

```
ADD A B GIVING TOTAL
```

which are spelt out in words.

Clearly, then, there are lots of ways of expressing a computation in human friendly *external* syntax. This section is about how we can use string rewrite rules and *parsing algorithms* to extract from an external syntax phrase its essential structure – it's internal syntax. Our approach will be to use so-called Context Free string rewrite rules which admit polynomial time parsing algorithms: in particular the GLL algorithm developed here at Royal Holloway By Professor Elizabeth Scott and me.

If we specify the grammar carefully, we can arrange things so that the derivations naturally generate terms suitable for use with eSOS. To assist in this task we shall use the GIFT annotations which apply simple-to-understand local rewrites to the derivation tree. The GIFT idea was also developed here.

Semantics specification using attribute equations

In the rewrite execution model, a lightly modified derivation tree is successively rewritten to some final value. An alternative way of specifying program semantics is associate attributes with each node of the derivation tree, and give equations that specify how attribute values are to be constructed. In a less declarative style, we might even allow fragments of some programming language to be used to specify the computations.

This hybrid of string rewriting and equations (which is usually called an *attribute grammar*) is the most well known way to build language translators, and nearly all real parser generation tools offer some flavour of attribute evaluation. In principle, it can be an efficient technique because the underlying derivation tree itself is not changed, well at least not in simple applications. Tree rewriting can be expensive, so not having to rewrite is attractive. Where we want to model control flow, especially so-called abnormal contol flow such as arbitrary jumps, things can get more difficult.

In this section we shall look at one particular flavour of attribute evaluation (L-attribution with delayed attributes) which allows Java-style control flow to be modelled without modifying the tree. The delayed attribute idea was developed at Royal Holloway.

Case studies and the industrial context

So what is all this for? Other people have already built the programming languages we use to engineer our software designs, so why study language internals? Well, of course, there is the sheer pleasure of finding out how things work, which is the best motivation of all. But there is an industry focus to all this too.

In this section we shall look at a three domains — music, 3D CAD and image processing — as exemplars of the Domain Specific Language (DSL) concept. I hope that this material may also give you inspiration for your project work.

Right at the beginning we said *it's languages all the way down.* Well, it's also languages all the way across. Once a business process becomes too complex for a point and shoot menu interface, or once we want to automate processes with a script, we quickly get to the point where a custom DSL might seem like a good idea. There is a plethora of these languages hiding inside systems ranging from large scale database management through CAD tools, games and down into the embedded systems that run our vehicles and media systems. How do we know they don't have severe bugs? In a networked world, the fragility of these *ad hoc* little languages becomes a significant business risk, especially when we consider the security or our connected systems.

That's a big subject for a follow-on course, but formal specification and automatic interpreter generation give us at least a chance of being able to verify correct behaviour, as well as massively reducing the effort required to modify and extend little languages.

Laboratory sessions and the individual project

Program execution usually requires successive translations into multiple languages and formats on their way to being interpreted directly by hardware: a multi-layered translation from high level notations that are comfortable for humans to read and write, down through less and less abstract notations until we arrive at physical switching elements which perform the actual transformation of digital data.

Although the languages in these layers have wildly different roles in our software eco-systems, they all share two main *facets*. They have a *syntax* which describes the things that may be legally said in a language, and they have a *semantics* which specifies what should happen when a particular fragment of a language is encountered. For more than fifty years we have known how to formally specify syntax and automatically check that a particular program or input is syntactically legal. The story with semantics is less happy: although a variety of more or less mathematical notions and notations have been developed to allow *formal* specification of semantics, none have really found favour with the practitioner community.

Does this matter? After all, there are not very many programming languages in use, so perhaps you will never need to look at the inside of a language processor. Well it turns out that in many industrial and business processes are mediated by so-called *Domain Specific Languages* and if you are unlucky, you may one day be asked to modify or maintain such a language. I say unlucky, because most of these languages have been built in a fairly *ad hoc* way, and since they have few users they tend not to be well tested. As a result, their language processors are *fragile*. Our goal is to learn principled techniques that support robust and maintainable processors, and in these labs you will put these ideas into practice.