Formal Programming Language Semantics note 1

What is formal semantics?

This course is about the problem of giving precise, mathematical definitions of programming languages, and about the kinds of things that one can do with such a definition.

Syntax, semantics, pragmatics

The task of describing any language (natural or artificial) can roughly speaking be broken down into various components, e.g.

Syntax: This deals with the question "Which strings of symbols (or words, or tokens) count as valid expressions in the language?"

Semantics: This deals with the question "What do valid expressions actually *mean*?" In the case of programming languages, we typically interpret this question as "What do programs actually do when you run them?"

Pragmatics: This deals with the question "How are expressions of the language used in practice?"

For instance, we would expect a description of English *syntax* to tell us that

This is the malt that lay in the house that Jack built.

is a correctly formed sentence, while most other random rearrangements of these words yield gobbledegook. Likewise, a description of Java syntax will tell us that

```
int j = 1 ;
for (int i=2 ; i<=n ; i++) {j *= i}
```

is (I hope) a legal fragment of Java code, while a lot of other things aren't. (For a language like Java, a legal bit of code is typically one that will be accepted by the compiler.)

In fact, a syntax of a language will usually tell us a bit more than this: e.g. it will tell us how to associate a syntactic structure with a given expression, usually in the form of a *parse tree*. [Exercise: sketch plausible parse trees for the above English sentence and Java fragment.]

There are well-known formal techniques for describing the syntax of languages and defining parse trees, most notably *context-free grammars* (I will assume you know what these are). These were introduced by Noam Chomsky for the purpose of describing aspects of English syntax, but are now a standard tool for describing programming languages.

A description of the *semantics* of English would explain how the meaning of the above sentence can be inferred from the syntactic structure of the sentence, together with the meanings of the individual English words like “malt”, “house”, “built”.[1] Thus, it would make it clear that it was Jack who built the house, rather than the house that built Jack.

Likewise, a semantics of Java would tell us what the various constructs in the above fragment mean, and hence what will be the effect of running it. Thus, for instance, it will allow us to predict that if `n` holds the value `5`, the execution of the code will end in a state in which `j` holds the value `120`. Various formal techniques for specifying the semantics of programming languages are available, and they will be the main subject of this course. (One can also study formal approaches to natural language semantics, but this is very much harder — see e.g. much current research in linguistics and cognitive science.)

A description of English *pragmatics* might tell us that the above sentence is the kind of thing a English speaker might say in practice, whereas

> Jack built the house the malt the rat the cat killed ate lay in.

is not, even though it is a grammatically correct sentence in some theoretical sense. It will also, for example, explain the nature of the difference between the sentences

> Jack built a house.
> A house was built by Jack.

(These sentences have the same *meaning*, or very nearly so — for instance, they are true in exactly the same situations — but there is nevertheless some difference between the way they are used.) In addition, pragmatics covers the way in which speakers use certain words, constructions or intonation patterns to provide “cues” concerning the large-scale (or *discourse level*) structure of speech.

For programming languages, pragmatics would cover things not normally considered to be part of the language definition itself, such as commonly occurring *patterns* for achieving certain programming tasks, or ideas relating to “good programming style” and other software engineering issues.

[1]The term ‘semantics’ was apparently first used by one M. Bréal in his *Studies in the Science of Meaning*, published in 1900. He used the word to mean “the study of the way a word changes its meaning”. Nowadays, the word is often used in ordinary English to mean “the study of the meaning of words”. In technical disciplines like logic, computer science and linguistics, it is used to mean more generally “the study of meaning”. (Thus, to digress briefly into the semantics of semantics, we see that the word ‘semantics’ has itself changed its meaning, and means slightly different things to different people.)

[Exercise: How many possible parse trees can you associate with the English phrase "Formal Programming Language Semantics"? What different meanings do they correspond to? Which meaning do you think is pragmatically the most likely, based on what you know so far of the subject?]

Compile time and run time

In the case of natural languages, the distinctions between syntax, semantics and pragmatics are somewhat blurry. This is related to the fact that our brains appear to process all three aspects of speech concurrently, and they each feed into the others. For instance, consider the sentences:

Time flies like an arrow.
Fruit flies like a banana.

Here we use pragmatic information and real-world knowledge — what the speaker is likely to want to say — to determine which is the appropriate parse tree for the sentences.

For programming languages, the distinctions are often much more clear-cut, owing to the fact that in most modern languages there is a clear distinction between a *compilation phase* and an *execution phase*. We can therefore say that syntactic issues are those that are addressed at compile time, and a syntactically correct program is one that will get past an (ideal) compiler — while the semantics concerns its execution behaviour on an (ideal) run time system. Pragmatic issues are typically things not enforced by either the compile time or run time system, though they might be enforced e.g. by the software development environment the programmer is using.

In this course we will concentrate on languages with a clear separation between compilation and execution. If one wishes to describe languages in which these can be mixed (e.g. the run time system can call the compiler), the syntax/semantics distinction might become correspondingly blurry — though a judicious mix of the same formal tools should still suffice to describe the language.

Static versus dynamic semantics

In many languages, there are a bunch of aspects which count as syntactic by the above definition (they are checked at compile time), but will not normally be captured by a (sensible) context-free grammar of the language. These include things like *typing* rules for variables and expressions, and *scoping* rules ensuring that we only refer to things in a place where they have been declared. Thus, for example, in the above code, replacing `j *= i` by `j += "hello"` will result in a type error. There is some analogy here with the notion of *agreement* in natural language. Consider the utterance

* The house that Jack built contain some malt.

Clearly there is some kind of clash here between the singular noun "house" and the plural verb "contain". Issues of agreement are considered by linguists to be part of syntax, though they are not (most naturally) captured by context-free grammars.

Although we can regard issues such as typing and scoping as officially part of syntax, the techniques that are used to specify these aspects of a language are actually closer to those used in semantics. For this reason, these aspects are sometimes referred to as the *static semantics* of a language, in contrast to the run time or *dynamic semantics* described above. Static semantics, and especially type systems, are becoming increasingly important in a wide range of languages. In this course we will cover both static and dynamic semantics, thus giving an overview of the full range of techniques used to specify languages.

Formal versus informal semantics

Most widely used programming languages have official standard definitions. In many cases (e.g. Pascal, Ada, Java), these definitions consist of careful English prose, written by a panel of experts. Such descriptions attempt to cover not only every individual construct in the language but all possible combinations and interactions of these constructs. However, in the case of a large and complex language, it is in practice very difficult to be sure that such a description is really sufficiently precise and free from ambiguity. Moreover, the subtle issues that can arise from the interaction of several features tend to proliferate rapidly as the size of the language increases, and it is hard to be certain that one has taken account of all the possibilities. Furthermore, such English descriptions of programs have a tendency to become long-winded and incomprehensible.

In contrast to these *informal* definitions, we will be considering *formal* ways of specifying the semantics of programming languages. The best example to date of a fully formal definition for a full-scale programming language is the Definition of Standard ML (1990, revised 1997); this provides a good illustration of some of the techniques we will discuss. However, formal semantics has also had a major influence on the design of many other languages (e.g. Ada, Java), even though their official definitions are not presented in this way.

Note that people use the word *formal* in two slightly different ways: (a) mathematically precise and rigorous, though maybe expressed in ordinary "mathematical prose"; (b) expressed within a *formal system* whose rules are defined in terms of pure symbol-pushing. The descriptions of programming languages we consider in this course will certainly be formal in sense (a). Many of them will be close to being formal in sense (b) too, and will therefore lend themselves to manipulation on a computer.

John Longley