There is usually two parts to the definition of a (programming) language. The first part is what is called the syntax, that is, the sequence of characters, the language constructs (building blocks) that allow one to construct a program in that language. The second part of a language definition is what is termed as the Semantics of the language. Semantics often refers to the behaviour of a program in the language. This can in turn be defined in terms of the behaviour of the individual language constructs and how these interact with one another. This module will deal with language semantics.

Describing a Language

<table>
<thead>
<tr>
<th>Syntax: What sequence of characters constitute a program?</th>
<th>Deals mainly with grammars, lexical analysis and parsing.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Semantics: What does a program mean? What does a program do? What properties does a program have? When are two programs equivalent?</td>
<td></td>
</tr>
</tbody>
</table>

There are a number of reasons why we may want to describe the semantics of a language:

- The primary reason is that of understanding a language, because without it, we could not program correctly in that language. In particular, we would like to understand the language without knowing exactly how the compiler underneath works. This separation of concerns between the language behaviour and actual language implementation has a number or reasons. For example, for different architectures, language should have the same behaviour but clearly the implementation changes. Understanding a language also enables us to determine whether a compiler correctly implements the language and whether certain optimisations preserve the behaviour of a language.

- Another reason for the semantics of a language is that it allows us to express design choices in a language and permits us to compare language features independent of their syntactic representation. Semantics enables us to study how different features interact with one another, brings to light certain language ambiguities and permits cleaner and clearer ways of organising these design choices.

- Most importantly though, rigorous semantics gives us a foundation on which to prove properties of programs in the language and properties of the language itself. We will discuss this aspect at length in this course.
Methods for describing the semantics of a language vary from the obscure to the informal to the precise. Back in the day, when languages were hacked by a few colleagues in a lab or by some enthusiasts in some basement, few bothered giving descriptions of how the language behaved (or how it was expected to). Indeed, in many cases, the language itself evolved with the implementation of the compiler itself, whereby language features were somewhat ad-hoc and the behaviour was motivated by implementation concerns. In these cases, the only 'definition' of the language was the compiler itself.

Nowadays, many programming languages are described in natural language, e.g. the English standards documents for C, Java, XML, etc. Though these descriptions are reasonably accessible there are some major problems associated with them. For a start, it is very hard, if not impossible, to write really precise definitions in informal prose. The standards often end up being ambiguous or incomplete, or just too cumbersome to understand. That leads to differing implementations and flaky systems, as the language implementors and users do not have a common understanding of what it is. More fundamentally, natural language standards obscure the real structure of languages - it’s all too easy to add a feature and a quick paragraph of text without thinking about how it interacts with the rest of the language.

Instead, as we shall see in this course, one can develop mathematical definitions of how programs behave, using logic and set theory (e.g. the definition of Standard ML, the .NET CLR, recent work on XQuery, etc.). These require a little more background to understand and use, but for many purposes they are a much better tool than informal standards.

There are various ways how to give formal mathematical descriptions of semantics, most of which are complementary to one another. They are usually classified into three board categories called operational, denotational and axiomatic semantics. In this course we will briefly touch on the latter two, but mainly focus on the former as our main vehicle for expressing language behaviour.

Even though formal semantics supports reasoning about programs, analysing the runtime behaviour of a program can be a very expensive task, and for a large class of programs and languages, an intractable problem. We shall therefore seek ways how to asses the behaviour of a program by simply analysing its code (i.e. without running it). This is often refered to as the static semantics of a language. Static methods are
(Dynamic) Semantics Styles

Operational: A program’s meaning is given in terms of the computational steps induced when the program is executed.

Denotational: A program’s meaning is given in terms of an element in some abstract mathematical structure.

Axiomatic: A program’s meaning is given indirectly, in terms of a collection logical properties the program execution satisfies.

Slide 4

obviously less expensive, but at the same time, less refined, yielding approximations of program behaviour. In this course we will study how these static methods can be used, and prove properties showing how they are in accordance with their dynamic semantic counterpart.

Dynamic Vs Statics Semantics

Dynamic: gives meaning based on the runtime behaviour of a program. Analysis is expensive and often intractable.

Static: gives meaning based on the compile-time structure of the program. Analysis is less expensive but can only approximate the runtime behaviour.

Slide 5

We will be exposed to the art of distilling the essential language feature we want to study in terms of a tiny language exhibiting such feature (along with some other core constructs). Through these small languages, we shall filter out certain redundant language constructs which can be expressed in terms of other constructs. This modus-operandi will facilitate the formal analysis of the languages and constructs and also help us appreciate how even small languages involve delicate design choices.

There are a number of aims this module sets out to achieve. Indeed by the end of this course, the student should have:

• put to use technical tools such as inductive definitions, formal proofs etc..
• seen how different programming paradigms and features are formally expresses and how they interact with one another.
• a good understanding of both static and dynamic semantics and a better appreciation of how they are employed in industry-strength languages.

How does this module fit in with the rest of the course:
We shall consider simplistic, small programming languages (called calculi) which capture the particular language feature we are interested in.

The assumption will always be that we can scale up to more realistic programming languages.

It will assume the theory related to parsing and machines for parsing (Formal Languages and Automata). Indeed, as we shall see, our starting point will be abstract syntax, i.e., syntax that has already been parsed into a parse tree.

It ties in with the Compiling Techniques course. One needs to understand clearly how a language behaves before constructing a compiler for it. This course may also distill certain concepts from that module which at the time, may have seemed ad-hoc.

The concepts and mechanisms studied in this course may be extended to other modules such as algorithm and protocol descriptions in the Formal Methods course.

The student is encouraged not to limit herself to these notes and to consult other books for a more in-depth understanding of the course. In particular, students should consider:

- The Semantics of Programming Languages: An Elementary Introduction Using Structural Operational Semantics by Matthew Hennessy (for the operational semantics part).
- Types and Programming Languages by Benjamin C. Pierce (for the type systems part).
- Formal Semantics of Programming Languages by Glynn Winskel (other forms of semantics such as denotational and axiomatic).