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

Syntax: What sequence of characters constitute a program? Deals mainly with grammars, lexical analysis and parsing.

Semantics: What does a program mean? What does a program do? What properties does a program have? When are two programs equivalent?

Slide 1

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 broad 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 assess the behaviour of a program by simply analysing its code (i.e. without running it). This is often refereed 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:
Language Calculi: Micro to Macro

- 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.

Slide 6

- 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).
1 Expression Language

We start off with a very simple language called $L_1$. This language describes very simple arithmetic and boolean expressions. We shall take advantage of its simplicity as a foundation to illustrate the concepts we will be using throughout this module. Moreover, we shall use $L_1$ to show that, despite its limitations, we are already faced with non-trivial design decisions and interesting interactions amongst constructs.

$L_1$ essentially consists of the (infinite) set of expressions, $\text{Exp}$, whose elements, $e$, are inductively defined by the BNF rules given in Slide 7. Technically speaking, $L_1$ has three syntactic categories: the principal one, $\text{Exp}$, and two auxiliary ones, booleans $b \in \text{Bool}$ and numerals, $n \in \text{Num}$. Recall that (well-founded) inductive definitions give us infinite sets, whereby every element is finite. In our case, even though we can have an infinite number of elements in $\text{Exp}$ every element $e$ in $\text{Exp}$ is finite in structure. We will find it convenient to refer to the elements $v$ in the set $\text{Bool} \cup \text{Num}$ as values.

$L_1$ Definition

\[
e := b \mid n \mid e + e \mid e - e \mid e \leq e \mid e \&\& e \mid \neg e
\]

\[
b := \text{true} \mid \text{false}
\]

\[
n := 1 \mid 2 \mid 3 \mid \ldots
\]

- Expressions are inductively defined.
- Expressions are defined at the level of abstract syntax.
- Numerals are not the natural numbers. Booleans are not boolean values.

Slide 7

There are two other important points to highlight from our language definition. The first point is that we shall work with abstract syntax. In other words, we will assume that our programs have already been parsed, and expressions actually denote syntax trees rather than concrete syntax (see Slide 8). For this reason we will often use brackets as meta-syntax to precisely describe the syntax tree of syntax which can be ambiguously parsed. Thus we often represent $1 + 2 + 3$ as either $1 + (2 + 3)$ or $(1 + 2) + 3$ and not concern ourselves with how the ambiguity has been dealt with during parsing. This tree-view of syntax brings to the fore that syntax such as + and $\&\&$ are expression-forming operations, or constructors: they take two expressions and give back a new one. We will use this view when we give semantics to our expressions.

The second important point is that $\text{Num}$ is a set of numerals, whereby every numeral is merely a sequence of digits (Slide 9). For simplicity, we shall not be interested in how these numerals are actually defined, but they will be considered as atomic objects requiring no further analysis. Thus, the numeral $12$ will be treated as a single unit and never as the separate digits 1 and 2. More importantly though, numerals are distinct from numbers: the numeral $12$ will be syntax that denotes the number $12$, i.e., twelve (notice the difference in font used). Whereas we can add two numbers, we cannot add two numerals! Instead, we have to define a relation ourselves that computes the results of adding the numbers denoted by the numerals. These same principles are applicable to booleans as well.
Concrete Vs Abstract Syntax

The concrete syntax \( 3 - 2 - 1 \) is ambiguous and can represent different syntax trees. Clearly, the two syntax trees have different semantics.

Numbers Vs Numerals

**Numbers:** Mathematical entities we use in everyday life. We use the meta-syntax 1 to denote *oneness*, 2 to denote *twoness*, 3 to denote *threeness* etc.

**Numerals:** Syntax describing numbers. We use the syntax 1 to denote 1, 2 to denote 2, 3 to denote 3 etc., but we could have easily used 1, 10 and 11 to denote 1, 2 and 3 instead.

The same principle applies to the booleans *true* and *false*, which are syntactic representations of truth and falsity.

1.1 Big-Step Semantics

An operational semantics for our expression language \( L_1 \) describes how expressions evaluate into values. The first form of operational semantics we shall consider is called *big-step* operational semantics, sometimes also referred to as *natural semantics*. The distinguishing feature of this semantics is that it ignores the intermediate steps required by an expression to evaluate into a value. Intuitively, it is mainly concerned with the relationship between the initial and final state of an evaluation.

More formally, the big-step semantics of \( L_1 \) takes the form of a relation, denoted by the symbol \( \Downarrow \), between expressions, \( \text{Exp} \), and values, \( \text{Bool} \cup \text{Num} \). It is in fact, defined to be the *least* relation defined by the rules in Slides 10 and 11, whereby we write \( e \Downarrow v \) to denote that \( (e, v) \in \Downarrow \).

The rules defining the relation consist of two *axioms* (rules with no premises) called E_NUM and E_BOOL and five *inductive* rules. The axioms express the fact that values, *i.e.*, numerals and booleans, already denote evaluated expressions, so we do not need to evaluate them further. The other five rules are called inductive because their definition relies on premises, who themselves rely on the same rules. There are two further points that are important to point out in the above rules:

1. Rules such E_ADD rely on side-conditions to obtain the final answer. Recall that numerals themselves
**Big-Step Semantics for L₁**

\[ \downarrow : \text{Exp} \times (\text{Bool} \cup \text{Num}) \]

- \( n \downarrow n \) \( \text{NUM} \)
- \( b \downarrow b \) \( \text{BOOL} \)
- \( e₁ \downarrow n₁ \) \( e₂ \downarrow n₂ \) \( e₁ + e₂ \downarrow n₃ \) \( \text{EADD} \) \( \text{where } n₃ = n₁ + n₂ \)
- \( e₁ \downarrow n₁ \) \( e₂ \downarrow n₂ \) \( e₁ - e₂ \downarrow n₃ \) \( \text{ESUB} \) \( \text{where } n₃ = n₁ - n₂ \)

**Slide 10**

**Big-Step Semantics for L₁**

- \( e₁ \downarrow b₁ \) \( e₂ \downarrow b₂ \) \( e₁ \& e₂ \downarrow b₃ \) \( \text{EAND} \) \( \text{where } b₃ = b₁ \land b₂ \)
- \( e \downarrow b \) \( e\neg \downarrow b₁ \) \( \text{ENOT} \) \( \text{where } b₁ = \neg b \)
- \( e₁ \downarrow n₁ \) \( e₂ \downarrow n₂ \) \( e₁ \leq e₂ \downarrow b \) \( \text{ELEQ} \) \( \text{where } b = n₁ \leq n₂ \)

**Slide 11**
cannot be added together (or operated on directly for that matter). The rule however relies on the assumed direct correspondence between a numeral \( n \) and the natural number \( n \). Thus the side condition imposes conditions on the natural numbers corresponding to the numerals and not the numerals themselves (notice the difference in font).

2. Our rules are *rule schemas*. Because each \( e, n, b \) and \( v \) are meta-variables, each rule is really a pattern for an infinite collection of rules. To obtain a *rule instance* we need to instantiate every metavariable with a corresponding expression, numeral, boolean and value instance.

### Slide 12

Relations defined inductively provide a mechanism for determining whether a pair is in the relation or not. In other words, we can show that a pair is in a relation by providing a *derivation* justifying its membership, using the same rules defining the relation. For instance, to show that \((1 + (2 + 3)) \leq 7 \Downarrow \text{true}\) all we need to do is give a witness derivation for the statement, i.e., a proof justifying its membership (Slide 13).

Inductively defined relations also provide a mechanism for proving properties about the relations themselves. For instance, we can show that our definition for the big-step semantics of \( L_1 \) observes *determinacy* (Slide 14). Intuitively, determinacy states that if we can evaluate an expression into a value, then we will obtain the same value *every time* we evaluate the expression. Note that when stating properties, top-level universal quantifiers are usually left out. Thus, what determinacy really says is “*For all expressions* \( e \) and values \( v_1, v_2 \) if \( e \Downarrow v_1 \) and \( e \Downarrow v_2 \) then it must be the case that \( v_1 = v_2 \).” Note also that another way how to view determinacy is by saying that our relation \( \Downarrow \) is indeed a function.

**Theorem** (Normalisation - does not hold for \( L_1 \)). *For any expression* \( e \), *then there exists a value* \( v \) *such that* \( e \Downarrow v \)

Inductively defined relations also provide a mechanism for disproving properties about themselves. For instance, we can show that the following Property is does not hold for our relation \( \Downarrow \). As we shall see in more detail later on, the way we prove Theorem 1 is different from how we disprove the Normalisation Theorem stated above. In the first case, our proof is by induction. In the second case, it suffices to provide a
A proof that \((1+(2+3)) \leq 7 \Downarrow true\)

\[
\begin{array}{c}
\text{eNum} \quad \text{eNum} \quad \text{eNum} \\
1 \Downarrow 1 \\
2 \Downarrow 2 \\
3 \Downarrow 3 \\
\text{EAdd} \\
2+3 \Downarrow 5 \\
\text{EAdd} \\
1+(2+3) \Downarrow 6 \\
\text{EAdd} \\
7 \Downarrow 7 \\
\text{ELeq} \\
(1+(2+3)) \leq 7 \Downarrow true
\end{array}
\]

Slide 13

Properties about the relation \(\Downarrow\)

Theorem 1 (Determinacy). If \(e \Downarrow v_1\) and \(e \Downarrow v_2\) then \(v_1 = v_2\)

Slide 14

counter-example. For instance, we can argue that there is no \(v\) such that \(1 + \text{true} \Downarrow v\) - thus the expression \(1 + \text{true}\) is our counter example.

Despite the simplicity of \(\mathcal{L}_1\), we are already in a position to appreciate certain design decisions that have been made in its big-step evaluation semantics. For instance, one alternative semantic interpretation for our conjunction operator is referred to as the short-circuiting semantics and is captured by the rules in Slide 15; in fact this is the semantic interpretation given for the constructor in languages such as Java. If we substituted \(\text{EAnd}\) for \(\text{EAnd1}\) and \(\text{EAnd2}\), then we would obtain a different semantics for our language. The formal definition of our language helps us clarify and determine this discrepancy through a witness-example with a different evaluation behaviour in the two semantics. For instance, consider the expression \(\text{false} \&\& 2\). In our original semantics this expression does not evaluate, but in the short-circuiting semantics, it evaluates into \(\text{false}\).

Short-Circuiting Semantics for \(\&\&\) in \(\mathcal{L}_1\)

\[
\begin{array}{c}
c_1 \Downarrow \text{false} \\
\text{EAnd1} \\
c_1 \&\& c_2 \Downarrow \text{false}
\end{array}
\]

\[
\begin{array}{c}
c_1 \Downarrow \text{true} \\
\text{EAnd2} \\
c_2 \Downarrow b \\
c_1 \&\& c_2 \Downarrow b
\end{array}
\]

Slide 15
One can also already appreciate how different semantics interpretations can lead to different properties for a language semantics. For instance, assume a third semantics for $L_1$, whereby we keep $\text{eAnd}$ but add $\text{eAnd1}$. This addition, though useful because it allows us to short-circuit evaluating expressions in certain cases, causes the loss of the Determinacy property of our semantics, discussed in Slide 14. In fact, the expression $\text{false} \& \& 2$ could then non-deterministically evaluate to $\text{false}$ and also sometimes not-evaluate at all, which some may deem as an undesirable property in a programming language.

### 1.2 Small-Step Semantics

An alternative form of operational semantics is small-step semantics, more commonly referred to as structural-operational semantics. In small-step semantics, the emphasis is on the individual steps of a computation, without focussing on the final result of a computation. Thus, whereas big-step semantics would immediately give us a final value for an expression (if it exists), small-step semantics would describe more explicitly the individual steps of the computation leading to this final value.

More formally, the small-step semantics of $L_1$ takes the form of a relation, denoted by the symbol $\rightarrow$, between expressions, $\text{Exp}$. This means that the relation now has type $\text{Exp} \times \text{Exp}$, which is different from the type of $\triangleright$, which was $\text{Exp} \times (\text{Bool} \cup \text{Num})$. It is defined to be the least relation satisfying the rules in Slides 16, 17 and 18, whereby we write $e_1 \rightarrow e_2$ to denote that $\langle e_1, e_2 \rangle \in \rightarrow$.

The rules defining the relation consist of five axioms, namely $\text{rAdd1}$, $\text{rSub1}$, $\text{rLeq1}$, $\text{rAnd1}$ and $\text{rNot1}$ and nine inductive rules. The axioms describe reductions involving values, whereas the reduction rules describe how reductions occur inside larger expressions, sometimes referred to as (expression) contexts. We highlight the fact that these small-step rules already reflect certain design decisions wrt. the reduction strategy of expressions. In fact, rules such as $\text{Add2}$ and $\text{Add3}$ restrict the reduction order of binary operators, whereby the left expression needs to be reduced to a value before the right expression can be reduced.

Small-step semantics expresses the evaluation of an expression into a value differently from big-step semantics. For instance, we can describe the evaluation of the expression $(1 + (2 + 3)) \leq 7$ to the value $\text{true}$ through the reduction sequence:

$$(1 + (2 + 3)) \leq 7 \rightarrow (1 + 5) \leq 7 \rightarrow 6 \leq 7 \rightarrow \text{true}$$

whereby every reduction step entails a derivation of its own. We here show the derivation for the first reduction step.

When we are only interested in the final value of a reduction sequence, we may want to abstract away from the intermediary expressions required to reduce to this value. To express this mathematically, we need to consider a relation which expresses multi-step evaluations in Fig. 20. If we expand this (inductive) definition, we realise that $e \rightarrow^* e'$ holds if there exists a reduction sequence of length $n$ where $n \geq 0$ such that:

$$e \rightarrow e_1 \rightarrow e_2 \rightarrow \ldots \rightarrow e_{n-1} \rightarrow e'$$

Clearly, when the reduction length $n = 0$, then $e = e'$ (which constitutes the base case of our inductive definition).

Both reduction relations $\rightarrow$ and $\rightarrow^*$ observe determinism. In fact, determinism for $\rightarrow^*$ follows from the determinism of $\rightarrow$. The inductive nature of the definition of the two relations permits us to prove these statements by induction. We simply state these properties at this point and then tackle the proofs in the coming section.

We have seen two different operational semantics for our language $L_1$. We have also stated that when possible, both semantics compute expressions down to at most one single value (Slide 14 and Slide 21). We would also like to ensure that our two semantics are in agreement. By this we mean that whenever an expression evaluates to a value, $e \downarrow v$, then it is also the case that this expression reduces in a finite number of steps to the same value, $e \rightarrow^* v$, and conversely that whenever an expression reduces (in a finite number of steps) to a value, $e \rightarrow^* v$ then it also evaluates to that same value, $e \downarrow v$ (cf. Slide 22).

For our purposes, the small-step semantics seems very heavyweight. However, Slide 23 outlines aspects whereby a small-step semantics for a language carry clear advantages over big-step semantics.
Small-Step Semantics for $L_1$

\[\rightarrow : \text{Exp} \times \text{Exp}\]

- **RAdd1**: $n_1 + n_2 \rightarrow n_3$ where $n_3 = n_1 + n_2$
- **RAdd2**: $e_1 \rightarrow e_3$ (and similar for $e_2 \rightarrow e_3$)
- **RAdd3**: $v_1 + e_2 \rightarrow v_1 + e_3$

Slide 16

Small-Step Semantics for $L_1$

- **RSub1**: $n_1 - n_2 \rightarrow n_3$ where $n_3 = n_1 - n_2$
- **RSub2**: $e_1 \rightarrow e_3$ (and similar for $e_2 \rightarrow e_3$)
- **RSub3**: $v_1 - e_2 \rightarrow v_1 - e_3$
- **RLeq1**: $n_1 \leq n_2 \rightarrow b$ where $b = n_1 \leq n_2$
- **RLeq2**: $e_1 \leq e_2 \rightarrow e_3 \leq e_2$ (and similar for $v_1 \leq e_2 \rightarrow v_1 \leq e_3$)

Slide 17
1.3 Exercises

1. Is the proof for \(((1+2)+3)\leq7\downarrow \text{true}\) different?

2. Prove that \(\neg((1\leq2)\&\neg\text{false})\downarrow \text{false}\).

3. Can you prove that \(\neg((1\leq\text{false})\&\neg2)\downarrow \text{false}\)?
Derivation for $(1 + (2 + 3)) \leq 7 \rightarrow (1 + 5) \leq 7$

\[
\begin{align*}
(2 + 3) & \rightarrow 5 \quad \text{RADD1} \\
(1 + (2 + 3)) & \rightarrow (1 + 5) \quad \text{RADD3} \\
(1 + (2 + 3)) \leq 7 & \rightarrow (1 + 5) \leq 7 \quad \text{RLEQ2}
\end{align*}
\]

Multi-step Reduction Relation

\[ e_1 \rightarrow^* e_2 \overset{\text{def}}{=} \begin{cases} 
  e_1 = e_2 \\
  \text{or} \\
  \text{there exists an } e_3 \text{ such that } \\
  e_1 \rightarrow e_3 \text{ and } e_3 \rightarrow^* e_2
\end{cases} \]

Determinism for Reductions

**Theorem 2** (Determinism). If $e \rightarrow e_1$ and $e \rightarrow e_2$ then $e_1 = e_2$.

**Theorem 3** (Determinism). If $e \rightarrow^* v_1$ and $e \rightarrow^* v_2$ then $v_1 = v_2$. 
Correspondence between Evaluations and Reductions in \( L_1 \)

**Theorem 4** (Correspondence).

- If \( e \Downarrow v \) then \( e \rightarrow^* v \).
- If \( e \rightarrow^* v \) then \( e \Downarrow v \).

---

**Advantages of Small-Step Semantics**

- In big-step semantics, we cannot distinguish between infinite looping (divergence) and abnormal termination. In small-step semantics looping is reflected by infinite derivation sequences and abnormal termination by a finite derivation sequence that is stuck.
- In big-step semantics, non-determinism may suppress divergence whereas this does not happen in small-step semantics.
- In big-step semantics, we cannot express interleaving of computations in languages expressing concurrency. Small-step semantics easily expresses interleavings.