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 and use it as a foundation to illustrate the concepts we will be using throughout this course. 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.

<table>
<thead>
<tr>
<th>( L_1 ) Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>( e ::= b \mid n \mid e+e \mid e-e \mid e\leq e \mid e&amp;&amp;e \mid \neg e )</td>
</tr>
<tr>
<td>( b ::= \text{true} \mid \text{false} )</td>
</tr>
<tr>
<td>( n ::= 0 \mid 1 \mid 2 \mid 3 \mid \ldots \mid 9 \mid 0n \mid 1n \mid 2n \mid \ldots \mid 9n )</td>
</tr>
</tbody>
</table>

- 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 L1 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.

1.1.1 Inductive Rules

More formally, the big-step semantics of L1 takes the form of a relation, denoted by the symbol ⇓, between expressions, Exp, and values, BOOL∪NUM. It is in fact, defined to be the least relation defined by the rules in Slides 10 and 11, whereby we write e ⇓ v to denote that (e, v) ∈ ⇓.

The rules defining the relation consist of two axioms (rules with no premises) called eNum and eBool 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
Big-Step Semantics for $L_1$

$\downarrow: \mbox{Exp} \times (\mbox{Bool} \cup \mbox{Num})$

- $n \downarrow n$
- $b \downarrow b$

- $e_1 \downarrow n_1$, $e_2 \downarrow n_2$  \( \frac{}{e_1 + e_2 \downarrow n_3} \)
  where $n_3 = n_1 + n_2$

- $e_1 \downarrow n_1$, $e_2 \downarrow n_2$  \( \frac{}{e_1 - e_2 \downarrow n_3} \)
  where $n_3 = n_1 - n_2$

Slide 10

Big-Step Semantics for $L_1$

- $e_1 \downarrow b_1$, $e_2 \downarrow b_2$  \( \frac{}{e_1 \& e_2 \downarrow b_3} \)
  where $b_3 = b_1 \land b_2$

- $e \downarrow b$  \( \frac{-e \downarrow \neg b_1}{\neg e \downarrow b_1} \)
  where $b_1 = \neg b$

- $e_1 \downarrow n_1$, $e_2 \downarrow n_2$  \( \frac{}{e_1 \leq e_2 \downarrow b} \)
  where $b = n_1 \leq n_2$

Slide 11
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 EADD rely on side-conditions to obtain the final answer. Recall that numerals themselves 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.

### 1.1.2 Big-Step semantics as a Proof System

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 $(1 + (2 + 3)) \leq 7$, i.e., $(1 + (2 + 3)) \leq 7.\text{true} \in \downarrow$, all we need to do is give a witness derivation for the statement, i.e., a proof justifying its membership.

Slide 13 depicts one such proof. They are typically read bottom up and they form a tree, with the final conclusion being the root and the premises of the rule being the branches; these branches are either subtrees themselves or else axioms i.e., they do not have any premises, in which case they form the leaves of the tree. One way to construct such a derivation is to start from the conclusion and perform a proof search amongst the rules on Slide 10 and Slide 11 to determine which one to use in order to justify the conclusion. If we find an axiom that does the job i.e., a rule with no premises, our proof construction is completed; else the application of the rule will yield further subgoals i.e., the premises of the rule, that we need to justify as well. We iteratively keep on repeating this process for every (sub)goal until all the branches of the tree are axioms (leaves).
A proof that \((1 + (2 + 3)) \leq 7\) ↓ true

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

In our specific case, since, at the top level, expression \((1 + (2 + 3)) \leq 7\) is an integer comparison, i.e., \(\leq\), we only need to search for rules that apply to integer comparison expressions. Stated otherwise, the top level node of the expression can be used to guide our proof search. A close inspection of the rules on Slide 10 and Slide 11 leads us to conclude that only one rule can be applied, namely \(\text{ELeq}\). Applying this rule yields two subgoals we need to prove for the two sub expressions of \(\leq\) i.e., \(1 + (2 + 3)\) and 7. More precisely, we need to show that \(1 + (2 + 3)\) evaluates to some value \(v_1\) i.e.,

\[1 + (2 + 3) \Downarrow v_1 \quad (1)\]

and 7 evaluates to some other value \(v_2\), i.e.,

\[7 \Downarrow v_2, \quad (2)\]

whereby (the meaning of) \(v_1\) is less than or equal to (the meaning of) \(v_2\). The derivation of (1) follows the same pattern, this time searching for rules that match the top level node \(\text{+}\); once the derivation is completed, it should result that \(v_1\) is the value expression 6 (we leave this derivation to the reader). The derivation of (2) is less involving since we can justify it using an axiom, namely \(\text{ENum}\), from which we determine that \(v_2\) is of the form 7. Since (the meaning of) 6 is less than or equal to (the meaning of) 7, we can justify the conclusion \((1 + (2 + 3)) \leq 7\) ↓ true through \(\text{ELeq}\).

1.1.3 Properties of the Big-Step Semantics

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 \(L_1\) observe determinacy (Theorem 1 on 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 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.
1.1.4 Building Relations from Existing Ones

The formal treatment adopted so far for our small language allows us to formalise relationships amongst terms in the language. For instance, the natural semantics considered so far on Slide 10 and Slide 11 allows us to state precisely when two terms are considered to be equivalent. This is stated formally as Definition 2 on Slide 15. Note that this definition essentially defines a new relation, denoted by the symbol \( \equiv \), over the set of expressions \( \text{Exp} \) i.e., \( \text{Exp} \times \text{Exp} \); more importantly though, this relation is defined in terms of an already defined relation i.e., the relation \( \Downarrow \).

**Properties about the relation \( \Downarrow \)**

**Theorem 1** (Determinacy). \( \text{If } e \Downarrow v_1 \text{ and } e \Downarrow v_2 \text{ then } v_1 = v_2 \)

**Slide 15**

It turns out that the inductive nature of \( \Downarrow \) allows us to prove interesting and useful properties about the new relation \( \equiv \). For instance we can show that the relation is an equivalence relation i.e., it is reflexive, symmetric and transitive; we come to this proof later on in the course.

For now we limit ourselves to using the definition (and the inductive nature of \( \Downarrow \)) as a vehicle to show that two terms are equivalent. For instance, we can show (and give a justification for it in the process) that the terms \( 2+3 \) and \( 6-1 \) are equivalent. According to Definition 2, to determine this we have to show that:

1. For all values \( v \) that \( 2+3 \) evaluates to, the term \( 6-1 \) can also evaluate to that value \( v \).
2. For all values \( v \) that \( 6-1 \) evaluates to, the term \( 2+3 \) can also evaluate to that value \( v \).

Let us start with \( 2+3 \). Using rules \( \text{eAdd} \) and \( \text{eNum} \) we can conclude that this expression evaluates to the value 5. More specifically, we can show this through the following derivation:

\[
\begin{align*}
2 &\Downarrow 2 \\
3 &\Downarrow 3 \\
2 + 3 &\Downarrow 5
\end{align*}
\]

(3)

The implication of the first clause in Definition 2 requires us to show that \( 6-1 \) can also evaluate to 5. We can show this using rules \( \text{eSub} \) and \( \text{eNum} \) as shown below:

\[
\begin{align*}
6 &\Downarrow 6 \\
1 &\Downarrow 1 \\
6 - 1 &\Downarrow 5
\end{align*}
\]

(4)
We note that Definition 2 requires us to verify the above implication for every value that \( 2 + 3 \) evaluates to; this can be quite onerous in the general case since an expression may evaluate to a large number of distinct values. Fortunately however, for our specific case, Theorem 1 comes to our rescue. This property states that any value that \( 2 + 3 \) evaluates to will be (syntactically) equivalent to 5. This means that, in each case, we can reuse proof (4) to show that the right-hand side of the implication holds.

Definition 2 also requires us to show the converse i.e., for every value that \( 6 - 1 \) evaluates to, \( 2 + 3 \) can also evaluate to it. By (4) we know that \( 6 - 1 \) evaluates to 5 and by Theorem 1 we know that this is the only value that it can evaluate to. We can show the right hand side of the implication holds i.e., that \( 2 + 3 \) can also evaluate to 5 by reusing derivation (3).

1.1.5 Design Decisions

Despite the simplicity of \( L_1 \), 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 16; in fact this is the semantic interpretation given for the constructor in languages such as Java. If we substituted \( \texttt{eAnd} \) for \( \texttt{eAnd1} \) and \( \texttt{eAnd2} \), 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 \( \texttt{false && 2} \). In our original semantics this expression does not evaluate, but in the short-circuiting semantics, it evaluates into \( \texttt{false} \).

**Slide 16**

Short-Circuiting Semantics for \( \&\& \) in \( L_1 \)

\[
\begin{align*}
\frac{e_1 \Downarrow \texttt{false}}{e_1 \&\& e_2 \Downarrow \texttt{false}} & \quad \texttt{EAnd1} \\
\frac{e_1 \Downarrow \texttt{true} \quad e_2 \Downarrow b}{e_1 \&\& e_2 \Downarrow b} & \quad \texttt{EAnd2}
\end{align*}
\]

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 \( \texttt{EAnd} \) but add \( \texttt{EAnd1} \). 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 \( \texttt{false \&\& 2} \) could then non-deterministically evaluate to \( \texttt{false} \) 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.
Small-Step Semantics for L₁

$$\rightarrow : \text{Exp} \times \text{Exp}$$

\[
\begin{align*}
n₁ + n₂ & \rightarrow n₃ & \text{rAdd1} & \text{where } n₃ = n₁ + n₂ \\
e₁ & \rightarrow e₃ & \text{rAdd2} \\
e₁ + e₂ & \rightarrow e₃ + e₂ & \text{rAdd3}
\end{align*}
\]

Slide 17

Small-Step Semantics for L₁

\[
\begin{align*}
n₁ - n₂ & \rightarrow n₃ & \text{rSub1} & \text{where } n₃ = n₁ - n₂ \\
e₁ & \rightarrow e₃ & \text{rSub2} \\
e₁ - e₂ & \rightarrow e₃ - e₂ & \text{rSub3}
\end{align*}
\]

\[
\begin{align*}
n₁ ≤ n₂ & \rightarrow b & \text{rLeq1} & \text{where } b = n₁ ≤ n₂ \\
e₁ & \rightarrow e₃ & \text{rLeq2} \\
e₁ ≤ e₂ & \rightarrow e₃ ≤ e₂ & \text{rLeq3}
\end{align*}
\]

Slide 18

1.2.1 Inductive Rules

More formally, the small-step semantics of L₁ takes the form of a relation, denoted by the symbol $$\rightarrow$$, between expressions, Exp. This means that the relation now has type Exp × Exp, which is different from the type of $$\downarrow$$, which was Exp × (Bool ∪ Num). It is defined to be the least relation satisfying the rules in Slides 17, 18 and 19, whereby we write $$e₁ \rightarrow e₂$$ to denote that $$(e₁, e₂) \in \rightarrow$$.

The rules defining the relation consist of five axioms, namely rAdd1, rSub1, rLeq1, rAnd1 and rNot1 and nine inductive rules. The axioms describe reductions involving values, whereas the inductive 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 Add2 and Add3 restrict the reduction order of binary operators,
Small-Step Semantics for $L_1$

$$b_1 \& \& b_2 \rightarrow b_3 \quad \text{where} \quad b_3 = b_1 \land b_2$$

$$e_1 \rightarrow e_3 \quad \text{rAnd2} \quad e_2 \rightarrow e_3 \quad \text{rAnd3}$$

$$v_1 \& \& e_2 \rightarrow v_1 \& \& e_3 \quad \text{rAnd3}$$

$$\neg b_1 \rightarrow b_2 \quad \text{rNot1} \quad \text{where} \quad b_2 = \neg b_1$$

$$e_1 \rightarrow e_2 \quad \text{rNot2}$$

whereby the left expression needs to be reduced to a value before the right expression can be reduced.

1.2.2 Formally Proving Evaluation as a Sequence of Steps

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. Slide 20 shows the derivation for the first reduction step.
1.2.3 Small-Step Transitive Closure

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 (see Slide 21). 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).

Multi-step Reduction Relation

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

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 (see Slide 22) and then tackle the proofs in the coming section.

Determinism for Reductions

**Theorem 3** (Determinism). If $e \rightarrow e_1$ and $e \rightarrow e_2$ then $e_1 = e_2$.

**Theorem 4** (Determinism). If $e \rightarrow^* v_1$ and $e \rightarrow^* v_2$ then $v_1 = v_2$.

1.2.4 Comparison

We have seen two different operational semantics for our language L. We have also stated that when possible, both semantics compute expressions down to *at most one single* value (Slide 14 and Slide 22). 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
Correspondence between Evaluations and Reductions in \( L_1 \)

**Theorem 5** (Correspondence).

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

**Slide 23**

of steps to the same value, \( e \longrightarrow^* v \), and conversely that whenever an expression reduces (in a finite number of steps) to a value, \( e \longrightarrow^* v \) then it also evaluates to that same value, \( e \Downarrow v \) (cf. Slide 23).

In conclusion, note that for our purposes, the small-step semantics seems very heavyweight. However, Slide 24 outlines aspects whereby a small-step semantics for a language carry clear advantages over big-step semantics.

**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.

**Slide 24**

1.3 Exercises

1. Is the proof for \( ((1+2)+3) \leq 7 \Downarrow \text{true} \) different from the one discussed earlier for \( (1+(2+3)) \leq 7 \Downarrow \text{true} \)?

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

3. Can you prove that \( \neg((1\leq\text{false}) \& \& \text{false}) \Downarrow \text{false} \)?

4. Show that \( \neg((1\leq7) \Downarrow 5)) \equiv \neg((1\leq2) \& \& \text{false}) \)

5. (a) Define a function \( \text{op} : \text{Exp} \rightarrow \text{Nat} \) which takes an expressions and returns the number of operators contained in it. For example, we would then have \( \text{op}(\neg((1+7) \leq 5)) = 3 \).
(b) Define a function \( \text{arg} : \text{Exp} \rightarrow \text{Nat} \) which takes an expressions and returns the number of arguments contained in it. For example, we would then have \( \text{arg}(\neg ((4+7) \leq 5)) = 3. \)

(c) Prove that, the following property does not hold:

**Property 1**: For all \( e \in \text{Exp} \) we have \( \text{op}(e) \leq \text{arg}(e) \).

(d) Show that Property 1 holds for the sub-language:

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

6. Using rules as in Slide 12, define inductively the relation \( \rightarrow^* : \text{Exp} \rightarrow \text{Exp} \).

7. Consider the extended language defined as \( e ::= \ldots \mid e_1 / e_2 \):

   (a) Give a big-step semantics for this extended language.
   (b) Give a small-step semantics for this extended language.

8. Can you prove that \( 1 + 2 \equiv 2 + 1 \)?

9. Can you prove that \( 1 + \text{true} \equiv \text{false} \& 1 \)?

16