Today

We will introduce some basic tools used throughout the course:
- Concrete vs. abstract syntax
- Abstract syntax trees
- Induction over expressions

L_{Arith}

We will start out with a very simple (almost trivial) “programming language” called L_{Arith} to illustrate these concepts.

Namely, expressions with integers, $+$ and $\times$.

Examples:
- $1 + 2 \quad \longrightarrow \quad 3$
- $1 + 2 \times 3 \quad \longrightarrow \quad 7$
- $(1 + 2) \times 3 \quad \longrightarrow \quad 9$

- **Concrete syntax**: the actual syntax of a programming language
  - Specify using context-free grammars (or generalizations)
  - Used in compiler/interpreter front-end, to decide how to interpret strings as programs

- **Abstract syntax**: the “essential” constructs of a programming language
  - Specify using so-called Backus Naur Form (BNF) grammars
  - Used in specifications and implementations to describe the abstract syntax trees of a language.
Concrete vs. abstract syntax

Abstract syntax trees

Structural Induction

Concrete vs. abstract syntax

Abstract syntax trees

Structural Induction

CFG vs. BNF

Abstract Syntax Trees (ASTs)

We view a BNF grammar to define a collection of abstract syntax trees, for example:

\[ \begin{array}{c}
\text{+} \\
\text{1} \\
\text{×} \\
\text{2} \\
\text{+} \\
\text{3} \\
\text{1} \\
\text{2} \\
\end{array} \]

These can be represented in a program as trees, or in other ways (which we will cover in due course)

1. Context-free grammar giving concrete syntax for expressions

\[
E \rightarrow E \text{ PLUS } F \mid F \\
F \rightarrow F \text{ TIMES } F \mid \text{NUM} \mid \text{LPAREN } E \text{ RPAREN}
\]

2. Needs to handle precedence, parentheses, etc.

3. Tokenization (+ → PLUS, etc.), comments, whitespace usually handled by a separate stage

BNF conventions

1. We will usually use BNF-style rules to define abstract syntax trees
2. and assume that concrete syntax issues such as precedence, parentheses, whitespace, etc. are handled elsewhere.
3. Convention: the subscripts on occurrences of $e$ on the RHS don’t affect the meaning, just for readability
4. Convention: we will freely use parentheses in abstract syntax notation to disambiguate
5. e.g. 

\[(1 + 2) \times 3 \quad \text{vs.} \quad 1 + (2 \times 3)\]

BNF grammars

1. BNF grammar giving abstract syntax for expressions

\[
\text{Expr} \ni e \quad ::= \quad e_1 + e_2 \mid e_1 \times e_2 \mid n \in \mathbb{N}
\]

2. This says: there are three kinds of expressions

   - Additions $e_1 + e_2$, where two expressions are combined with the + operator
   - Multiplications $e_1 \times e_2$, where two expressions are combined with the × operator
   - Numbers $n \in \mathbb{N}$

3. Much like CFG rules, we can “derive” more complex expressions:

\[e \rightarrow e_1 + e_2 \rightarrow 3 + e_2 \rightarrow 3 + (e_3 \times e_4) \rightarrow \cdots\]
Languages for examples

- We will use several languages for examples throughout the course:
  - Java: typed, object-oriented
  - Python: untyped, object-oriented with some functional features
  - Haskell: typed, functional
  - Scala: typed, combines functional and OO features
  - Sometimes others, to discuss specific features
- You do not need to already know all these languages!

ASTs in Java

- In Java ASTs can be defined using a class hierarchy:

```java
abstract class Expr {
    abstract public int size();
}
class Num extends Expr {
    public int n;
    Num(int _n) {
        n = _n;
    }
}
class Plus extends Expr {
    public Expr e1;
    public Expr e2;
    Plus(Expr _e1, Expr _e2) {
        e1 = _e1;
        e2 = _e2;
    }
}
class Times extends Expr {... // similar
}
```

- Traverse ASTs by adding a method to each class:

```java
... // similar
```
Python is similar, but shorter (no types):

```python
class Expr:
    pass  # "abstract"

class Num(Expr):
    def __init__(self, n):
        self.n = n
    def size(self): return 1

class Plus(Expr):
    def __init__(self, e1, e2):
        self.e1 = e1
        self.e2 = e2
    def size(self):
        return self.e1.size() + self.e2.size() + 1

class Times(Expr):  # similar...
    def __init__(self, e1, e2):
        self.e1 = e1
        self.e2 = e2
    def size(self):
        return self.e1.size() + self.e2.size() + 1
```

In Haskell, ASTs are easily defined as `datatypes`:

```haskell
data Expr = Num Integer  
           | Plus Expr Expr  
           | Times Expr Expr
```

Likewise one can easily write functions to traverse them:

```haskell
size :: Expr -> Integer
size (Num n) = 1
size (Plus e1 e2) =
    (size e1) + (size e2) + 1
size (Times e1 e2) =
    (size e1) + (size e2) + 1
```

In Scala, can define ASTs conveniently using `case classes`:

```scala
abstract class Expr

case class Num(n: Integer) extends Expr

case class Plus(e1: Expr, e2: Expr) extends Expr

case class Times(e1: Expr, e2: Expr) extends Expr
```

Again one can easily write functions to traverse them using pattern matching:

```scala
def size (e: Expr): Int = e match {
    case Num(n) => 1
    case Plus(e1,e2) =>
        size(e1) + size(e2) + 1
    case Times(e1,e2) =>
        size(e1) + size(e2) + 1
}
```

Java:

```java
new Plus(new Num(2), new Num(2))
```

Python:

```python
Plus(Num(2),Num(2))
```

Haskell:

```haskell
Plus(Num(2),Num(2))
```

Scala: (the “new” is optional for case classes):

```scala
new Plus(new Num(2),new Num(2))
new Plus(new Num(2),new Num(2))
new Plus(new Num(2),new Num(2))
```
Infix notation and operator precedence rules are convenient for programmers (looks like familiar math) but complicate language front-end. Some languages, notably LISP/Scheme/Racket, eschew infix notation. All programs are essentially so-called S-Expressions:

\[ s ::= a \mid (a_1 \cdots a_n) \]

so their concrete syntax is very close to abstract syntax.

For example

\[ 1 + 2 \quad \longrightarrow \quad (+ 1 2) \]
\[ 1 + 2 * 3 \quad \longrightarrow \quad (+ 1 (* 2 3)) \]
\[ (1 + 2) * 3 \quad \longrightarrow \quad (* (+ 1 2) 3) \]

The three most important reasoning techniques for programming languages are:

- (Mathematical) induction
  - (over \( \mathbb{N} \))
- (Structural) induction
  - (over ASTs)
- (Rule) induction
  - (over derivations)

We will briefly review the first and present structural induction. We will cover rule induction later.

Recall the principle of mathematical induction.

**Mathematical induction**

Given a property \( P \) of natural numbers, if:

- \( P(0) \) holds
- for any \( n \in \mathbb{N} \), if \( P(n) \) holds then \( P(n+1) \) also holds

Then \( P(n) \) holds for all \( n \in \mathbb{N} \).

A similar principle holds for expressions:

**Induction on structure of expressions**

Given a property \( P \) of expressions, if:

- \( P(n) \) holds for every number \( n \in \mathbb{N} \)
- for any expressions \( e_1, e_2 \), if \( P(e_1) \) and \( P(e_2) \) holds then \( P(e_1 + e_2) \) also holds
- for any expressions \( e_1, e_2 \), if \( P(e_1) \) and \( P(e_2) \) holds then \( P(e_1 \times e_2) \) also holds

Then \( P(e) \) holds for all expressions \( e \).

Note that we are performing induction over abstract syntax trees, not numbers!
Proof of expression induction principle

Define the size of an expression in the obvious way:

\[
\begin{align*}
    \text{size}(n) &= 1 \\
    \text{size}(e_1 + e_2) &= \text{size}(e_1) + \text{size}(e_2) + 1 \\
    \text{size}(e_1 \times e_2) &= \text{size}(e_1) + \text{size}(e_2) + 1 
\end{align*}
\]

Given \( P(\cdot) \) satisfying the assumptions of expression induction, we prove the property

\[
Q(n) = \text{for all } e \text{ with } \text{size}(e) < n \text{ we have } P(e)
\]

Since any expression \( e \) has a finite size, \( P(e) \) holds for any expression.

Proof.

We prove that \( Q(n) \) holds for all \( n \) by induction on \( n \):

- The base case \( n = 0 \) is vacuous
- For \( n + 1 \), then assume \( Q(n) \) holds and consider any \( e \) with \( \text{size}(e) < n + 1 \). Then there are three cases:
  - if \( e = m \in \mathbb{N} \) then \( P(e) \) holds by part 1 of expression induction principle
  - if \( e = e_1 + e_2 \) then \( \text{size}(e_1) < \text{size}(e) \leq n \) and similarly for \( \text{size}(e_2) < \text{size}(e) \leq n \). So, by induction, \( P(e_1) \) and \( P(e_2) \) hold, and by part 2 of expression induction principle \( P(e) \) holds.
  - if \( e = e_1 \times e_2 \), the same reasoning applies.

Summary

- We covered:
  - Concrete vs. Abstract syntax
  - Abstract syntax trees
  - Abstract syntax of L_{Arith} in several languages
  - Structural induction over syntax trees

- This might seem like a lot to absorb, but don’t worry! We will revisit and reinforce these concepts throughout the course.

- Next time:
  - Evaluation
  - A simple interpreter
  - Operational semantics rules