Elements of Programming Languages

Lecture 1: Abstract syntax

James Cheney

University of Edinburgh

September 19, 2024

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         ---> 3
1 + 2 * 3     ---> 7
(1 + 2) * 3 ---> 9
```

Concrete vs. abstract syntax

- **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.

Context-free grammars

- Context-free grammars give concrete syntax for expressions

$$
\begin{aligned}
E &\rightarrow E\ \texttt{PLUS}\ F \mid F \\
F &\rightarrow F\ \texttt{TIMES}\ F \mid \texttt{NUM} \mid \texttt{LPAREN}\ E\ \texttt{RPAREN}
\end{aligned}
$$

- Needs to handle precedence, parentheses, etc.
- Tokenization ($+ \rightarrow$ PLUS, etc.), comments, whitespace usually handled by a separate stage

BNF grammars

- BNF grammars give abstract syntax for expressions

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

- 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 $\times$ operator
  - Numbers $n \in \mathbb{N}$
- Much like CFG rules, we can "derive" more complex expressions:

$$e \to e_1 + e_2 \to 3 + e_2 \to 3 + (e_3 \times e_4) \to \cdots$$

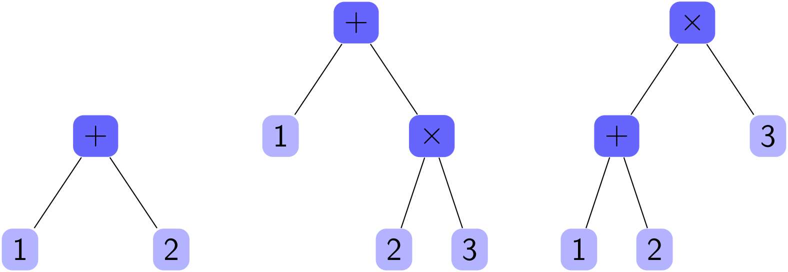
BNF conventions

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

$$(1+2)\times 3 \qquad \text{vs.} \qquad 1+(2\times 3)$$

Abstract Syntax Trees (ASTs)

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



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

Languages for examples

- We will use several languages for examples throughout the course:
  - Java: statically typed (ish), object-oriented
  - Python: dynamically typed, object-oriented with some functional features
  - Haskell: statically typed, functional
  - Scala: typed (ish), combines functional and OO features
  - Sometimes others, to discuss specific features (e.g. Rust, C)
- You do **not** need to already know all these languages!

ASTs in Java

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

```
abstract class Expr {}
class Num extends Expr {
  public int n;
  Num(int _n) {
    n = _n;
  }
}
```

ASTs in Java

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

```
...
class Plus extends Expr {
  public Expr e1;
  public Expr e2;
  Plus(Expr _e1, Expr _e2) {
    e1 = _e1;
    e2 = _e2;
  }
}
class Times extends Expr {... // similar
}
```

ASTs in Java

- Traverse ASTs by adding a method to each class:

```
abstract class Expr {
  abstract public int size();
}
class Num extends Expr { ...
  public int size() { return 1;}
}
class Plus extends Expr { ...
  public int size() {
    return e1.size() + e2.size() + 1;
  }
}
class Times extends Expr {... // similar
}
```

ASTs in Python

- Python is similar, but shorter (using dataclasses):

```
class Expr:
    pass # "abstract"
@dataclass
class Num(Expr):
    n: int
    def size(self): return 1
@dataclass
class Plus(Expr):
    e1: Expr
    e2: Expr
    def size(self):
        return self.e1.size() + self.e2.size() + 1
class Times(Expr): # similar...
```

ASTs in Haskell

- In Haskell, ASTs are easily defined as *datatypes*:

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

- Likewise one can easily write functions to traverse them:

```
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
```

ASTs in Scala

- In Scala, can define ASTs conveniently using *case classes*:

```
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:

```
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
}
```

Creating ASTs

- Java:

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

- Python:

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

- Haskell:

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

- Scala: (the "new" is optional for case classes:)

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

Precedence, Parentheses and Parsimony

- 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:

$$s ::= a \mid (a\ s_1\ \cdots\ s_n)$$

  so their concrete syntax is very close to abstract syntax.
- For example

```
1 + 2          ---> (+ 1 2)
1 + 2 * 3      ---> (+ 1 (* 2 3))
(1 + 2) * 3    ---> (* (+ 1 2) 3)
```

The three most important reasoning techniques

- The three most important reasoning techniques for programming languages are:
  - (Mathematical) induction
  - (Structural) induction
  - (Rule) induction
- We will briefly review the first and present structural induction.
- We will cover rule induction later.

The three most important reasoning techniques

- The three most important reasoning techniques for programming languages are:
  - (Mathematical) induction
    - (over $\mathbb{N}$)
  - (Structural) induction
  - (Rule) induction
- We will briefly review the first and present structural induction.
- We will cover rule induction later.

The three most important reasoning techniques

- The three most important reasoning techniques for programming languages are:
  - (Mathematical) induction
    - (over $\mathbb{N}$)
  - (Structural) induction
    - (over ASTs)
  - (Rule) induction
- We will briefly review the first and present structural induction.
- We will cover rule induction later.

The three most important reasoning techniques

- 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.

Induction

- 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 $n \in \mathbb{N}$.

Induction over expressions

- 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:

$$\begin{aligned} size(n) &= 1 \\ size(e_1 + e_2) &= size(e_1) + size(e_2) + 1 \\ size(e_1 \times e_2) &= size(e_1) + size(e_2) + 1 \end{aligned}$$

Given $P(-)$ satisfying the assumptions of expression induction, we need to use induction over $\mathbb{N}$ to show $P(e)$ holds for any e. We will use $\mathbb{N}$-induction to prove $Q(n)$ for any n where:

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

Since any expression e has a finite size, $P(e)$ holds for any expression because $Q(size(e) + 1)$ holds and implies $P(e)$.

Proof of expression induction principle

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 $size(e) < n + 1$. 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 $size(e_1) < size(e) \leq n$ and similarly for $size(e_2) < size(e) \leq n$. 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