Lecture 2 – Syntax and Semantics (1)

COSE212: Programming Languages

Jihyeok Park

PLRG

2023 Fall

Recall

We learn language features of **Scala**:

- **Basic Features**
  - Built-in Data Types
  - Variables
  - Functions
  - Conditionals
- **Object-Oriented Programming (OOP)**
  - Case Classes
- **Algebraic Data Types (ADTs)**
  - Pattern Matching
- **Functional Programming (FP)**
  - First-class Functions
  - Recursion
- **Immutable Collections**
  - Lists
  - Options and Pairs
  - Maps and Sets
  - For Comprehensions

Definition (Programming Language)

A **programming language** is defined by

- **Syntax**: a grammar that defines the structure of programs
- **Semantics**: a set of rules that defines the meaning of programs

We will learn how to define the **syntax** and **semantics** of a programming language.

We define a programming language for **arithmetic expressions** (AE) as the running example.

Arithmetic Expressions

Let's consider the arithmetic expressions (AE) supporting **addition** and **multiplication** of integers:

- `4 + 2`
- `1 * 24`
- `-42 + 4 * 10`
- `(1 + 2) * (2 + 3)`
- ...

There are **infinitely many** AEs.

How to define all the valid AEs (**syntax**)?

How to define the expected result of each AE (**semantics**)?

Contents

Contents

Backus-Naur Form (BNF)

Backus-Naur Form (BNF) is a notation for **context-free grammar**:

- A **nonterminal** has a name and a set of **production rules** consisting of sequences of terminals and nonterminals.
- A **terminal** is a symbol that appears in the final output.

For example, a nonterminal <number> produces all strings representing integers (allowing leading zeros) as follows:

```
<digit>  ::= "0" | "1" | "2" | "3" | "4"
           | "5" | "6" | "7" | "8" | "9"

<nat>    ::= <digit> | <digit> <nat>

<number> ::= <nat> | "-" <nat>
```

Concrete Syntax

Let's define the **concrete syntax** of AE in BNF:

```
<expr> ::= <number>
         | <expr> "+" <expr>
         | <expr> "*" <expr>
         | "(" <expr> ")"
```

It is the **surface-level** representation of programs with all the syntactic details to decide whether a given string is a valid AE or not.

For example, `(1+2)*3` is a valid AE:

```
<expr> ⇒ <expr>*<expr>                ⇒ (<expr>)*<expr>
       ⇒ (<expr>+<expr>)*<expr>       ⇒ (<number>+<expr>)*<expr>
       ⇒ (1+<expr>)*<expr>            ⇒ (1+<number>)*<expr>
       ⇒ (1+2)*<expr>                 ⇒ (1+2)*<number>
       ⇒ (1+2)*3
```

Concrete Syntax

Let's define the **concrete syntax** of AE in BNF:

```
<expr> ::= <number>
         | <expr> "+" <expr>
         | <expr> "*" <expr>
         | "(" <expr> ")"
```

We need **associativity** and **precedence** rules to disambiguate.

- "+" and "*" are **left-associative**.

```
"1 + 2 + 3 + 4 + 5"   ==   "((((1 + 2) + 3) + 4) + 5)"
"1 * 2 * 3 * 4 * 5"   ==   "((((1 * 2) * 3) * 4) * 5)"
```

- "*" has higher **precedence** than "+".

```
"1 + 2 * 3 + 4 * 5"   ==   "((1 + (2 * 3)) + (4 * 5))"
```

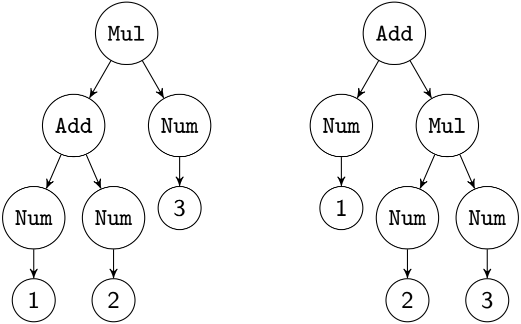
Abstract Syntax

Let's define the **abstract syntax** of AE in BNF:

$$
\begin{array}{rcll}
e & ::= & n & (\texttt{Num}) \\
  & \mid & e + e & (\texttt{Add}) \\
  & \mid & e \times e & (\texttt{Mul})
\end{array}
$$

It captures only the essential structure of AE rather than the details.

The **abstract syntax trees (ASTs)** of `(1+2)*3` and `1+2*3` are as follows:



Concrete vs. Abstract Syntax

While **concrete syntax** is the **surface-level** representation of programs, **abstract syntax** is the **essential** representation of programs.

There might be **multiple** concrete syntax for the **same** abstract syntax:

```
<expr> ::= <number>
         | <expr> "+" <expr>
         | <expr> "*" <expr>
         | "(" <expr> ")"
```

```
<expr> ::= <number>
         | "(" "+" <expr> <expr> ")"
         | "(" "*" <expr> <expr> ")"
```

```
<expr> ::= <number>
         | "ADD[" <expr> ";" <expr> "]"
         | "MUL[" <expr> ";" <expr> "]"
```

$$
\begin{array}{rcll}
e & ::= & n & (\texttt{Num}) \\
 & \mid & e + e & (\texttt{Add}) \\
 & \mid & e \times e & (\texttt{Mul})
\end{array}
$$

Concrete vs. Abstract Syntax

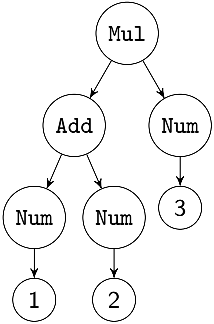
While **concrete syntax** is the **surface-level** representation of programs, **abstract syntax** is the **essential** representation of programs.

There might be **multiple** concrete syntax for the **same** abstract syntax:

$(1 + 2) * 3$

$(*(+12)3)$ $\Rightarrow$

$\texttt{MUL}[\texttt{ADD}[1;2];3]$



Contents

Semantics

There exist diverse ways to define **semantics** of programming languages.

- **Axiomatic semantics** defines the meaning of a program by specifying the properties that hold after its execution.

$$\{x = n \wedge y = m\}\ \mathtt{z := x + y}\ \{z = n + m\}$$

- **Denotational semantics** defines the meaning of a program by mapping it to a mathematical object that represents its meaning.

$$[\![e + e]\!] = [\![e]\!] + [\![e]\!]$$

- **Operational semantics** defines the meaning of a program by specifying how it executes on a machine.

$$\frac{\vdash e_1 \Rightarrow n_1 \qquad \vdash e_2 \Rightarrow n_2}{\vdash e_1 + e_2 \Rightarrow n_1 + n_2}$$

- . . .

Operational Semantics

In this course, we will focus on **operational semantics**, and there are two different representative styles:

- **Big-Step Operational (Natural) Semantics** defines the meaning of a program by specifying how it executes on a machine in one big step.

$$\frac{\vdash e_1 \Rightarrow n_1 \qquad \vdash e_2 \Rightarrow n_2}{\vdash e_1 + e_2 \Rightarrow n_1 + n_2}$$

- **Small-Step Operational (Reduction) Semantics** defines the meaning of a program by specifying how it executes on a machine step-by-step.

$$\frac{e_1 \rightarrow e_1'}{e_1 + e_2 \rightarrow e_1' + e_2}$$

Inference Rules

Operational semantics is defined by **inference rules**.

An **inference rule** consists of multiple **premises** and one **conclusion**:

$$\frac{\mathit{premise}_1 \qquad \mathit{premise}_2 \qquad \cdots \qquad \mathit{premise}_n}{\mathit{conclusion}}$$

meaning that "if all the premises are true, then the conclusion is true":

$$\mathit{premise}_1 \land \mathit{premise}_2 \land \cdots \land \mathit{premise}_n \implies \mathit{conclusion}$$

For example,

$$\frac{A \implies B \qquad B \implies C}{A \implies C}$$

means that "if A implies B, and B implies C, then A implies C".

Big-Step Operational (Natural) Semantics

$$\boxed{\vdash e \Rightarrow n}$$

It means that *"the expression e evaluates to the number n"*.

Let's define the **big-step operational (natural) semantics** of AE:

$$\begin{array}{rrll} e & ::= & n & (\texttt{Num}) \\ & \mid & e + e & (\texttt{Add}) \\ & \mid & e \times e & (\texttt{Mul}) \end{array} \quad \Longrightarrow$$

$$\texttt{Num}\ \frac{}{\vdash n \Rightarrow n}$$

$$\texttt{Add}\ \frac{\vdash e_1 \Rightarrow n_1 \qquad \vdash e_2 \Rightarrow n_2}{\vdash e_1 + e_2 \Rightarrow n_1 + n_2}$$

$$\texttt{Mul}\ \frac{\vdash e_1 \Rightarrow n_1 \qquad \vdash e_2 \Rightarrow n_2}{\vdash e_1 \times e_2 \Rightarrow n_1 \times n_2}$$

Big-Step Operational (Natural) Semantics

$$\texttt{Num}\ \frac{}{\vdash n \Rightarrow n} \qquad \texttt{Add}\ \frac{\vdash e_1 \Rightarrow n_1 \qquad \vdash e_2 \Rightarrow n_2}{\vdash e_1 + e_2 \Rightarrow n_1 + n_2} \qquad \texttt{Mul}\ \frac{\vdash e_1 \Rightarrow n_1 \qquad \vdash e_2 \Rightarrow n_2}{\vdash e_1 \ \times e_2 \Rightarrow n_1 \times n_2}$$

Let's prove $\vdash (1+2)*3 \Rightarrow 9$ by drawing a **derivation tree**:

$$\texttt{Mul}\ \frac{\texttt{Add}\ \dfrac{\texttt{Num}\ \dfrac{}{\vdash 1 \Rightarrow 1} \quad \texttt{Num}\ \dfrac{}{\vdash 2 \Rightarrow 2}}{\vdash 1 + 2 \Rightarrow 3} \quad \texttt{Num}\ \dfrac{}{\vdash 3 \Rightarrow 3}}{\vdash (1+2)*3 \Rightarrow 9}$$

Let's prove $\vdash 1 + 2 * 3 \Rightarrow 7$ by drawing a **derivation tree**:

$$\frac{}{\vdash 1 + 2 * 3 \Rightarrow}$$

Small-Step Operational (Reduction) Semantics

$$\boxed{e_0 \rightarrow e_1}$$

It means that *"e_0 is reduced to e_1 as the result of one-step evaluation"*.

Let's define the **small-step operational (reduction) semantics** of AE:

$$\begin{array}{rcll} e & ::= & n & (\texttt{Num}) \\ & \mid & e + e & (\texttt{Add}) \\ & \mid & e \times e & (\texttt{Mul}) \end{array} \implies$$

$$\frac{e_1 \rightarrow e_1'}{e_1 + e_2 \rightarrow e_1' + e_2} \qquad \frac{e_1 \rightarrow e_1'}{e_1 \times e_2 \rightarrow e_1' \times e_2}$$

$$\frac{e_2 \rightarrow e_2'}{n_1 + e_2 \rightarrow n_1 + e_2'} \qquad \frac{e_2 \rightarrow e_2'}{n_1 \times e_2 \rightarrow n_1 \times e_2'}$$

$$\frac{}{n_1 + n_2 \rightarrow n_1 + n_2} \qquad \frac{}{n_1 \times n_2 \rightarrow n_1 \times n_2}$$

Small-Step Operational (Reduction) Semantics

$$\frac{e_1 \to e_1'}{e_1 + e_2 \to e_1' + e_2} \qquad \frac{e_2 \to e_2'}{n_1 + e_2 \to n_1 + e_2'} \qquad \frac{}{n_1 + n_2 \to n_1 + n_2}$$

$$\frac{e_1 \to e_1'}{e_1 \times e_2 \to e_1' \times e_2} \qquad \frac{e_2 \to e_2'}{n_1 \times e_2 \to n_1 \times e_2'} \qquad \frac{}{n_1 \times n_2 \to n_1 \times n_2}$$

Let's prove $(1 + 2) * 3 \to^* 9$ by showing a **reduction sequence**:

(Note that $\to^*$ denotes the reflexive-transitive closure of $\to$.)

$$(1 + 2) * 3 \quad \to \quad 3 * 3 \quad \to \quad 9$$

Let's prove $1 + 2 * 3 \to^* 7$ by showing a **reduction sequence**:

$$1 + 2 * 3 \quad \to$$

Summary

1. Syntax
   - Backus-Naur Form (BNF)
   - Concrete Syntax
   - Abstract Syntax
   - Concrete vs. Abstract Syntax
2. Operational Semantics
   - Inference Rules
   - Big-Step Operational (Natural) Semantics
   - Small-Step Operational (Reduction) Semantics

Next Lecture

- Syntax and Semantics (2)

Jihyeok Park
jihyeok_park@korea.ac.kr
https://plrg.korea.ac.kr