

Lecture 2 – Syntax and Semantics (1)

COSE212: Programming Languages

Jihyeok Park



2024 Fall

- Before entering the world of PL, we learned the basics of **Scala** language in the previous lecture.
- In this course, you will learn how to:
 - **design** programming languages in a **mathematical** way.
 - **implement** their **interpreters** using **Scala**.
- We will grow a programming language from arithmetic expressions (AE) into a more complex language by adding more features.
- In this lecture, we will learn how to **design** a programming language in a **mathematical** way.

1. Programming Languages

2. Syntax

- Concrete Syntax

- Abstract Syntax

- Concrete vs. Abstract Syntax

3. Operational Semantics

- Inference Rules

- Big-Step Operational (Natural) Semantics

- Small-Step Operational (Reduction) Semantics

1. Programming Languages

2. Syntax

Concrete Syntax

Abstract Syntax

Concrete vs. Abstract Syntax

3. Operational Semantics

Inference Rules

Big-Step Operational (Natural) Semantics

Small-Step Operational (Reduction) Semantics

Definition (Programming Language)

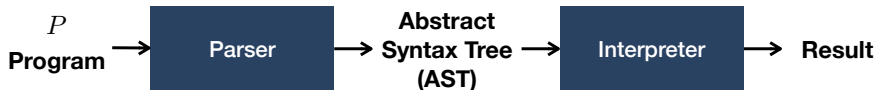
A **programming language** is defined by

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

Definition (Programming Language)

A **programming language** is defined by

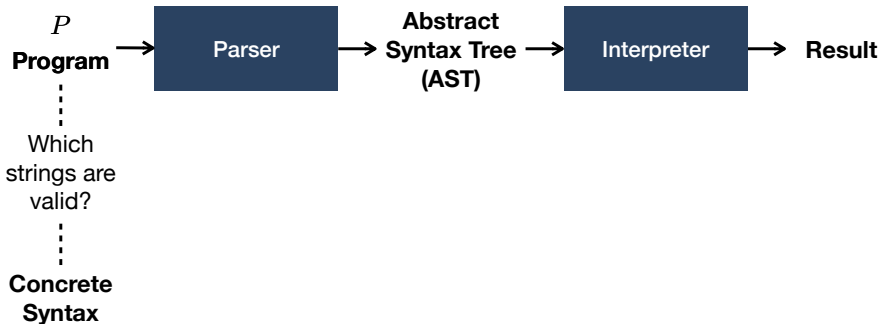
- **Syntax**: a grammar that defines the **structure** of programs
- **Semantics**: a set of rules that defines the **meaning** of programs



Definition (Programming Language)

A **programming language** is defined by

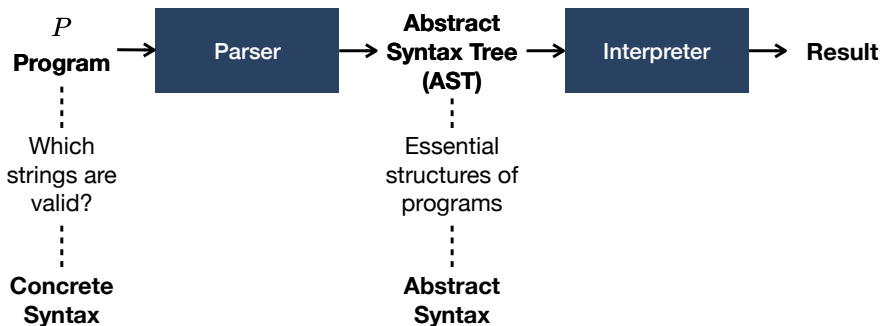
- **Syntax**: a grammar that defines the **structure** of programs
- **Semantics**: a set of rules that defines the **meaning** of programs



Definition (Programming Language)

A **programming language** is defined by

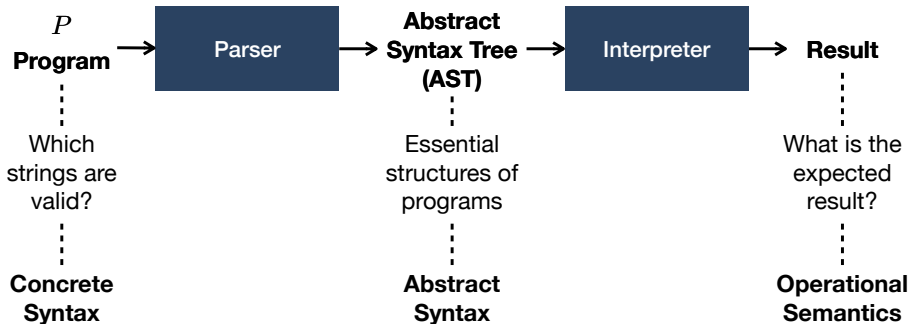
- **Syntax**: a grammar that defines the **structure** of programs
- **Semantics**: a set of rules that defines the **meaning** of programs



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



For example, let's consider the arithmetic expressions (AE) supporting **addition** and **multiplication** of number (integer) values.

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

There are **infinitely many** AEs.

Which strings are valid AEs? – (**concrete syntax**)

What does parsing result of each AE look like? – (**abstract syntax**)

What is the evaluation result of each AE? – (**operational semantics**)

1. Programming Languages

2. Syntax

- Concrete Syntax

- Abstract Syntax

- Concrete vs. Abstract Syntax

3. Operational Semantics

- Inference Rules

- Big-Step Operational (Natural) Semantics

- Small-Step Operational (Reduction) Semantics

We use a variant of the **extended Backus-Naur form (EBNF)** to define the concrete/abstract syntax of programming languages.

We use the different notation for concrete and abstract syntax:

Description	Concrete Syntax	Abstract Syntax
Terminal	"a"	a
Nonterminal	<expr>	<i>e</i>
Optional	<expr>?	<i>e</i> [?]
Zero or more repetition	<expr>*	<i>e</i> [*]
One or more repetition	<expr>+	<i>e</i> ⁺

For example, we can define a concrete syntax of integers as follows:

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

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:

$\langle \text{expr} \rangle \Rightarrow \langle \text{expr} \rangle * \langle \text{expr} \rangle$	$\Rightarrow (\langle \text{expr} \rangle) * \langle \text{expr} \rangle$
$\Rightarrow (\langle \text{expr} \rangle + \langle \text{expr} \rangle) * \langle \text{expr} \rangle$	$\Rightarrow (\langle \text{number} \rangle + \langle \text{expr} \rangle) * \langle \text{expr} \rangle$
$\Rightarrow (1 + \langle \text{expr} \rangle) * \langle \text{expr} \rangle$	$\Rightarrow (1 + \langle \text{number} \rangle) * \langle \text{expr} \rangle$
$\Rightarrow (1 + 2) * \langle \text{expr} \rangle$	$\Rightarrow (1 + 2) * \langle \text{number} \rangle$
$\Rightarrow (1 + 2) * 3$	

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

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

We need **associativity** and **precedence** rules to remove ambiguity:

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

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

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

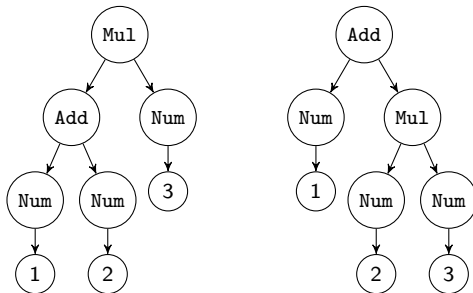
```
"1 + 2 * 3" == "1 + (2 * 3)"
```

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

Numbers	$n \in \mathbb{Z}$	(BigInt)
Expressions	$e ::= n$	(Num)
	$e + e$	(Add)
	$e * e$	(Mul)

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":



While **concrete syntax** is the **surface-level** representation of programs, **abstract syntax** captures the **essential structure** 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> "]"
```

$$\begin{array}{ll} n \in \mathbb{Z} & (\text{BigInt}) \\ e ::= n & (\text{Num}) \\ \quad | e + e & (\text{Add}) \\ \quad | e * e & (\text{Mul}) \end{array}$$

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

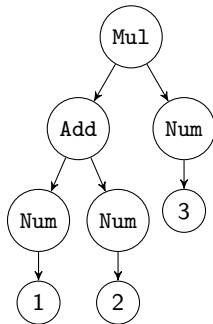
There might be **multiple** concrete syntax for the **same** abstract syntax:

$(1 + 2) * 3$

$(* (+ 1 2) 3)$

\Rightarrow

MUL[ADD[1; 2]; 3]



1. Programming Languages

2. Syntax

Concrete Syntax

Abstract Syntax

Concrete vs. Abstract Syntax

3. Operational Semantics

Inference Rules

Big-Step Operational (Natural) Semantics

Small-Step Operational (Reduction) 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.

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

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

$$\llbracket e + e \rrbracket = \llbracket e \rrbracket + \llbracket e \rrbracket$$

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

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

- ...

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.

$$\frac{\dots}{\vdash e \Rightarrow n}$$

(The execution result of an expression e is n because of)

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

$$e \rightarrow e' \rightarrow e'' \rightarrow \dots \rightarrow n$$

(An expression e is reduced to e' , then to e'' , and so on until n .)

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

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

$$\frac{\text{premise}_1 \quad \text{premise}_2 \quad \dots \quad \text{premise}_n}{\text{conclusion}}$$

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

$$\text{premise}_1 \wedge \text{premise}_2 \wedge \dots \wedge \text{premise}_n \implies \text{conclusion}$$

For example,

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

means that “if A implies B , and B implies C , then A implies C ”.

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

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

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

$$\begin{array}{lcl}
 e ::= n & \text{(Num)} & \\
 | e + e & \text{(Add)} & \\
 | e * e & \text{(Mul)} &
 \end{array}
 \quad \Rightarrow \quad
 \begin{array}{c}
 \text{NUM} \frac{}{\vdash n \Rightarrow n} \\
 \\
 \text{ADD} \frac{\vdash e_1 \Rightarrow n_1 \quad \vdash e_2 \Rightarrow n_2}{\vdash e_1 + e_2 \Rightarrow n_1 + n_2} \\
 \\
 \text{MUL} \frac{\vdash e_1 \Rightarrow n_1 \quad \vdash e_2 \Rightarrow n_2}{\vdash e_1 * e_2 \Rightarrow n_1 \times n_2}
 \end{array}$$

$$\text{NUM} \frac{}{\vdash n \Rightarrow n} \quad \text{ADD} \frac{\vdash e_1 \Rightarrow n_1 \quad \vdash e_2 \Rightarrow n_2}{\vdash e_1 + e_2 \Rightarrow n_1 + n_2} \quad \text{MUL} \frac{\vdash e_1 \Rightarrow n_1 \quad \vdash e_2 \Rightarrow n_2}{\vdash e_1 * e_2 \Rightarrow n_1 \times n_2}$$

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

$$\begin{array}{c} \text{NUM} \frac{}{\vdash 1 \Rightarrow 1} \quad \text{NUM} \frac{}{\vdash 2 \Rightarrow 2} \\ \text{ADD} \frac{}{\vdash 1 + 2 \Rightarrow 3} \quad \text{NUM} \frac{}{\vdash 3 \Rightarrow 3} \\ \text{MUL} \frac{}{\vdash (1 + 2) * 3 \Rightarrow 9} \end{array}$$

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

$$\vdash 1 + (2 * 3) \Rightarrow$$

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

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:

$$\begin{array}{lcl}
 e ::= & n & \text{(Num)} \\
 & | & e + e \quad \text{(Add)} \\
 & | & e * e \quad \text{(Mul)}
 \end{array}
 \implies
 \begin{array}{c}
 \frac{e_1 \rightarrow e'_1}{e_1 + e_2 \rightarrow e'_1 + e_2} \qquad \frac{e_1 \rightarrow e'_1}{e_1 * e_2 \rightarrow e'_1 * e_2} \\
 \\
 \frac{e_2 \rightarrow e'_2}{n_1 + e_2 \rightarrow n_1 + e'_2} \qquad \frac{e_2 \rightarrow e'_2}{n_1 * e_2 \rightarrow n_1 * e'_2} \\
 \\
 \frac{}{n_1 + n_2 \rightarrow n_1 + n_2} \qquad \frac{}{n_1 * n_2 \rightarrow n_1 \times n_2}
 \end{array}$$

$$\frac{e_1 \rightarrow e'_1}{e_1 + e_2 \rightarrow e'_1 + e_2}$$

$$\frac{e_2 \rightarrow e'_2}{n_1 + e_2 \rightarrow n_1 + e'_2}$$

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

$$\frac{e_1 \rightarrow e'_1}{e_1 * e_2 \rightarrow e'_1 * e_2}$$

$$\frac{e_2 \rightarrow e'_2}{n_1 * e_2 \rightarrow n_1 * e'_2}$$

$$\frac{}{n_1 * n_2 \rightarrow n_1 \times n_2}$$

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

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

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

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

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

1. Programming Languages

2. Syntax

- Concrete Syntax

- Abstract Syntax

- Concrete vs. Abstract Syntax

3. Operational Semantics

- Inference Rules

- Big-Step Operational (Natural) Semantics

- Small-Step Operational (Reduction) Semantics

(See the language specification of AE.¹)

¹<https://github.com/ku-plrg-classroom/docs/blob/main/cose212/ae/ae-spec.pdf>

- Syntax and Semantics (2)

Jihyeok Park

`jihyeok_park@korea.ac.kr`

`https://plrg.korea.ac.kr`