Lecture 2 – Syntax and Semantics (1)

COSE212: Programming Languages

Jihyeok Park

2025 Fall

Recall

- 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.

Contents

Contents

Programming Languages

Definition (Programming Language)

A **programming language** is defined by

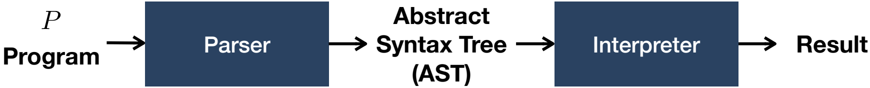
- **Syntax**: a grammar that defines the **structure** of programs
- **Semantics**: a set of rules that defines the **meaning** of programs

Programming Languages

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

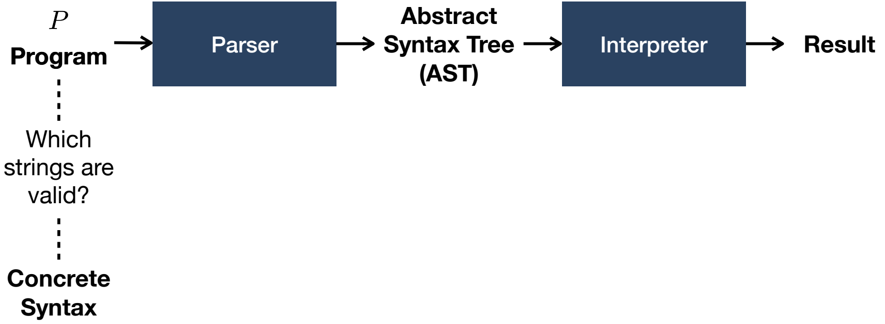


Programming Languages

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

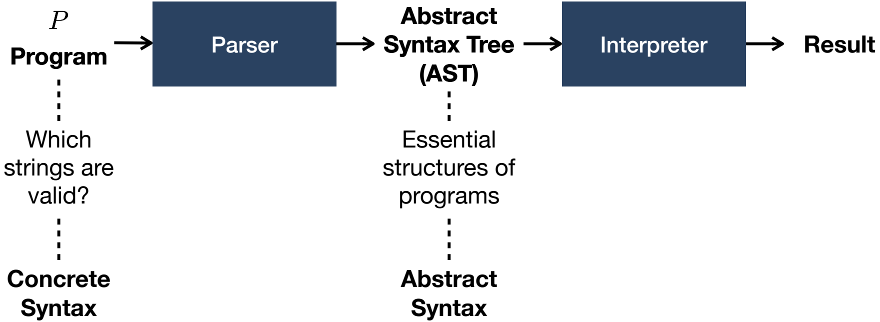


Programming Languages

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

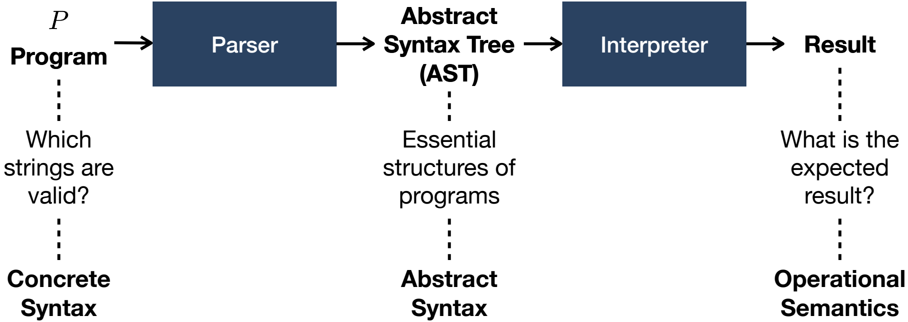


Programming Languages

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



Example – Arithmetic Expressions

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**)

Contents

Extended Backus-Naur Form (EBNF)

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>`	e
Optional	`<expr>?`	$e^?$
Zero or more repetition	`<expr>*`	e^*
One or more repetition	`<expr>+`	e^+

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>+
```

Concrete Syntax

PLRG

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 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)"
```

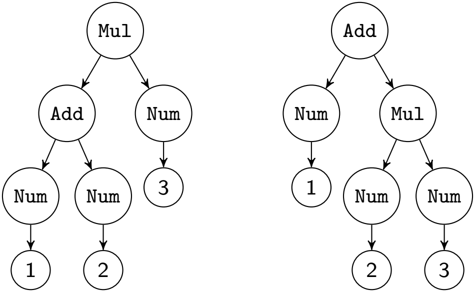
Abstract Syntax

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

$$
\begin{array}{llcll}
\text{Numbers} & n & \in & \mathbb{Z} & (\texttt{BigInt}) \\
\text{Expressions} & e & ::= & n & (\texttt{Num}) \\
 & & \mid & e \texttt{ + } e & (\texttt{Add}) \\
 & & \mid & e \texttt{ * } e & (\texttt{Mul})
\end{array}
$$

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"`:



Concrete vs. Abstract Syntax

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> "]"
```

$$
\begin{array}{rcll}
n & \in & \mathbb{Z} & (\texttt{BigInt}) \\
e & ::= & n & (\texttt{Num}) \\
 & \mid & e \texttt{ + } e & (\texttt{Add}) \\
 & \mid & e \texttt{ * } e & (\texttt{Mul})
\end{array}
$$

Concrete vs. Abstract Syntax

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

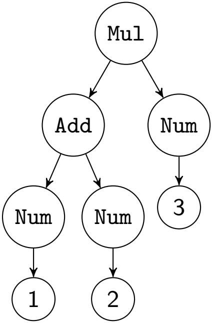
There might be **multiple** concrete syntax for the **same** abstract syntax:

```
(1 + 2) * 3

(* (+ 1 2) 3)

MUL[ADD[1; 2]; 3]
```

$\Rightarrow$



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.

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

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

$$[\![e_1\ \texttt{+}\ e_2]\!] = [\![e_1]\!] + [\![e_2]\!]$$

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

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

- ...

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**.

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

  (The execution result of an expression e is n because of $\ldots$.)

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

  $$e \rightarrow e' \rightarrow e'' \rightarrow \ldots \rightarrow n$$

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

Inference Rules

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

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

$$\frac{\textit{premise}_1 \qquad \textit{premise}_2 \qquad \ldots \qquad \textit{premise}_n}{\textit{conclusion}}$$

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

$$\textit{premise}_1 \wedge \textit{premise}_2 \wedge \ldots \wedge \textit{premise}_n \implies \textit{conclusion}$$

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"*.
(Syllogism – 삼단논법)

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

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

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

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

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

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

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

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

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

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

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

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

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:

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

$$\frac{e_1 \rightarrow e_1'}{e_1 \texttt{ + } e_2 \rightarrow e_1' \texttt{ + } e_2} \qquad \frac{e_1 \rightarrow e_1'}{e_1 \texttt{ * } e_2 \rightarrow e_1' \texttt{ * } e_2}$$

$$\frac{e_2 \rightarrow e_2'}{n_1 \texttt{ + } e_2 \rightarrow n_1 \texttt{ + } e_2'} \qquad \frac{e_2 \rightarrow e_2'}{n_1 \texttt{ * } e_2 \rightarrow n_1 \texttt{ * } e_2'}$$

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

Small-Step Operational (Reduction) Semantics

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

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

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

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

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

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

$$1 \texttt{ + } 2 \texttt{ * } 3 \quad \to$$

Summary

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

Next Lecture

- Syntax and Semantics (2)

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