

# Lecture 3 – Syntax and Semantics (2)

## COSE212: Programming Languages

Jihyeok Park



2023 Fall

We learn how to define **syntax** and **semantics** of a programming language with (AE) as an example.

- **Syntax**

- Backus-Naur Form (BNF)
- Concrete Syntax
- Abstract Syntax
- Concrete vs. Abstract Syntax

- **Semantics**

- Inference Rules
- Big-Step Operational (Natural) Semantics
- Small-Step Operational (Reduction) Semantics

We learn how to define **syntax** and **semantics** of a programming language with (AE) as an example.

- **Syntax**

- Backus-Naur Form (BNF)
- Concrete Syntax
- Abstract Syntax
- Concrete vs. Abstract Syntax

- **Semantics**

- Inference Rules
- Big-Step Operational (Natural) Semantics
- Small-Step Operational (Reduction) Semantics

In this lecture, we learn how to implement the **interpreter** for AE.

- **Parser**: from **strings** to **abstract syntax trees (ASTs)**
- **Interpreter**: from **ASTs** to **values**

## 1. Parsers

- ADTs for Abstract Syntax
- Parsers for Concrete Syntax

## 2. Interpreters

## 1. Parsers

- ADTs for Abstract Syntax
- Parsers for Concrete Syntax

## 2. Interpreters

Let's define Scala **ADTs** to represent the **abstract syntax** of AE.

$$\begin{array}{lcl} e & ::= & n \quad (\text{Num}) \\ & | & e + e \quad (\text{Add}) \\ & | & e * e \quad (\text{Mul}) \end{array}$$

Let's define Scala **ADTs** to represent the **abstract syntax** of AE.

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

```
// expressions
enum Expr:
  // numbers
  case Num(number: BigInt) // `BigInt` rather than `Int` for integers
  // additions
  case Add(left: Expr, right: Expr)
  // multiplications
  case Mul(left: Expr, right: Expr)
```

Let's define Scala **ADTs** to represent the **abstract syntax** of AE.

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

```
// expressions
enum Expr:
  // numbers
  case Num(number: BigInt) // `BigInt` rather than `Int` for integers
  // additions
  case Add(left: Expr, right: Expr)
  // multiplications
  case Mul(left: Expr, right: Expr)
```

For example, the AE expressions  $1 + 2 * 3$  is represented as follows:

```
Add(Num(1), Mul(Num(2), Num(3)))
```



We learned the **concrete syntax** of AE in the last lecture.

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

Then, how can we implement a **parser** for AE?

---

<sup>1</sup><https://github.com/scala/scala-parser-combinators>

<sup>2</sup>[https://en.wikipedia.org/wiki/Parsing\\_expression\\_grammar](https://en.wikipedia.org/wiki/Parsing_expression_grammar)

We learned the **concrete syntax** of AE in the last lecture.

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

Then, how can we implement a **parser** for AE?

Let's use **parser combinators** in Scala!

I will explain basic ideas of parser combinators in this lecture. If you are interested in details, please refer to here<sup>1</sup>, and **parsing expression grammars (PEGs)**.<sup>2</sup>

---

<sup>1</sup><https://github.com/scala/scala-parser-combinators>

<sup>2</sup>[https://en.wikipedia.org/wiki/Parsing\\_expression\\_grammar](https://en.wikipedia.org/wiki/Parsing_expression_grammar)

What can we do with **parser combinators** in Scala?

What can we do with **parser combinators** in Scala?

- We can use **regular expressions** ("...".r) as parsers
- We can **combine** them using ~ (sequence) and | (alternative)
- We can **transform** the result of a parser using ^^.

What can we do with **parser combinators** in Scala?

- We can use **regular expressions** ("**...**".r) as parsers
- We can **combine** them using ~ (sequence) and | (alternative)
- We can **transform** the result of a parser using ^^.

For example, let's implement a parser for **list of integers**:

```
type P[+T] = PackratParser[T]
lazy val num : P[BigInt] = "-?[0-9]+".r ^^ { BigInt(_) }
lazy val numSeq: P[List[BigInt]] =
  (num <~ ",") ~ numSeq ^^ { case x ~ xs => x :: xs } |
  num ^^ { case x => List(x) } |
  "" ^^ { case _ => Nil }
lazy val list : P[List[BigInt]] = "[" ~> numSeq <~ "]"

parseAll(num, "-042").get // -042 : BigInt
parseAll(numSeq, "-042, 4, 20").get // List(-42, 4, 20) : List[BigInt]
parseAll(list, "[-042, 4, 20]").get // List(-42, 4, 20) : List[BigInt]
```

What can we do with **parser combinators** in Scala?

- We can use **regular expressions** ("**...**".r) as parsers
- We can **combine** them using ~ (sequence) and | (alternative)
- We can **transform** the result of a parser using ^^.

For example, let's implement a parser for **list of integers**:

```
type P[+T] = PackratParser[T]

lazy val num    : P[BigInt]      = "-?[0-9]+".r ^^ { BigInt(_) }
lazy val numSeq : P[List[BigInt]] = rep1sep(num, ",")
lazy val list   : P[List[BigInt]] = "[" ~> numSeq <~ "]"

parseAll(num,      "-042").get           // -042                : BigInt
parseAll(numSeq,   "-042, 4, 20").get    // List(-42, 4, 20) : List[BigInt]
parseAll(list,     "[-042, 4, 20]").get  // List(-42, 4, 20) : List[BigInt]
```

We can simplify it using rep1sep (repeat one or more times separated by ","). There are other helper functions that help us write parsers.

Let's implement a **parser** for AE using Scala **parser combinators**.

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

Let's implement a **parser** for AE using Scala **parser combinators**.

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

```
lazy val num: P[BigInt] = "-?[0-9]+".r ^^ { BigInt(_) }
lazy val expr: P[Expr] =
  import Expr.*
  lazy val e0: P[Expr] = (e0<~"+">~e1 ^^ { case l~r => Add(l, r) } | e1
  lazy val e1: P[Expr] = (e1<~"*">~e2 ^^ { case l~r => Mul(l, r) } | e2
  lazy val e2: P[Expr] = num ^^ { case n => Num(n) } | e3
  lazy val e3: P[Expr] = "(" ~> e0 <~ ")"
  e0

parseAll(expr, "42").get // Num(42) : Expr
parseAll(expr, "-1 + 7").get // Add(Num(-1),Num(7)) : Expr
parseAll(expr, "1 + 2 * 3").get // Add(Num(1),Mul(Num(2),Num(3))) : Expr
```



You **don't need to know** the details of parser combinators.

We **provide all parsers** of programming languages in this course.

If you want to use the parser, please just call `Expr` as follows:

```
val x: Expr = Expr("42")           // Num(42)           : Expr
val y: Expr = Expr("-1 + 7")       // Add(Num(-1),Num(7)) : Expr
val z: Expr = Expr("1 + 2 * 3")    // Add(Num(1),Mul(Num(2),Num(3))) : Expr
```

If you want to get the **string representation** of the expression, please use `getString` method as follows: method.

```
x.getString // "42"           : String
y.getString // "(-1 + 7)"      : String
z.getString // "(1 + (2 * 3))" : String
```

You **don't need to know** the details of parser combinators.

We **provide all parsers** of programming languages in this course.

If you want to use the parser, please just call `Expr` as follows:

```
val x: Expr = Expr("42")           // Num(42)                : Expr
val y: Expr = Expr("-1 + 7")       // Add(Num(-1),Num(7))      : Expr
val z: Expr = Expr("1 + 2 * 3")    // Add(Num(1),Mul(Num(2),Num(3))) : Expr
```

If you want to get the **string representation** of the expression, please use `getString` method as follows: method.

```
x.getString // "42"                : String
y.getString // "(-1 + 7)"          : String
z.getString // "(1 + (2 * 3))"     : String
```

## 1. Parsers

ADTs for Abstract Syntax  
Parsers for Concrete Syntax

## 2. Interpreters

We will implement the **interpreter** for AE according to the following **big-step operational (natural) semantics**:

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

$$\text{NUM} \frac{}{\vdash n \Rightarrow n}$$

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

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

We will implement the **interpreter** for AE according to the following **big-step operational (natural) semantics**:

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

$$\text{NUM} \frac{}{\vdash n \Rightarrow n}$$

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

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

```
type Value = BigInt
```

```
def interp(expr: Expr): Value = expr match
  case Num(n)      => n
  case Add(l, r)   => interp(l) + interp(r)
  case Mul(l, r)   => interp(l) * interp(r)
```

```
interp(Expr("42"))           // 42 : Value (= BigInt)
interp(Expr("-1 + 7"))       // 6  : Value (= BigInt)
interp(Expr("1 + 2 * 3"))    // 7  : Value (= BigInt)
```

- Please see this document<sup>3</sup> on GitHub.
- It is just an exercise, and you **don't need to submit** anything.
- However, some exam questions might be related to this exercise.

---

<sup>3</sup><https://github.com/ku-plrg-classroom/docs/tree/main/cose212/ae>.

## 1. Parsers

ADTs for Abstract Syntax  
Parsers for Concrete Syntax

## 2. Interpreters

- Identifiers (1)

Jihyeok Park

[jihyeok\\_park@korea.ac.kr](mailto:jihyeok_park@korea.ac.kr)

<https://plrg.korea.ac.kr>