Lecture 3 – Syntax and Semantics (2)

COSE212: Programming Languages

Jihyeok Park

PLRG

2024 Fall

Recall

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

- **Syntax**
  - Concrete Syntax
  - Abstract Syntax
  - Concrete vs. Abstract Syntax
- **Semantics**
  - Inference Rules
  - Big-Step Operational (Natural) Semantics
  - Small-Step Operational (Reduction) Semantics

Recall

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

- **Syntax**
  - 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 will learn how to implement the **interpreter** for AE.

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

Contents

1. Parsers
   ADTs for Abstract Syntax
   Parsers for Concrete Syntax

2. Interpreters

Contents

1. Parsers
    - ADTs for Abstract Syntax
    - Parsers for Concrete Syntax
2. Interpreters

ADTs for Abstract Syntax

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

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

ADTs for Abstract Syntax

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

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

```scala
// 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)
```

ADTs for Abstract Syntax

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

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

```scala
// 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, an AE expression $1 \texttt{ + } (2 \texttt{ * } 3)$ is represented as follows:

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

Parsers for Concrete Syntax

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?

[1] https://github.com/scala/scala-parser-combinators

[2] https://en.wikipedia.org/wiki/Parsing_expression_grammar

Parsers for Concrete Syntax

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[1], and **parsing expression grammars (PEGs)**.[2]

[1] https://github.com/scala/scala-parser-combinators

[2] https://en.wikipedia.org/wiki/Parsing_expression_grammar

Parser Combinators

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

Parser Combinators

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

- **regular expressions** (`"...".r`) as parsers.

```
lazy val parser: Parser[String] = "-?[0-9]+".r // parsing integers
```

Parser Combinators

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

- **regular expressions** (`"...".r`) as parsers.

```
lazy val parser: Parser[String] = "-?[0-9]+".r // parsing integers
```

- **combine** them using sequence (~, <~, ~>) and alternative (|).

```
lazy val parser1: Parser[X] = ...
lazy val parser2: Parser[Y] = ...
lazy val parser3: Parser[X] = ...
parser1 ~ parser2   // Parser[X ~ Y]
parser1 <~ parser2  // Parser[X]  (discard the result of `parser2`)
parser1 ~> parser2  // Parser[Y]  (discard the result of `parser1`)
parser1 | parser3   // Parser[X]
```

Parser Combinators

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

- **regular expressions** ("...".r) as parsers.

```
lazy val parser: Parser[String] = "-?[0-9]+".r // parsing integers
```

- **combine** them using sequence (~, <~, ~>) and alternative (|).

```
lazy val parser1: Parser[X] = ...
lazy val parser2: Parser[Y] = ...
lazy val parser3: Parser[X] = ...
parser1 ~ parser2   // Parser[X ~ Y]
parser1 <~ parser2  // Parser[X]  (discard the result of `parser2`)
parser1 ~> parser2  // Parser[Y]  (discard the result of `parser1`)
parser1 | parser3   // Parser[X]
```

- **transform** the result of a parser using the operator (^^).

```
lazy val parser1: Parser[X] = ...
val f: X => Y = ...
parser1 ^^ f  // Parser[Y]  (apply `f` to the result of `parser1`)
```

Parser Combinators

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

"[]"   "[7]"   "[-042, 4, 20]"

```
type P[+T] = PackratParser[T]
lazy val num    : P[BigInt]       = "-?[0-9]+".r ^^ { BigInt(_) }
```

Parser Combinators

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

"[]"   "[7]"   "[-042, 4, 20]"

```
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     }
```

Parser Combinators

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

"[]" "[7]" "[-042, 4, 20]"

```
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 <~ "]"
```

Parser Combinators

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

"[]"   "[7]"   "[-042, 4, 20]"

```
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(list, "[]").get            // Nil                : List[BigInt]
parseAll(list, "[7]").get           // List(7)            : List[BigInt]
parseAll(list, "[-042, 4, 20]").get // List(-42, 4, 20)  : List[BigInt]
```

Parser Combinators

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

`"[]"` `"[7]"` `"[-042, 4, 20]"`

```
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(list, "[]").get            // Nil               : List[BigInt]
parseAll(list, "[7]").get           // List(7)           : 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.

Parsers using Parser Combinators

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

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

Parsers using Parser Combinators

PLRG

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
```

Parsers using Parser Combinators

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 form** of the expression, please use `str` method as follows:

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

Contents

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**:

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

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

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

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

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

```
type Value = BigInt
def interp(expr: Expr): Value = ???
```

Interpreters

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

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

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

```scala
type Value = BigInt
def interp(expr: Expr): Value = expr match
  case Num(n)    => ???
  case Add(l, r) => ???
  case Mul(l, r) => ???
```

Interpreters

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

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

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

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

Interpreters

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

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

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

```scala
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) => ???
```

Interpreters

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

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

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

```scala
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)
```

Interpreters

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

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

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

```
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"))     // interp(Num(42))                          = 42
interp(Expr("-1+7"))   // interp(Add(Num(-1), Num(7)))             = 6
interp(Expr("1+2*3")) // interp(Add(Num(1), Mul(Num(2), Num(3)))) = 7
```

Exercise #1

https://github.com/ku-plrg-classroom/docs/tree/main/cose212/ae

- Please see above document on GitHub:
  - Implement `interp` function.
  - Implement `countNums` function.

- It is just an exercise, and you **don't need to submit** anything.
- However, some exam questions might be related to this exercise.

Summary

1. Parsers
   - ADTs for Abstract Syntax
   - Parsers for Concrete Syntax

2. Interpreters

Next Lecture

- Identifiers (1)

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