CS320

How to Design Programs

Sukyoung Ryu

February 27, 2019

Homework #1

- Available from the course webpage
- Due Wednesday, March 6 (before midnight)
- No late submission!
- Cheating is strongly forbidden.
  Cheating will get you an F.

Programming Languages

"Virtually every language consists of

- a peculiar syntax,
- some behavior associated with each syntax,
- numerous useful libraries, and
- a collection of idioms that programmers of that language use." [1]

[1] Programming Languages: Application and Interpretation (PLAI), Shriram Krishnamurthi

Syntax

- Concrete syntax
  - `3 + 4` (infix)
  - `3 4 +` (postfix)
  - `+ 3 4` (prefix)
- Abstract syntax
  - `(add (num 3) (num 4))`

Semantics

- Mathematical techniques
  - Denotational semantics
  - Operational semantics
  - Axiomatic semantics
- Interpreter semantics
  to explain a language, write an interpreter for it

Programming Language

A *programming language* is defined by

- a grammar for programs
- rules for evaluating any program to produce a result

A Grammar for Algebra Programs

A grammar of Algebra in *BNF* (Backus-Naur Form):

⟨Algebra⟩	::=	(⟨Algebra⟩ + ⟨Algebra⟩)	addition
	|	(⟨Algebra⟩ - ⟨Algebra⟩)	subtraction
	|	⟨num⟩	number
⟨num⟩	::=	1, 42, 17, ...	number

Each *meta variable*, such as ⟨Algebra⟩, defines a set.

Using a BNF Grammar: $\langle \text{num} \rangle$

$$\langle \text{num} \rangle \quad ::= \quad 1,\ 42,\ 17,\ \ldots \quad \text{number}$$

The set $\langle \text{num} \rangle$ is the set of all numbers.

To make an example $\langle \text{num} \rangle$, pick an element from it:

$$2 \quad \in \quad \langle \text{num} \rangle$$
$$298 \quad \in \quad \langle \text{num} \rangle$$

Using a BNF Grammar: $\langle \text{Algebra} \rangle$

$$
\begin{array}{lcll}
\langle \text{Algebra} \rangle & ::= & (\ \langle \text{Algebra} \rangle + \langle \text{Algebra} \rangle\ ) & \text{addition} \\
 & | & (\ \langle \text{Algebra} \rangle - \langle \text{Algebra} \rangle\ ) & \text{subtraction} \\
 & | & \langle \text{num} \rangle & \text{number}
\end{array}
$$

To make an example $\langle \text{Algebra} \rangle$:

- choose one case in the grammar
- pick an example for each meta variable
- combine the examples with literal text

Using a BNF Grammar: $\langle$Algebra$\rangle$

$$
\begin{array}{lcll}
\langle \text{Algebra} \rangle & ::= & (\ \langle \text{Algebra} \rangle + \langle \text{Algebra} \rangle\ ) & \text{addition} \\
& \mid & (\ \langle \text{Algebra} \rangle - \langle \text{Algebra} \rangle\ ) & \text{subtraction} \\
& \mid & \langle \text{num} \rangle & \text{number}
\end{array}
$$

To make an example $\langle$Algebra$\rangle$:

- choose one case in the grammar — $\langle \text{num} \rangle$
- pick an example for each meta variable — $7 \in \langle \text{num} \rangle$
- combine the examples with literal text — $7 \in \langle \text{Algebra} \rangle$

Using a BNF Grammar: $\langle$Algebra$\rangle$

$$\begin{array}{lcll} \langle\text{Algebra}\rangle & ::= & (\ \langle\text{Algebra}\rangle + \langle\text{Algebra}\rangle\ ) & \text{addition} \\ & | & (\ \langle\text{Algebra}\rangle - \langle\text{Algebra}\rangle\ ) & \text{subtraction} \\ & | & \langle\text{num}\rangle & \text{number} \end{array}$$

To make an example $\langle$Algebra$\rangle$:

- choose one case in the grammar $(\ \langle\text{Algebra}\rangle + \langle\text{Algebra}\rangle\ )$
- pick an example for each meta variable

  $8 \in \langle\text{num}\rangle \subseteq \langle\text{Algebra}\rangle$
- combine the examples with literal text $(8\ +\ 8) \in \langle\text{Algebra}\rangle$

Type Definitions

$$
\begin{array}{l}
\texttt{trait}\ \mathit{type\_id} \\
\texttt{case class}\ \mathit{variant\_id}_1(\mathit{field\_id}_{11}\colon \mathit{type}_{11},\ \cdots, \\
\qquad\qquad\qquad\qquad \mathit{field\_id}_{1n}\colon \mathit{type}_{1n})\ \texttt{extends}\ \mathit{type\_id} \\
\ldots \\
\texttt{case class}\ \mathit{variant\_id}_m(\mathit{field\_id}_{m1}\colon \mathit{type}_{m1},\ \cdots, \\
\qquad\qquad\qquad\qquad \mathit{field\_id}_{ml}\colon \mathit{type}_{ml})\ \texttt{extends}\ \mathit{type\_id}
\end{array}
$$

Shapes

```
trait Shape
case class Triangle(a: Int, b: Int, c: Int) extends Shape
case class Rectangle(h: Int, w: Int) extends Shape
case class Square(side: Int) extends Shape
```

Shapes

trait $\mathit{type\_id}$
case class $\mathit{variant\_id}_1(\mathit{field\_id}_{11}\colon \mathit{type}_{11}, \cdots,$
$\qquad \mathit{field\_id}_{1n}\colon \mathit{type}_{1n})$ extends $\mathit{type\_id}$
$\ldots$
case class $\mathit{variant\_id}_m(\mathit{field\_id}_{m1}\colon \mathit{type}_{m1}, \cdots,$
$\qquad \mathit{field\_id}_{ml}\colon \mathit{type}_{ml})$ extends $\mathit{type\_id}$

```
trait Shape
case class Triangle(a: Int, b: Int, c: Int) extends Shape
case class Rectangle(h: Int, w: Int) extends Shape
case class Square(side: Int) extends Shape
```

Type Definitions

```
trait type_id
case class variant_id_1(field_id_11: type_11, ···,
                        field_id_1n: type_1n) extends type_id
···
case class variant_id_m(field_id_m1: type_m1, ···,
                        field_id_ml: type_ml) extends type_id
```

- A constructor $variant\_id_i$ is defined for each variant.
- A type $variant\_id_i$ is defined for each variant as well.
- Each constructor takes an argument for each field of its variant.
- Each field has an annotated type $type_{ij}$.
- Each field is accessed by its name $field\_id_{ij}$.

Shapes

- A constructor *variant_id$_i$* is defined for each variant.
- Each constructor takes an argument for each field of its variant.
- Each field has an annotated type *type$_{ij}$*.
- Each field is accessed by its name *field_id$_{ij}$*.

```
trait Shape
case class Triangle(a: Int, b: Int, c: Int) extends Shape
case class Rectangle(h: Int, w: Int) extends Shape
case class Square(side: Int) extends Shape
```

```
val t = Triangle(3, 4, 5)
val r = Rectangle(5, 3)
t.a == r.w
```

Shapes

- A constructor *variant_id$_i$* is defined for each variant.
- Each constructor takes an argument for each field of its variant.
- Each field has an annotated type *type$_{ij}$*.
- Each field is accessed by its name *field_id$_{ij}$*.

```
trait Shape
case class Triangle(a: Int, b: Int, c: Int) extends Shape
case class Rectangle(h: Int, w: Int) extends Shape
case class Square(side: Int) extends Shape
```

```
val t = Triangle(3, 4, 5)
val r = Rectangle(5, 3)
t.a == r.w // res1: Boolean = true
```

A Grammar for Arithmetic Expressions

```
<AE> ::= <num>
       | {+ <AE> <AE>}
       | {- <AE> <AE>}
```

```
trait AE
case class Num(num: Int) extends AE
case class Add(left: AE, right: AE) extends AE
case class Sub(left: AE, right: AE) extends AE
```

```
val ae = Add(Num(3), Sub(Num(8), Num(2)))
ae.left
```

A Grammar for Arithmetic Expressions

```
<AE> ::= <num>
       | {+ <AE> <AE>}
       | {- <AE> <AE>}
```

```
trait AE
case class Num(num: Int) extends AE
case class Add(left: AE, right: AE) extends AE
case class Sub(left: AE, right: AE) extends AE
```

```
val ae = Add(Num(3), Sub(Num(8), Num(2)))
ae.left
```

A Grammar for Arithmetic Expressions

```
<AE> ::= <num>
       | {+ <AE> <AE>}
       | {- <AE> <AE>}
```

```
trait AE
case class Num(num: Int) extends AE
case class Add(left: AE, right: AE) extends AE
case class Sub(left: AE, right: AE) extends AE

val ae = Add(Num(3), Sub(Num(8), Num(2)))
ae.left
```

A Grammar for Arithmetic Expressions

```
<AE> ::= <num>
       | {+ <AE> <AE>}
       | {- <AE> <AE>}
```

```
trait AE
case class Num(num: Int) extends AE
case class Add(left: AE, right: AE) extends AE
case class Sub(left: AE, right: AE) extends AE
```

```
val ae = Add(Num(3), Sub(Num(8), Num(2)))
ae.left // type: AE, value: Num(3)
```

Pattern Matching

$$\textit{expr}\ \texttt{match}\ \{$$
$$\quad \texttt{case}\ \textit{variant\_id}_1(\textit{field\_id}_{11}, \ldots)\ \texttt{=>}\ \textit{expr}_1$$
$$\quad \ldots$$
$$\quad \texttt{case}\ \textit{variant\_id}_m(\textit{field\_id}_{m1}, \ldots)\ \texttt{=>}\ \textit{expr}_m$$
$$\}$$

```
// perimeter : Shape => Int
def perimeter(sh: Shape): Int = sh match {
  case Triangle(a, b, c) => a + b + c
  case Rectangle(h, w) => 2 * (h + w)
  case Square(s) => 4 * s
}
perimeter(Triangle(3, 4, 5))
```

Pattern Matching

expr `match` {
  `case` *variant_id*$_1$(*field_id*$_{11}$, …) => *expr*$_1$
  …
  `case` *variant_id*$_m$(*field_id*$_{m1}$, …) => *expr*$_m$
}

```
// perimeter : Shape => Int
def perimeter(sh: Shape): Int = sh match {
  case Triangle(a, b, c) => a + b + c
  case Rectangle(h, w) => 2 * (h + w)
  case Square(s) => 4 * s
}
perimeter(Triangle(3, 4, 5))
```

Pattern Matching

$$\begin{array}{l} \mathit{expr}\ \texttt{match}\ \{ \\ \quad \texttt{case}\ \mathit{variant\_id}_1(\mathit{field\_id}_{11}, \ldots)\ \texttt{=>}\ \mathit{expr}_1 \\ \quad \ldots \\ \quad \texttt{case}\ \mathit{variant\_id}_m(\mathit{field\_id}_{m1}, \ldots)\ \texttt{=>}\ \mathit{expr}_m \\ \} \end{array}$$

```
// interp : AE => Int
def interp(ae: AE): Int = ae match {
  case Num(n) => n
  case Add(l, r) => interp(l) + interp(r)
  case Sub(l, r) => interp(l) - interp(r)
}
interp(ae)
```

Pattern Matching

$$\begin{array}{l} \textit{expr}\ \texttt{match}\ \{ \\ \quad \texttt{case}\ \textit{variant\_id}_1(\textit{field\_id}_{11}, \ldots)\ \texttt{=>}\ \textit{expr}_1 \\ \quad \ldots \\ \quad \texttt{case}\ \textit{variant\_id}_m(\textit{field\_id}_{m1}, \ldots)\ \texttt{=>}\ \textit{expr}_m \\ \} \end{array}$$

```
// interp : AE => Int
def interp(ae: AE): Int = ae match {
  case Num(n) => n
  case Add(l, r) => interp(l) + interp(r)
  case Sub(l, r) => interp(l) - interp(r)
}
interp(ae)
```

Pattern Matching

$$\mathit{expr}\ \texttt{match}\ \{$$
$$\quad \texttt{case}\ \mathit{variant\_id}_1(\mathit{field\_id}_{11}, \ldots)\ \texttt{=>}\ \mathit{expr}_1$$
$$\quad \ldots$$
$$\quad \texttt{case}\ \mathit{variant\_id}_m(\mathit{field\_id}_{m1}, \ldots)\ \texttt{=>}\ \mathit{expr}_m$$
$$\}$$

```
// ... : AE => ...
def ...(ae: AE): ... = ae match {
  case Num(n) => ...
  case Add(l, r) => ...
  case Sub(l, r) => ...
}
```

How to Design Programs

- Determine the data representation
  - `trait` and `case class`
- Write tests
  - `test`
- Create a template for the implementation
  - `match`
- Finish implementation case-by-case
- Run tests

Tests

```
println("Hello world!")

error("message") // throws an error with "[ERROR] message"

test(1, 1)       // prints nothing
test(1, 0)       // prints "FAIL: 1 is not equal to 0"
```

Tests

```
println("Hello world!")

error("message") // throws an error with "[ERROR] message"

test(1, 1)         // prints nothing
test(1, 0)         // prints "FAIL: 1 is not equal to 0"

// prints nothing
testExc(error("it is a message"), "message")
// prints "FAIL[file:14]: it should throw an error but result is 1"
textExc(1, "message")
// prints "FAIL[file:8]: "[ERROR] other" does not contain "message""
textExc(error("other"), "message")
```

Lists

- A *list* is either the constant `Nil`, or it is a pair whose second value is a list.

```
val x: List[Int] = Nil
val y: List[Int] = List(1, 2, 3, 4)
y.length
42 :: y
y.reverse
y.contains(1)
y.map(_ * 2)
y.foldLeft(0)(_ + _)
```

https://www.tutorialspoint.com/scala/scala_lists.htm

https://www.scala-lang.org/api/2.12.3/scala/collection/immutable/List.html

Lists

- A *list* is either the constant `Nil`, or it is a pair whose second value is a list.

```
val x: List[Int] = Nil             // List()
val y: List[Int] = List(1, 2, 3, 4) // List(1, 2, 3, 4)
y.length                            // 4
42 :: y                             // List(42, 1, 2, 3, 4)
y.reverse                           // List(4, 3, 2, 1)
y.contains(1)                       // true
y.map(_ * 2)                        // List(2, 4, 6, 8)
y.foldLeft(0)(_ + _)                // 10
```

https://www.tutorialspoint.com/scala/scala_lists.htm

https://www.scala-lang.org/api/2.12.3/scala/collection/immutable/List.html

Homework #1

- Available from the course webpage
- Due Wednesday, March 6 (before midnight)
- No late submission!
- Cheating is strongly forbidden.

  Cheating will get you an F.

Sukyoung Ryu
sryu.cs@kaist.ac.kr
http://plrg.kaist.ac.kr