CSE M226: Programming Languages

Lecture 11: Lambda Calculus and The Basics of Scala

Rokan Uddin Faruqui

Dept. of Computer Science and Engineering
University of Chittagong, Bangladesh
Email: rufaruqui@cu.ac.bd

- Very general mathematical language.

- Very general mathematical language.
- Simple syntax, powerful semantics.

- Very general mathematical language.
- Simple syntax, powerful semantics.
- Encapsulates function abstraction (definition) and application.
- Functions easily used as values.

- Very general mathematical language.
- Simple syntax, powerful semantics.
- Encapsulates function abstraction (definition) and application.
- Functions easily used as values.
- Functional programming languages can be viewed as syntactic variants.

Concrete Syntax for λ Calculus

```
1  <expression> ::=
2      <variable>
3      | <constant>
4      | ( <expression1><expression2> )
5      | ( λ<variable>.<expression> )
6
```

Concrete Syntax for λ Calculus

- Variables - lower case identifiers
- Constants - any predefined object - impure/applied lambda calculus
- Rule 3 - application of *expression*1 to *expression*1
  - operator
  - operand
- Rule 4 - lambda abstraction
  - bound variable
  - body

Application (Combinations)

- $(E_1 \quad E_2)$ means: apply the result of evaluating E_1 to E_2

Application (Combinations)

- $(E_1 \quad E_2)$ means: apply the result of evaluating E_1 to E_2
- E_1 should be a function, either predefined (a constant) or an abstraction.

Application (Combinations)

- $(E_1 \quad E_2)$ means: apply the result of evaluating E_1 to E_2
- E_1 should be a function, either predefined (a constant) or an abstraction.
- In the pure lambda calculus, every (closed) expression is a function.

Application (Combinations)

- $(E_1 \;\; E_2)$ means: apply the result of evaluating E_1 to E_2
- E_1 should be a function, either predefined (a constant) or an abstraction.
- In the pure lambda calculus, every (closed) expression is a function.

Example

(succ 1) where *succ* is the successor

Application (Combinations)

- $(E_1 \;\; E_2)$ means: apply the result of evaluating E_1 to E_2
- E_1 should be a function, either predefined (a constant) or an abstraction.
- In the pure lambda calculus, every (closed) expression is a function.

Example

(succ 1) where *succ* is the successor

(zerop 0) where *zerop* is the zero predicate (test if zero)

Application (Combinations)

- $(E_1 \;\; E_2)$ means: apply the result of evaluating E_1 to E_2
- E_1 should be a function, either predefined (a constant) or an abstraction.
- In the pure lambda calculus, every (closed) expression is a function.

Example

(succ 1) where *succ* is the successor

(zerop 0) where *zerop* is the zero predicate (test if zero)

(+ 3 5) where + is the addition operation function

Application (Combinations)

- $(E_1 \;\; E_2)$ means: apply the result of evaluating E_1 to E_2
- E_1 should be a function, either predefined (a constant) or an abstraction.
- In the pure lambda calculus, every (closed) expression is a function.

Example

(succ 1) where *succ* is the successor

(zerop 0) where *zerop* is the zero predicate (test if zero)

(+ 3 5) where + is the addition operation function

Abstractions

anonymous functions

$$(\lambda x.x)$$

is similar to

$$f(x) = x$$

but has no name

Example

- $(\lambda m.(+\ m\ 1))) \equiv f(m) = m + 1$

Abstractions

anonymous functions

$$(\lambda x.x)$$

is similar to

$$f(x) = x$$

but has no name

Example

- $(\lambda m.(+\ m\ 1))) \equiv f(m) = m + 1$
- $(\lambda m.\ (\lambda n.(-\ m\ n)))) \equiv g(m,n) = m - n$

Scala: A scalable Language

- It was designed to grow with the demands of its users

Scala: A scalable Language

- It was designed to grow with the demands of its users
- It runs on the standard Java platform and interoperates seamlessly with all Java libraries.

Scala: A scalable Language

- It was designed to grow with the demands of its users
- It runs on the standard Java platform and interoperates seamlessly with all Java libraries.
- Scala is a blend of object-oriented and functional programming concepts in a statically typed language

Scala: A scalable Language

- It was designed to grow with the demands of its users
- It runs on the standard Java platform and interoperates seamlessly with all Java libraries.
- Scala is a blend of object-oriented and functional programming concepts in a statically typed language
- The combination of both styles in Scala makes it possible to express new kinds of programming patterns and component abstractions

Scala: A scalable Language

- It was designed to grow with the demands of its users
- It runs on the standard Java platform and interoperates seamlessly with all Java libraries.
- Scala is a blend of object-oriented and functional programming concepts in a statically typed language
- The combination of both styles in Scala makes it possible to express new kinds of programming patterns and component abstractions

Scala: A scalable Language

- A language that grows on.
  - Growing new types
    - Eric Raymond introduced the cathedral and bazaar as two metaphors of software

Scala: A scalable Language

- A language that grows on.
  - Growing new types
    - Eric Raymond introduced the cathedral and bazaar as two metaphors of software development
    - The cathedral is a near-perfect building that takes a long time to build. Once built, it stays unchanged for a long time.

Scala: A scalable Language

- A language that grows on.
  - Growing new types
    - Eric Raymond introduced the cathedral and bazaar as two metaphors of software development
    - The cathedral is a near-perfect building that takes a long time to build. Once built, it stays unchanged for a long time.
    - The bazaar, by contrast, is adapted and extended each day by the people working in it.

Scala: A scalable Language

- A language that grows on.
  - Growing new types
    - Eric Raymond introduced the cathedral and bazaar as two metaphors of software development
    - The cathedral is a near-perfect building that takes a long time to build. Once built, it stays unchanged for a long time.
    - The bazaar, by contrast, is adapted and extended each day by the people working in it.
    - In Raymond's work the bazaar is a metaphor for open-source software development

Scala: A scalable Language

- A language that grows on.
  - Growing new types
    - Eric Raymond introduced the cathedral and bazaar as two metaphors of software development
    - The cathedral is a near-perfect building that takes a long time to build. Once built, it stays unchanged for a long time.
    - The bazaar, by contrast, is adapted and extended each day by the people working in it.
    - In Raymond's work the bazaar is a metaphor for open-source software development
    - Scala lets you add new types that can be used as conveniently as built-in types.

Scala: A scalable Language

- A language that grows on.
  - Growing new control constructs
    - The same extension principle also applies to control structures.

What makes Scala scalable?

- Scala is object-oriented
  - pure object oriented (primitives value in Java *integer* is not object)

What makes Scala scalable?

- Scala is object-oriented
  - pure object oriented (primitives value in Java *integer* is not object)
  - mixin composition

What makes Scala scalable?

- Scala is object-oriented
  - pure object oriented (primitives value in Java *integer* is not object)
  - mixin composition
- Scala is functional

What makes Scala scalable?

- Scala is object-oriented
  - pure object oriented (primitives value in Java *integer* is not object)
  - mixin composition
- Scala is functional
  - functions are first-class

What makes Scala scalable?

- Scala is object-oriented
  - pure object oriented (primitives value in Java *integer* is not object)
  - mixin composition
- Scala is functional
  - functions are first-class
  - map input with output instead of changing value in memory (no side effects)

What makes Scala scalable?

- Scala is object-oriented
  - pure object oriented (primitives value in Java *integer* is not object)
  - mixin composition
- Scala is functional
  - functions are first-class
  - map input with output instead of changing value in memory (no side effects)

Scala's roots

- Scala adopts a large part of the syntax of *Java* and *C#*

Scala's roots

- Scala adopts a large part of the syntax of *Java* and *C#*
- Expressions, statements, and blocks are mostly as in Java, as is the syntax of classes, packages and imports

Scala's roots

- Scala adopts a large part of the syntax of *Java* and *C#*
- Expressions, statements, and blocks are mostly as in Java, as is the syntax of classes, packages and imports
- Object model was pioneered by *Smalltalk* and taken up subsequently by *Ruby*
- Its approach to functional programming is quite similar

Scala's roots

- Scala adopts a large part of the syntax of *Java* and *C#*
- Expressions, statements, and blocks are mostly as in Java, as is the syntax of classes, packages and imports
- Object model was pioneered by *Smalltalk* and taken up subsequently by *Ruby*
- Its approach to functional programming is quite similar in spirit to the *ML* family of languages, which has *SML*, *OCaml*, and *F#* as prominent members

Scala's roots

- Scala adopts a large part of the syntax of *Java* and *C#*
- Expressions, statements, and blocks are mostly as in Java, as is the syntax of classes, packages and imports
- Object model was pioneered by *Smalltalk* and taken up subsequently by *Ruby*
- Its approach to functional programming is quite similar in spirit to the *ML* family of languages, which has *SML*, *OCaml*, and *F#* as prominent members
- *C++* is another scalable language that can be adapted and extended through operator overloading and its template system; compared to Scala it is built on a lower-level, more systems-oriented core

Scala's roots

- OOP + FP $\longrightarrow$ Ruby, Smalltalk, and Python

Basics of the Language

- Data Types

Basics of the Language

- Data Types
- Statements and Expressions

Basics of the Language

- Data Types
- Statements and Expressions
- Statement Separator and Blocks

Basics of the Language

- Data Types
- Statements and Expressions
- Statement Separator and Blocks
- Comments

Basics of the Language

- Data Types
- Statements and Expressions
- Statement Separator and Blocks
- Comments
- Declarations

Basics of the Language

- Data Types
- Statements and Expressions
- Statement Separator and Blocks
- Comments
- Declarations
- Composite Types: Cartesian Products

Basics of the Language

- Data Types
- Statements and Expressions
- Statement Separator and Blocks
- Comments
- Declarations
- Composite Types: Cartesian Products
- Nested Declarations

Functions

```
1 (a:Int) => 2*a
  (a:Int, b:Int) => a+b
3 (a:Int, f:Int=>Int) => f(a)

5 In general
  Type1, Type2, ..., TypeN => OutputType
7
```

Functions

```
((a:Int)=>2*a)(3)

((a:Int, f:Int=>Int) => f(a))(3,((a:Int)=>2*a))
```

Functions

```
1
  val f1 = (a:Int) => 2*a
3
  val f2 = (a:Int, b:Int) => a+b
5
  val f3 = (a:Int, f:Int=>Int) => f(a)
7
In general
9 val name = <anonymous-function-definition>
```

Functions with Type

```
val f1 = (a:Int) => 2*a
val f1:(Int => Int) = a => 2*a

val f2:((Int,Int)=>Int) = (a:Int, b:Int) => a+b
val f2:((Int,Int)=>Int) = (a, b) => a+b

val f3:((Int,Int=>Int)=>Int) = (a:Int, f:Int=>Int)
    => f(a)
val f3:((Int,Int=>Int)=>Int) = (a, f) => f(a)
```