Introduction to Clean

Mónika Mészáros
E-mail: bonnie@inf.elte.hu

Department of Programming Languages and Compilers
ELTE

May 21, 2009.

Outline of the presentation

Outline of the presentation

Outline of the presentation

Outline of the presentation

Outline of the presentation

1. Functional programming languages
2. Evaluation
3. Characteristics of Clean
4. Clean basics
5. Lists, functions on lists
6. Polymorphic functions
7. Exercises

Outline of the presentation

1. Functional programming languages
2. Evaluation
3. Characteristics of Clean
4. Clean basics
5. Lists, functions on lists
6. Polymorphic functions
7. Exercises

Outline of the presentation

1. Functional programming languages
2. Evaluation
3. Characteristics of Clean
4. Clean basics
5. Lists, functions on lists
6. Polymorphic functions
7. Exercises

Functional programming languages

- Subset of declarative programming languages: computation is defined by set of declarations
- Specification of problem, refinement of problem are the main concerns
- Type, class, function definitions, initial expression
- Computation means evaluation of the initial expression (rewriting rules)
- Program components solving subproblems do not cause side-effects
- Mathematical model of computation: λ-calculus (Church, 1932-33, computationally equivalent to Turing machine)

Functional programming languages

- Subset of declarative programming languages: computation is defined by set of declarations
- Specification of problem, refinement of problem are the main concerns
- Type, class, function definitions, initial expression
- Computation means evaluation of the initial expression (rewriting rules)
- Program components solving subproblems do not cause side-effects
- Mathematical model of computation: λ-calculus (Church, 1932-33, computationally equivalent to Turing machine)

Functional programming languages

- Subset of declarative programming languages: computation is defined by set of declarations
- Specification of problem, refinement of problem are the main concerns
- Type, class, function definitions, initial expression
- Computation means evaluation of the initial expression (rewriting rules)
- Program components solving subproblems do not cause side-effects
- Mathematical model of computation: λ-calculus (Church, 1932-33, computationally equivalent to Turing machine)

Functional programming languages

- Subset of declarative programming languages: computation is defined by set of declarations
- Specification of problem, refinement of problem are the main concerns
- Type, class, function definitions, initial expression
- Computation means evaluation of the initial expression (rewriting rules)
- Program components solving subproblems do not cause side-effects
- Mathematical model of computation: λ-calculus (Church, 1932-33, computationally equivalent to Turing machine)

Functional programming languages

- Subset of declarative programming languages: computation is defined by set of declarations
- Specification of problem, refinement of problem are the main concerns
- Type, class, function definitions, initial expression
- Computation means evaluation of the initial expression (rewriting rules)
- Program components solving subproblems do not cause side-effects
- Mathematical model of computation: λ-calculus (Church, 1932-33, computationally equivalent to Turing machine)

Functional programming languages

- Subset of declarative programming languages: computation is defined by set of declarations
- Specification of problem, refinement of problem are the main concerns
- Type, class, function definitions, initial expression
- Computation means evaluation of the initial expression (rewriting rules)
- Program components solving subproblems do not cause side-effects
- Mathematical model of computation: λ-calculus (Church, 1932-33, computationally equivalent to Turing machine)

Evaluation

- **Evaluation** = sequence of rewriting (reduction) steps
- **A reduction step**: substitution (rewriting) of a function application by its definition in the body, until we reach normal form
- **Evaluation strategy**: selection order of redexes (reducible expressions), well-known strategies: **lazy** (function application first), **strict** (arguments first), **paralell**
- **Normal form** is unique (in confluent rewriting systems), lazy evaluation order always finds the normal form, if it exists

Evaluation

- **Evaluation** = sequence of rewriting (reduction) steps
- **A reduction step**: substitution (rewriting) of a function application by its definition in the body, until we reach normal form
- **Evaluation strategy**: selection order of redexes (reducible expressions), well-known strategies: **lazy** (function application first), **strict** (arguments first), **paralell**
- **Normal form** is unique (in confluent rewriting systems), lazy evaluation order always finds the normal form, if it exists

Evaluation

- **Evaluation** = sequence of rewriting (reduction) steps
- **A reduction step**: substitution (rewriting) of a function application by its definition in the body, until we reach normal form
- **Evaluation strategy**: selection order of redexes (reducible expressions), well-known strategies: **lazy** (function application first), **strict** (arguments first), **paralell**
- **Normal form** is unique (in confluent rewriting systems), lazy evaluation order always finds the normal form, if it exists

Evaluation

- **Evaluation** = sequence of rewriting (reduction) steps
- **A reduction step**: substitution (rewriting) of a function application by its definition in the body, until we reach normal form
- **Evaluation strategy**: selection order of redexes (reducible expressions), well-known strategies: **lazy** (function application first), **strict** (arguments first), **paralell**
- **Normal form** is unique (in confluent rewriting systems), lazy evaluation order always finds the normal form, if it exists

Examples of evaluation

```
inc       x = x + 1
square    x = x * x
squareinc x = square (inc x)
```

Evaluation of **squareinc 7**:

- **strict**:

```
squareinc 7 -> square (inc 7) -> square (7+1)
    -> square 8 -> 8*8 -> 64
```

- **lazy**:

```
squareinc 7 -> square (inc 7)
    -> (inc 7) * (inc 7) -> (7+1) * (inc 7)
    -> 8 * (inc 7) -> 8 * (7+1) -> 8*8 -> 64
```

Clean uses lazy evaluation.

Examples of evaluation

```
inc       x = x + 1
square    x = x * x
squareinc x = square (inc x)
```

Evaluation of **squareinc 7**:

- **strict**:

  ```
  squareinc 7 -> square (inc 7) -> square (7+1)
      -> square 8 -> 8*8 -> 64
  ```

- **lazy**:

  ```
  squareinc 7 -> square (inc 7)
      -> (inc 7) * (inc 7) -> (7+1) * (inc 7)
      -> 8 * (inc 7) -> 8 * (7+1) -> 8*8 -> 64
  ```

Clean uses lazy evaluation.

Examples of evaluation

```
inc       x = x + 1
square    x = x * x
squareinc x = square (inc x)
```

Evaluation of **squareinc 7**:

- **strict**:

```
squareinc 7 -> square (inc 7) -> square (7+1)
    -> square 8 -> 8*8 -> 64
```

- **lazy**:

```
squareinc 7 -> square (inc 7)
    -> (inc 7) * (inc 7) -> (7+1) * (inc 7)
    -> 8 * (inc 7) -> 8 * (7+1) -> 8*8 -> 64
```

Clean uses lazy evaluation.

Examples of evaluation

```
inc       x = x + 1
square    x = x * x
squareinc x = square (inc x)
```

Evaluation of **squareinc 7**:

- **strict**:

```
squareinc 7 -> square (inc 7) -> square (7+1)
    -> square 8 -> 8*8 -> 64
```

- **lazy**:

```
squareinc 7 -> square (inc 7)
    -> (inc 7) * (inc 7) -> (7+1) * (inc 7)
    -> 8 * (inc 7) -> 8 * (7+1) -> 8*8 -> 64
```

Clean uses lazy evaluation.

Examples of evaluation

```
inc       x = x + 1
square    x = x * x
squareinc x = square (inc x)
```

Evaluation of **squareinc 7**:

- **strict**:

```
squareinc 7 -> square (inc 7) -> square (7+1)
    -> square 8 -> 8*8 -> 64
```

- **lazy**:

```
squareinc 7 -> square (inc 7)
    -> (inc 7) * (inc 7) -> (7+1) * (inc 7)
    -> 8 * (inc 7) -> 8 * (7+1) -> 8*8 -> 64
```

Clean uses lazy evaluation.

Characteristics of Clean

- No destructive assignments
- Referential transparency - equational reasoning (same expression means always the same value)
- Strongly typed (every subexpression has a static type), type deduction, polymorphism, abstract algebraic data types
- Higher order functions (argument or value is a function) example:
  `twice f x = f (f x) //f is a function`
- Currying - functions with 1 argument
  `(+) x y vs. ((+) x) y)`

```
inc = (+) 1
```

Characteristics of Clean

- No destructive assignments
- Referential transparency - equational reasoning (same expression means always the same value)
- Strongly typed (every subexpression has a static type), type deduction, polymorphism, abstract algebraic data types
- Higher order functions (argument or value is a function) example:
  `twice f x = f (f x) //f is a function`
- Currying - functions with 1 argument
  `(+) x y vs. ((+) x) y)`

```
inc = (+) 1
```

Characteristics of Clean

- No destructive assignments
- Referential transparency - equational reasoning (same expression means always the same value)
- Strongly typed (every subexpression has a static type), type deduction, polymorphism, abstract algebraic data types
- Higher order functions (argument or value is a function) example:
  `twice f x = f (f x) //f is a function`
- Currying - functions with 1 argument
  `(+) x y vs. ((+) x) y)`

```
inc = (+) 1
```

Characteristics of Clean

- No destructive assignments
- Referential transparency - equational reasoning (same expression means always the same value)
- Strongly typed (every subexpression has a static type), type deduction, polymorphism, abstract algebraic data types
- Higher order functions (argument or value is a function) example:
  `twice f x = f (f x) //f is a function`
- Currying - functions with 1 argument
  `(+) x y vs. ((+) x) y)`

```
inc = (+) 1
```

Characteristics of Clean

- No destructive assignments
- Referential transparency - equational reasoning (same expression means always the same value)
- Strongly typed (every subexpression has a static type), type deduction, polymorphism, abstract algebraic data types
- Higher order functions (argument or value is a function) example:
  `twice f x = f (f x) //f is a function`
- Currying - functions with 1 argument
  `(+) x y vs.  ((+) x) y)`

`inc = (+) 1`

Characteristics of Clean

- No destructive assignments
- Referential transparency - equational reasoning (same expression means always the same value)
- Strongly typed (every subexpression has a static type), type deduction, polymorphism, abstract algebraic data types
- Higher order functions (argument or value is a function) example:
  `twice f x = f (f x) //f is a function`
- Currying - functions with 1 argument
  `(+) x y vs. ((+) x) y)`

```
inc = (+) 1
```

Characteristics of Clean

- Recursion

```
fac 0             = 1
fac n | n > 0 = n * fac (n-1)
```

- Lazy evaluation and strictness analysis

```
take 5 ( map inc [1 .. ] )
```

- Zermelo-Fraenkel set-expressions

```
[ <expression> \\ <generator> | <filter> ]
<generator> : <value> <- <list>
```

```
[ x * x \\ x <- [ 1 .. ] | odd x ]
=> [1, 9, 25, ..]
```

Characteristics of Clean

- Recursion

```
fac 0           = 1
fac n | n > 0 = n * fac (n-1)
```

- Lazy evaluation and strictness analysis

```
take 5 ( map inc [1 .. ] )
```

- Zermelo-Fraenkel set-expressions

```
[ <expression> \\ <generator> | <filter> ]
<generator> : <value> <- <list>
```

```
[ x * x \\ x <- [ 1 .. ] | odd x ]
=> [1, 9, 25, ..]
```

Characteristics of Clean

- Recursion

```
fac 0           = 1
fac n | n > 0 = n * fac (n-1)
```

- Lazy evaluation and strictness analysis

```
take 5 ( map inc [1 .. ] )
```

- Zermelo-Fraenkel set-expressions

```
[ <expression> \\ <generator> | <filter> ]
<generator> : <value> <- <list>
```

```
[ x * x \\ x <- [ 1 .. ] | odd x ]
=> [1, 9, 25, ..]
```

Characteristics of Clean

- Recursion

```
fac 0           = 1
fac n | n > 0 = n * fac (n-1)
```

- Lazy evaluation and strictness analysis

```
take 5 ( map inc [1 .. ] )
```

- Zermelo-Fraenkel set-expressions

```
[ <expression> \\ <generator> | <filter> ]
<generator> : <value> <- <list>
```

```
[ x * x \\ x <- [ 1 .. ] | odd x ]
=> [1, 9, 25, ..]
```

Characteristics of Clean

- Pattern matching of arguments

```
<function name> <pattern>  or
<function name> <pattern> | <condition>
```

```
fac 0           = 1
fac n | n > 0 = n * fac (n-1)
```

- Off-side rule determining scope of identifiers

```
add4 = twice inc          //inc mean local inc
where
    inc x = x+2           //local inc declaration
add = ... inc ...         //inc means global inc
```

Characteristics of Clean

- Pattern matching of arguments

  `<function name> <pattern>  or`
  `<function name> <pattern> | <condition>`

```
fac 0           = 1
fac n | n > 0 = n * fac (n-1)
```

- Off-side rule determining scope of identifiers

```
add4 = twice inc          //inc mean local inc
where
    inc x = x+2           //local inc declaration
add = ... inc ...         //inc means global inc
```

Characteristics of Clean

- Pattern matching of arguments

  `<function name> <pattern>  or`
  `<function name> <pattern> | <condition>`

```
fac 0           = 1
fac n | n > 0 = n * fac (n-1)
```

- Off-side rule determining scope of identifiers

```
add4 = twice inc         //inc mean local inc
where
    inc x = x+2          //local inc declaration
add = ... inc ...        //inc means global inc
```

First program in Clean

```
//this is a compilation unit;
//filename: test.icl
module test

//imports modules from Standard Environment
import StdEnv

//function definitions

fac 0 = 1
fac n | n > 0 = n * fac (n-1)

//initial expression
Start = fac 5
```

Quadratic equation

```
module quadratic
import StdEnv

qeq :: Real Real Real -> (String, [Real])
qeq a b c
  | a == 0.0     = ("not quadratic", [])
  | delta < 0.0  = ("complex roots", [])
  | delta == 0.0 = ("one root", [~b/2.0*a])
  | delta > 0.0  = ("two roots",
    [(~b+radix)/(2.0*a), (~b-radix)/(2.0*a)])
      where
        delta = b*b-4.0*a*c
        radix = sqrt delta
Start = qeq 1.0 (-4.0) 1.0
```

Lists

- A **list** is a sequence of values of same type a
  The type of this list is `[a]`
- **Defining a list**:
  - `[]` - empty list
  - `[`e_1`, `e_2`, ..., `e_n`]` - enumerate the elements
  - `[e : list]` - the list's first element is e, the other elements are elements of `list`
- **Example**:

```
l = ['a', 'b', 'c']
```

```
z :: [[Int]]
z = [[1,2,3],[1,2]]
```

Lists

- A **list** is a sequence of values of same type a
  The type of this list is `[a]`
- **Defining a list**:
  - `[]` - empty list
  - `[`e_1`, `e_2`, ..., `e_n`]` - enumerate the elements
  - `[e : list]` - the list's first element is e, the other elements are elements of `list`
- **Example**:

```
l = ['a', 'b', 'c']
```

```
z :: [[Int]]
z = [[1,2,3],[1,2]]
```

Lists

- A **list** is a sequence of values of same type a
  The type of this list is `[a]`
- **Defining a list**:
  - `[]` - empty list
  - $[e_1,\ e_2,\ \ldots,\ e_n]$ - enumerate the elements
  - `[e : list]` - the list's first element is e, the other elements are elements of `list`
- **Example**:

```
l = ['a', 'b', 'c']
```

```
z :: [[Int]]
z = [[1,2,3],[1,2]]
```

Lists

- A **list** is a sequence of values of same type a
  The type of this list is `[a]`
- **Defining a list**:
  - `[]` - empty list
  - `[`e_1`, `e_2`, ..., `e_n`]` - enumerate the elements
  - `[e : list]` - the list's first element is e, the other elements are elements of `list`
- **Example**:

```
l = ['a', 'b', 'c']
```

```
z :: [[Int]]
z = [[1,2,3],[1,2]]
```

Standard functions on lists

```
hd [x : xs]    = x
hd []          = abort "hd of []"
```

```
tl [x : xs]    = xs
tl []          = abort "tl of []"
```

```
sum []         = 0
sum [x : xs]   = x + sum xs
```

```
length []      = 0
length [x:xs]  = 1 + length xs
```

Standard functions on lists

```
hd [x : xs]    = x
hd []          = abort "hd of []"
```

```
tl [x : xs]    = xs
tl []          = abort "tl of []"
```

```
sum []         = 0
sum [x : xs]   = x + sum xs
```

```
length []      = 0
length [x:xs]  = 1 + length xs
```

Standard functions on lists

```
hd [x : xs]     = x
hd []           = abort "hd of []"
```

```
tl [x : xs]     = xs
tl []           = abort "tl of []"
```

```
sum []          = 0
sum [x : xs]    = x + sum xs
```

```
length []       = 0
length [x:xs]   = 1 + length xs
```

Standard functions on lists

```
hd [x : xs]    = x
hd []          = abort "hd of []"
```

```
tl [x : xs]    = xs
tl []          = abort "tl of []"
```

```
sum []         = 0
sum [x : xs]   = x + sum xs
```

```
length []      = 0
length [x:xs]  = 1 + length xs
```

Polymorphic type

- Types can be **parametrised** - eg. [Int] - [a]
- A function that can be applied to values of different types is called as **polymorphic function**.

```
length :: [a] -> Int  // a is a type variable
hd     :: [a] -> a
```

- The functionality of the polymorphic function doesn't depend on the actual type.

Polymorphic type

- Types can be **parametrised** - eg. [Int] - [a]
- A function that can be applied to values of different types is called as **polymorphic function**.

```
length :: [a] -> Int  // a is a type variable
hd     :: [a] -> a
```

- The functionality of the polymorphic function doesn't depend on the actual type.

Polymorphic type

- Types can be **parametrised** - eg. [Int] - [a]
- A function that can be applied to values of different types is called as **polymorphic function**.

```
length :: [a] -> Int  // a is a type variable
hd     :: [a] -> a
```

- The functionality of the polymorphic function doesn't depend on the actual type.

Polymorphic type

- Types can be **parametrised** - eg. [Int] - [a]
- A function that can be applied to values of different types is called as **polymorphic function**.

```
length :: [a] -> Int  // a is a type variable
hd     :: [a] -> a
```

- The functionality of the polymorphic function doesn't depend on the actual type.

Functions on lists

- 1. Last element of a list
- 2. Every element but last
- 3. N-th element of a list
- 4. The first n elements of a list
- 5. Reverse a list

Functions on lists

- 1. Last element of a list
- 2. Every element but last
- 3. N-th element of a list
- 4. The first n elements of a list
- 5. Reverse a list

Functions on lists

- 1. Last element of a list
- 2. Every element but last
- 3. N-th element of a list
- 4. The first n elements of a list
- 5. Reverse a list

Functions on lists

- 1. Last element of a list
- 2. Every element but last
- 3. N-th element of a list
- 4. The first n elements of a list
- 5. Reverse a list

Functions on lists

- 1. Last element of a list
- 2. Every element but last
- 3. N-th element of a list
- 4. The first n elements of a list
- 5. Reverse a list

Solutions

1. Last element of a list

```
last [x]        = x
last [x : xs]   = last xs
last []         = abort "last of []"
```

2. Every element but last

```
init []         = []
init [x]        = []
init [x : xs]   = [x: init xs]
```

Solutions

1. Last element of a list

```
last [x]        = x
last [x : xs]   = last xs
last []         = abort "last of []"
```

2. Every element but last

```
init []         = []
init [x]        = []
init [x : xs]   = [x: init xs]
```

Solutions

3. N-th element of a list

```
index [x : xs] 0 = x
index [x : xs] n = index xs (n - 1)
index [] _       = abort "index out of range"
```

Usage: `index [1,2,3] 2`

With more confortable infix notation: `[1,2,3] !! 2`

```
(!!) infixl 9 :: [a] Int -> a
(!!) list i = index list i
```

4. The first n elements of a list

```
take 0 _        = []
take n [x : xs] = [x : take (n - 1) xs]
take n []       = []
```

Solutions

3. N-th element of a list

```
index [x : xs] 0 = x
index [x : xs] n = index xs (n - 1)
index [] _       = abort "index out of range"
```

Usage: `index [1,2,3] 2`

With more confortable infix notation: `[1,2,3] !! 2`

```
(!!) infixl 9 :: [a] Int -> a
(!!) list i = index list i
```

4. The first n elements of a list

```
take 0 _          = []
take n [x : xs] = [x : take (n - 1) xs]
take n []         = []
```

Solutions

3. N-th element of a list

```
index [x : xs] 0 = x
index [x : xs] n = index xs (n - 1)
index [] _       = abort "index out of range"
```

Usage: `index [1,2,3] 2`

With more confortable infix notation: `[1,2,3] !! 2`

```
(!!) infixl 9 :: [a] Int -> a
(!!) list i = index list i
```

4. The first n elements of a list

```
take 0 _          = []
take n [x : xs] = [x : take (n - 1) xs]
take n []         = []
```

Solutions

3. N-th element of a list

```
index [x : xs] 0 = x
index [x : xs] n = index xs (n - 1)
index [] _       = abort "index out of range"
```

Usage: `index [1,2,3] 2`
With more confortable infix notation: `[1,2,3] !!  2`

```
(!!) infixl 9 :: [a] Int -> a
(!!) list i = index list i
```

4. The first n elements of a list

```
take 0 _        = []
take n [x : xs] = [x : take (n - 1) xs]
take n []       = []
```

Solutions

5. Reverse a list

- 1st solution:

```
reverse []     = []
reverse [x:xs] = reverse xs ++ [x]
```

- 2nd solution:

```
reverse list = reverse_ list []
  where
    reverse_ [x:xs] acc = reverse_ xs [x:acc]
    reverse_ []     acc = acc
```

Solutions

5. Reverse a list

- 1st solution:

```
reverse []     = []
reverse [x:xs] = reverse xs ++ [x]
```

- 2nd solution:

```
reverse list = reverse_ list []
  where
    reverse_ [x:xs] acc = reverse_ xs [x:acc]
    reverse_ []     acc = acc
```

Functions on lists II.

- 6. Check two lists wether they are equal or not
- 7. Check two lists if the first is lexikographically less than the second

Functions on lists II.

- 6. Check two lists wether they are equal or not
- 7. Check two lists if the first is lexikographically less than the second

Solutions

6. Check two lists wether they are equal or not

```
eq [] []                = True
eq [a:as] [b:bs]
   | a == b             = as == bs
   | otherwise          = False
eq _ _                  = False
```

Solutions

7. Check two lists if the first is lexikographically less than the second

```
less [] []              = False
less [] _               = True
less _ []               = False
less [a:as] [b:bs]
       | a < b          = True
       | a > b          = False
       | otherwise      = as < bs
```

Higher order functions on lists

filter: selecting elements satisfying a property

```
filter :: (a -> Bool) [a] -> [a]
filter p [] = []
filter p [x : xs]
     | p x = [ x : filter p xs ]
     | otherwise = filter p xs
```

Higher order functions on lists

- 8.map: function applied elementwise (length is preserved)
- 9.foldr: elementwise consumer

Higher order functions on lists

- 8.map: function applied elementwise (length is preserved)
- 9.foldr: elementwise consumer

Solutions

8. map: function applied elementwise (length is preserved)

```
map :: (a -> b) [a] -> [b]
map f [] = []
map f [x : xs] = [ f x : map f xs ]
```

9. foldr: elemetwise consumer

```
foldr :: (a b -> b)  b  [a] -> b
foldr op e []       = e
foldr op e [x : xs] = op x (foldr op e xs)
```

Solutions

8. map: function applied elementwise (length is preserved)

```
map :: (a -> b) [a] -> [b]
map f [] = []
map f [x : xs] = [ f x : map f xs ]
```

9. foldr: elemetwise consumer

```
foldr :: (a b -> b)  b  [a] -> b
foldr op e []      = e
foldr op e [x : xs] = op x (foldr op e xs)
```

Exercise

- 10. Find the maximum of the list

10. find the maximum of the list

```
listmax :: [a] -> a | Ord a
listmax [x:xs] = foldl max x xs
  where
    max x y
      | x>y       = x
      | otherwise = y
```