Language for Calculational Programming

Zhenjiang Hu

The Graduate University for Advanced Studies

April 29, 2011

Language for Calculational Programming

What languages are we using for describing the Earth's orbit?

Language for Calculational Programming

What languages are we using for describing the Earth's orbit?

- A differential equation specifies how the Earth goes around the Sun.

$$m\frac{\mathrm{d}^2\mathbf{r}}{\mathrm{d}t^2} = -\mu\frac{\mathbf{r}}{r^3}$$

Language for Calculational Programming

What languages are we using for describing the Earth's orbit?

- A differential equation specifies how the Earth goes around the Sun.

$$m\frac{\mathrm{d}^2\mathbf{r}}{\mathrm{d}t^2} = -\mu\frac{\mathbf{r}}{r^3}$$

Then, what languages for program calculation?

Language for Calculational Programming

What languages are we using for describing the Earth's orbit?

- A differential equation specifies how the Earth goes around the Sun.

$$m\frac{\mathrm{d}^2\mathbf{r}}{\mathrm{d}t^2} = -\mu\frac{\mathbf{r}}{r^3}$$

Then, what languages for program calculation?

- A functional programming language plays the role because it

Language for Calculational Programming

What languages are we using for describing the Earth's orbit?

- A differential equation specifies how the Earth goes around the Sun.

$$m\frac{\mathrm{d}^2\mathbf{r}}{\mathrm{d}t^2} \;=\; -\mu\frac{\mathbf{r}}{r^3}$$

Then, what languages for program calculation?

- A functional programming language plays the role because it
  - provides compact notation based on mathematical concept, and

Language for Calculational Programming

What languages are we using for describing the Earth's orbit?

- A differential equation specifies how the Earth goes around the Sun.

$$m\frac{\mathrm{d}^2\mathbf{r}}{\mathrm{d}t^2} = -\mu\frac{\mathbf{r}}{r^3}$$

Then, what languages for program calculation?

- A functional programming language plays the role because it
  - provides compact notation based on mathematical concept, and
  - suitable for calculation based on equational reasoning

The Haskell Programming Language

Haskell is used in this course

The Haskell Programming Language

Haskell is used in this course

- It has been widely used in
  - development of practical applications, and
  - research on program transformation

The Haskell Programming Language

Haskell is used in this course

- It has been widely used in
  - development of practical applications, and
  - research on program transformation
- Language Definition and Systems available at

http://www.haskell.org/

The Haskell Programming Language

Haskell is used in this course

- It has been widely used in
  - development of practical applications, and
  - research on program transformation
- Language Definition and Systems available at

  http://www.haskell.org/

- Example sessions with GHCi or Hugs will be shown for demonstration

Expressions, Values, and Environments

Expressions, Values, and Environments

- An expression is a sequence of symbols like a formula in mathematics

Expressions, Values, and Environments

- An expression is a sequence of symbols like a formula in mathematics
- A value is an object of our concern

Expressions, Values, and Environments

- An expression is a sequence of symbols like a formula in mathematics
- A value is an object of our concern
  - Basic values: numbers, Booleans, characters, etc.

Expressions, Values, and Environments

- An expression is a sequence of symbols like a formula in mathematics
- A value is an object of our concern
  - Basic values: numbers, Booleans, characters, etc.
  - Compound values: sequences, lists, trees, etc.

Expressions, Values, and Environments

- An expression is a sequence of symbols like a formula in mathematics
- A value is an object of our concern
  - Basic values: numbers, Booleans, characters, etc.
  - Compound values: sequences, lists, trees, etc.
  - Functions

Expressions, Values, and Environments

- An expression is a sequence of symbols like a formula in mathematics
- A value is an object of our concern
  - Basic values: numbers, Booleans, characters, etc.
  - Compound values: sequences, lists, trees, etc.
  - Functions
- An environment is a correspondence of names to values

Expressions, Values, and Environments

- An expression is a sequence of symbols like a formula in mathematics
- A value is an object of our concern
  - Basic values: numbers, Booleans, characters, etc.
  - Compound values: sequences, lists, trees, etc.
  - Functions
- An environment is a correspondence of names to values
  - Definitions and evaluation of some constructs produce bindings names to values.

Expressions, Values, and Environments

- An expression is a sequence of symbols like a formula in mathematics
- A value is an object of our concern
  - Basic values: numbers, Booleans, characters, etc.
  - Compound values: sequences, lists, trees, etc.
  - Functions
- An environment is a correspondence of names to values
  - Definitions and evaluation of some constructs produce bindings names to values.
  - Environment changes according to the scope of the name.

Evaluation

Evaluation is a process of simplifying expressions by execution of program.

Example session of GHCi

```
? 3*8
24
?
```

Definition

Definition is used to give a name to a value and is expressed as NAME $=$ RHS.

Definition

Definition is used to give a name to a value and is expressed as NAME = RHS.

Definition

x = 3 * 8

Definition

Definition is used to give a name to a value and is expressed as NAME = RHS.

Definition

x = 3 * 8

- The value is the result of evaluation of RHS expression.

Definition

Definition is used to give a name to a value and is expressed as NAME = RHS.

Definition

x = 3 * 8

- The value is the result of evaluation of RHS expression.

The name is used to refer to the value in expressions.

Definition

Definition is used to give a name to a value and is expressed as NAME = RHS.

Definition

x = 3 * 8

- The value is the result of evaluation of RHS expression.

The name is used to refer to the value in expressions.

Name in an expression

x+21

Variables

Variable is a name in expressions.

Variables

Variable is a name in expressions.

- Names defined by Definitions are variables.

Variables

Variable is a name in expressions.

- Names defined by Definitions are variables.
- Names are also used for other purposes than variables.

Variables

Variable is a name in expressions.

- Names defined by Definitions are variables.
- Names are also used for other purposes than variables.
- Values represented by the variable may vary according to bindings, and not by assignment of procedural languages.

Environment

Environment is a correspondence of names to values they represent.

Environment

Environment is a correspondence of names to values they represent.

- Under an environment

  $$\{\ x{=}3{*}8,\ y{=}x{+}21\ \}$$

  evaluation gives

Environment

Environment is a correspondence of names to values they represent.

- Under an environment

$$\{\ x{=}3{*}8,\ y{=}x{+}21\ \}$$

evaluation gives

Example session of GHCi

```
? y
45
?
```

Local Declaration

Local declaration is a definition for local variables.

Local Declaration

Local declaration is a definition for local variables.

- Definition with local declarations

Local Declaration

Local declaration is a definition for local variables.

- Definition with local declarations
  - Environment produced by the definition after "where" is local to RHS of definition.

Definition with local declaration

x = y * (y + 5) where y=3

Local Declaration

Local declaration is a definition for local variables.

- Definition with local declarations
  - Environment produced by the definition after "where" is local to RHS of definition.

Definition with local declaration

x = y * (y + 5) where y=3

- Expression with local declarations

Local Declaration

Local declaration is a definition for local variables.

- Definition with local declarations
  - Environment produced by the definition after "where" is local to RHS of definition.

Definition with local declaration

x = y * (y + 5) where y=3

- Expression with local declarations
  - Environment produced by the definition is local to expr after "in".

Expression with local declaration

let y=3 in y * (y + 5)

Operators and Functions

Operators and Functions

- Expressions are composed of operators and operands

Operators and Functions

- Expressions are composed of operators and operands
  - Operators combine operands.

Operators and Functions

- Expressions are composed of operators and operands
  - Operators combine operands.
  - Operators represent operations associated by the environment

Operators and Functions

- Expressions are composed of operators and operands
  - Operators combine operands.
  - Operators represent operations associated by the environment
- Operators are defined by definitions which associate names with values

Operators and Functions

- Expressions are composed of operators and operands
  - Operators combine operands.
  - Operators represent operations associated by the environment
- Operators are defined by definitions which associate names with values
  - Operations are values same as numbers are.

Operators and Functions

- Expressions are composed of operators and operands
  - Operators combine operands.
  - Operators represent operations associated by the environment
- Operators are defined by definitions which associate names with values
  - Operations are values same as numbers are.
- Functions are generalization of operators; only their appearance may differ.

Operators and Functions

- Expressions are composed of operators and operands
  - Operators combine operands.
  - Operators represent operations associated by the environment
- Operators are defined by definitions which associate names with values
  - Operations are values same as numbers are.
- Functions are generalization of operators; only their appearance may differ.
- Expressions are constructed using functions as combinators of operands.

Functions

- Functions combine expression components to construct an expression, and

Functions

- Functions combine expression components to construct an expression, and stand for mapping.

Functions

- Functions combine expression components to construct an expression, and stand for mapping.
- Every function has its source and target.

$$f :: a -> b$$

Functions

- Functions combine expression components to construct an expression, and stand for mapping.
- Every function has its source and target.

$$f :: a \rightarrow b$$

  - Source and target of the function is sometimes called functionality.
- Argument is a value in the source, and

Functions

- Functions combine expression components to construct an expression, and stand for mapping.
- Every function has its source and target.

  $$f :: a -> b$$

  - Source and target of the function is sometimes called functionality.
- Argument is a value in the source, and
- Result is the value in the target corresponding to the argument.

Functions

- Functions combine expression components to construct an expression, and stand for mapping.
- Every function has its source and target.

$$f :: a -> b$$

  - Source and target of the function is sometimes called functionality.
- Argument is a value in the source, and
- Result is the value in the target corresponding to the argument.
- Functional application yields the result for the given argument.

Functional Abstraction

- Functional abstraction is a notation to express the function itself by specifying parameters.

Functional Abstraction

- Functional abstraction is a notation to express the function itself by specifying parameters.

Functional abstraction

```
\x -> x * x
```

Functional Abstraction

- Functional abstraction is a notation to express the function itself by specifying parameters.

Functional abstraction

```
\x -> x * x
```

  - represents a function by abstracting pattern from 1*1, 2*2, ...

Functional Abstraction

- Functional abstraction is a notation to express the function itself by specifying parameters.

Functional abstraction

```
\x -> x * x
```

  - represents a function by abstracting pattern from 1*1, 2*2, ...

- Repeated abstractions are possible for more parameters.

Repeated functional abstraction

```
\p ->(\x -> p * x)
```

Functional Abstraction

- Functional abstraction is a notation to express the function itself by specifying parameters.

Functional abstraction

```
\x -> x * x
```

  - represents a function by abstracting pattern from 1*1, 2*2, ...

- Repeated abstractions are possible for more parameters.

Repeated functional abstraction

```
\p ->(\x -> p * x)
```

- "\\" is a substitute for "λ" in lambda calculus; e.g., $\lambda x.x * x$

Functional Application

- Functional application is a notation to express the function applied to arguments.

$$f\ a$$

Functional Application

- Functional application is a notation to express the function applied to arguments.

$$f\ a$$

  - Simply juxtapose function with arguments.

Functional Application

- Functional application is a notation to express the function applied to arguments.

$$f\ a$$

  - Simply juxtapose function with arguments.
  - No parentheses required for arguments.

Functional Application

- Functional application is a notation to express the function applied to arguments.

$$f\ a$$

  - Simply juxtapose function with arguments.
  - No parentheses required for arguments.

Example session of GHCi

```
? (\x -> x * x) 5
25
? (\p ->(\x -> p * x) ) 2 5
10
?
```

Function Definitions

- Function definitions are expressed as LHS = RHS.

Function Definitions

- Function definitions are expressed as LHS = RHS.

Function definition

square x = x * x

Function Definitions

- Function definitions are expressed as LHS = RHS.

Function definition

square x = x * x

  - LHS: Function name followed by parameters

Function Definitions

- Function definitions are expressed as LHS = RHS.

Function definition

```
square x = x * x
```

- LHS: Function name followed by parameters
  - No parentheses required for parameters.

Function Definitions

- Function definitions are expressed as LHS = RHS.

Function definition

square x = x * x

- LHS: Function name followed by parameters
  - No parentheses required for parameters.
- RHS: Expression consists of parameters and others.

Function Definitions

- Function definitions are expressed as LHS = RHS.

Function definition

square x = x * x

  - LHS: Function name followed by parameters
    - No parentheses required for parameters.
  - RHS: Expression consists of parameters and others.
- Function definition adds an association of the function name and the function to the environment.

Function Definitions

- Function definitions are expressed as LHS = RHS.

Function definition

square x = x * x

  - LHS: Function name followed by parameters
    - No parentheses required for parameters.
  - RHS: Expression consists of parameters and others.
- Function definition adds an association of the function name and the function to the environment.
- Functionality may be declared with the definition.

Functional definition with type declaration

square :: Int −>Int
square x = x * x

Defining Function by Abstraction

- Function definition

  $$square\ x = x * x$$

  is same as

Defining Function by Abstraction

- Function definition

  $$square\ x = x * x$$

  is same as

  $$square = \backslash x \text{->} x * x$$

- Function definition is not special, but is simply a definition of a value.

Function Definitions with Local Declarations

- Function definitions may be local.

Function Definitions with Local Declarations

- Function definitions may be local.

Function definition with local declaration

```
sumsquares x y = square x + square y
                 where square z = z * z
```

Function Definitions with Local Declarations

- Function definitions may be local.

Function definition with local declaration

```
sumsquares x y = square x + square y
                 where square z = z * z
```

- The definition may be equally expressed as

Function Definitions with Local Declarations

- Function definitions may be local.

Function definition with local declaration

sumsquares x y = square x + square y
    where square z = z * z

- The definition may be equally expressed as

Function definition by expression with local declaration

sumsquares x y =
  let square z = z * z in square x + square y

Converting Operators into Functions

- Sectioning makes operator into function by parenthesizing.

Converting Operators into Functions

- Sectioning makes operator into function by parenthesizing.

Operator sectioning

```
? (*) 3 8
24
?
```

Converting Operators into Functions

- Sectioning makes operator into function by parenthesizing.

Operator sectioning

```
? (*) 3 8
24
?
```

- Binary operators become unary functions by sectioning.

Converting Operators into Functions

- Sectioning makes operator into function by parenthesizing.

Operator sectioning

```
? (*) 3 8
24
?
```

- Binary operators become unary functions by sectioning.

Sectioning operators to get unary functions

```
? (3*) 3
24
? (*8) 3
24
?
```

Converting Functions into Operators

- Embracing function name by grave accents (backquotes) ‘ makes a binary function into a binary operator.

Converting Functions into Operators

- Embracing function name by grave accents (backquotes) ‘ makes a binary function into a binary operator.

Binary function as an operator and user-defined operator

```
? 3 ‘sumsquares‘ 4
25
?
```

Precedence of Operators

- Precedence determines which of adjacent operators takes operand in-between.

Precedence of Operators

- Precedence determines which of adjacent operators takes operand in-between.
  - Example: multiplication (*) has higher precedence than addition (+)

Precedence of Operators

- Precedence determines which of adjacent operators takes operand in-between.
  - Example: multiplication (*) has higher precedence than addition (+)
- Precedence of operators reduces parentheses in expressions.

Precedence of Operators

- Precedence determines which of adjacent operators takes operand in-between.
  - Example: multiplication (*) has higher precedence than addition (+)
- Precedence of operators reduces parentheses in expressions.
  - Example: a+b*c is same as a+(b*c)

Association Order of Operators

- Association order determines which of adjacent operators of the same precedence takes operand in-between.

Association Order of Operators

- Association order determines which of adjacent operators of the same precedence takes operand in-between.
  - Example: a-b-c is equal to (a-b)-c

Association Order of Operators

- Association order determines which of adjacent operators of the same precedence takes operand in-between.
  - Example: a-b-c is equal to (a-b)-c
- Left associative operators (e.g., subtraction), and
- Right associative operators (e.g., exponentiation).

Association Order of Operators

- Association order determines which of adjacent operators of the same precedence takes operand in-between.
    - Example: a-b-c is equal to (a-b)-c
- Left associative operators (e.g., subtraction), and
- Right associative operators (e.g., exponentiation).
- Operator may be neither left nor right associative.

Association Order of Operators

- Association order determines which of adjacent operators of the same precedence takes operand in-between.
  - Example: a-b-c is equal to (a-b)-c
- Left associative operators (e.g., subtraction), and
- Right associative operators (e.g., exponentiation).
- Operator may be neither left nor right associative.
- Association order is not related to the associativity of operators.
  - Associativity is an algebraic property of operators:

$$(x \oplus y) \oplus z \;=\; x \oplus (y \oplus z)$$

Declaration of Precedence and Association Order

Operator Declaration

Declaration of Precedence and Association Order

Operator Declaration

- infixl r $\oplus$ declares

Declaration of Precedence and Association Order

Operator Declaration

- infixl r $\oplus$ declares
  - $\oplus$ has precedence r

Declaration of Precedence and Association Order

Operator Declaration

- infixl r $\oplus$ declares
  - $\oplus$ has precedence r
  - $\oplus$ is left associative

Declaration of Precedence and Association Order

Operator Declaration

- infixl r $\oplus$ declares
  - $\oplus$ has precedence r
  - $\oplus$ is left associative
- infixr r $\oplus$ declares
  - $\oplus$ has precedence r
  - $\oplus$ is right associative

Declaration of Precedence and Association Order

Operator Declaration

- infixl r $\oplus$ declares
  - $\oplus$ has precedence r
  - $\oplus$ is left associative
- infixr r $\oplus$ declares
  - $\oplus$ has precedence r
  - $\oplus$ is right associative
- infix r $\oplus$ declares
  - $\oplus$ has precedence r
  - $\oplus$ is niether left nor right associative

Declaration of Precedence and Association Order

Operator Declaration

- infixl r $\oplus$ declares
  - $\oplus$ has precedence r
  - $\oplus$ is left associative
- infixr r $\oplus$ declares
  - $\oplus$ has precedence r
  - $\oplus$ is right associative
- infix r $\oplus$ declares
  - $\oplus$ has precedence r
  - $\oplus$ is niether left nor right associative
- Operators with no declaration as "infix 9"

Standard Operators

```
infixl 9 !!
infixr 9 .
infixr 8 ^
infixl 7 *
infix  7 /, `div`, `rem`, `mod`
infixl 6 +, -
infix  5 \\
infixr 5 ++
infix  4 ==, /=, <, <=, >, >=
infix  4 elem`,`notElem`
infixr 3 &&
infixr 2 ||
```

Higher Order Functions

- Higher order function is a functions which takes functions as its arguments, or returns a function as its result.

Higher Order Functions

- Higher order function is a functions which takes functions as its arguments, or returns a function as its result.
- Functional composition produces a new function

  $$f\ .\ g$$

  from two functions

  $$f :: b -> c, \quad g :: a -> b$$

Higher Order Functions

- Higher order function is a functions which takes functions as its arguments, or returns a function as its result.
- Functional composition produces a new function

  $$f\ .\ g$$

  from two functions

  $$f :: b \rightarrow c, \quad g :: a \rightarrow b$$

  defined as

  $$(f\ .\ g)\ x = f\ (g\ x)$$

- Composing function (.) is a higher order function which
  - takes two functions as arguments, and
  - returns a function as result
- Functional composition is written as $f \circ g$ in mathematics

Partial Application of Functions

- Partial application is a functional application with partial arguments,

Partial Application of Functions

- Partial application is a functional application with partial arguments, which may produce a new function as its result.

Partial Application of Functions

- Partial application is a functional application with partial arguments, which may produce a new function as its result.
    - The add function (+) takes two integers to return an integer.

$$(+) = \backslash x \text{ -> } (\backslash y \text{ -> } (x + y))$$

Partial Application of Functions

- Partial application is a functional application with partial arguments, which may produce a new function as its result.
  - The add function $(+)$ takes two integers to return an integer.

$$(+) = \backslash x \to (\backslash y \to (x + y))$$

  - By giving an integer to this function to get another function

$$(1+) = (+)\ 1 = \backslash y \to (1 + y)$$

Partial Application of Functions

- Partial application is a functional application with partial arguments, which may produce a new function as its result.
  - The add function (+) takes two integers to return an integer.

$$(+) = \backslash x \text{ -> } (\backslash y \text{ -> } (x + y))$$

  - By giving an integer to this function to get another function

$$(1+) = (+)\ 1 = \backslash y \text{ -> } (1 + y)$$

  - Functionality of (+) is

$$(+) :: Int \text{ -> } (Int \text{ -> } Int)$$

Partial Application of Functions

- Partial application is a functional application with partial arguments, which may produce a new function as its result.
  - The add function (+) takes two integers to return an integer.

    $$(+) = \backslash x \text{ -> } (\backslash y \text{ -> } (x + y))$$

  - By giving an integer to this function to get another function

    $$(1+) = (+)\ 1 = \backslash y \text{ -> } (1 + y)$$

  - Functionality of (+) is

    $$(+) :: Int \text{ -> } (Int \text{ -> } Int)$$

  - Functionality of (1+) is

    $$(1+) :: Int \text{ -> } Int$$

Recursive Definitions

- Function definition may be recursive;

Recursive Definitions

- Function definition may be recursive; RHS of the definition may contain the function name to be defined.

Recursive definition of the factorial function

factorial 0 = 1
factorial (n+1) = (n+1) * factorial n

Recursive Definitions

- Function definition may be recursive; RHS of the definition may contain the function name to be defined.

Recursive definition of the factorial function

factorial 0 = 1
factorial (n+1) = (n+1) * factorial n

- Computation of functional application proceeds as ...
  - compare functional application pattern with LHS
  - arrange the parameter to replace the application with RHS
- Function may be defined using conditional expression

Recursive definition with conditional expression

```
factorial n =   if n==0 then 1
                        else n * factorial (n-1)
```

Data Types

- Data type is a set of values with operations

Data Types

- Data type is a set of values with operations
- Basic types: Type classes defined in Standard prelude in Haskell

Data Types

- Data type is a set of values with operations
- Basic types: Type classes defined in Standard prelude in Haskell
  - Int: integers

Data Types

- Data type is a set of values with operations
- Basic types: Type classes defined in Standard prelude in Haskell
  - Int: integers
  - Bool: Boolean values True and False

Data Types

- Data type is a set of values with operations
- Basic types: Type classes defined in Standard prelude in Haskell
  - Int: integers
  - Bool: Boolean values True and False
  - Char: characters

Data Types

- Data type is a set of values with operations
- Basic types: Type classes defined in Standard prelude in Haskell
  - Int: integers
  - Bool: Boolean values True and False
  - Char: characters
  - Float: floating point numbers

Data Types

- Data type is a set of values with operations
- Basic types: Type classes defined in Standard prelude in Haskell
  - Int: integers
  - Bool: Boolean values True and False
  - Char: characters
  - Float: floating point numbers
- Lists: a set of values of finite sequences of values of a component type

Data Types

- Data type is a set of values with operations
- Basic types: Type classes defined in Standard prelude in Haskell
  - Int: integers
  - Bool: Boolean values True and False
  - Char: characters
  - Float: floating point numbers
- Lists: a set of values of finite sequences of values of a component type
- Tuples: a set of values of tuples of values of any type

Data Types

- Data type is a set of values with operations
- Basic types: Type classes defined in Standard prelude in Haskell
  - Int: integers
  - Bool: Boolean values True and False
  - Char: characters
  - Float: floating point numbers
- Lists: a set of values of finite sequences of values of a component type
- Tuples: a set of values of tuples of values of any type
- Functions: a set of functions of which source and target are of any types

Int

- Notation for constants: decimal positional representation using digits '0' – '9'

Int

- Notation for constants: decimal positional representation using digits '0' – '9'
- Operations: $+$, $-$, *, ‘div‘, ‘rem‘

Int

- Notation for constants: decimal positional representation using digits '0' – '9'
- Operations: $+$, –, *, ‘div‘, ‘rem‘
  - Negation operation by function rather than prefix –

    negate :: Int –> Int

Int

- Notation for constants: decimal positional representation using digits '0' – '9'
- Operations: $+$, –, *, ‘div‘, ‘rem‘
  - Negation operation by function rather than prefix –

    negate :: Int –> Int

- Functions for judgement of even/odd

  odd :: Int –> Bool
  even :: Int –> Bool

Bool

- Boolean values are True and False

Bool

- Boolean values are True and False
- Operations

Bool

- Boolean values are True and False
- Operations
- logical or represented by ||
- logical and represented by &&

Bool

- Boolean values are True and False
- Operations
- logical or represented by ||
- logical and represented by &&
- Logical negation by function

$$not :: Bool -> Bool$$

Char

- Notation for constants: a single character enclosed by apostrophes ’.

Char

- Notation for constants: a single character enclosed by apostrophes ’.
  - Apostrophe and other special characters are escaped by \

Char

- Notation for constants: a single character enclosed by apostrophes '.
  - Apostrophe and other special characters are escaped by \
- Function returning corresponding integer value for character and its inverse.

$$ord :: Char \mathrel{-}\!\!> Int$$
$$chr :: Int \mathrel{-}\!\!> Char$$

Char

- Notation for constants: a single character enclosed by apostrophes '.
  - Apostrophe and other special characters are escaped by \
- Function returning corresponding integer value for character and its inverse.

$$ord :: Char -> Int$$
$$chr :: Int -> Char$$

Function ord

```
? ord '9' - ord '0'
9
? ord 'n' - ord 'b'
12
?
```

Lists

- A List is a sequence of values of the same type.

Lists

- A List is a sequence of values of the same type.
- List type of which elements are of type `t` is written as `[t]`.

Lists

- A List is a sequence of values of the same type.
- List type of which elements are of type `t` is written as `[t]`.
- A value of type `[t]` is either

Lists

- A List is a sequence of values of the same type.
- List type of which elements are of type `t` is written as `[t]`.
- A value of type `[t]` is either
  - an empty list denoted by `[]`, or

Lists

- A List is a sequence of values of the same type.
- List type of which elements are of type `t` is written as `[t]`.
- A value of type `[t]` is either
  - an empty list denoted by `[]`, or
  - a list composed of an element x of type `t` and a list xs of type `[t]` with the constructor :, i.e., x : xs.

Lists

- A List is a sequence of values of the same type.
- List type of which elements are of type `t` is written as `[t]`.
- A value of type `[t]` is either
  - an empty list denoted by `[]`, or
  - a list composed of an element x of type `t` and a list xs of type `[t]` with the constructor `:`, i.e., x `:` xs.
- Non-empty list is a sequence of enumerated elements.

List by enumeration

[1, 3, 7, 5]
[1..5]

List Constructor

- An empty list is produced with nil constructor `[]`.
- A non-empty list is composed with cons constructor `:`.

List Constructor

- An empty list is produced with nil constructor [].
- A non-empty list is composed with cons constructor :.
- Constructor is a function which produces values.

List Constructor

- An empty list is produced with nil constructor `[]`.
- A non-empty list is composed with cons constructor `:`.
- Constructor is a function which produces values.

$$[] :: [a]$$
$$(:) :: a \rightarrow [a] \rightarrow [a]$$

List Constructor

- An empty list is produced with nil constructor [].
- A non-empty list is composed with cons constructor :.
- Constructor is a function which produces values.

$$[] :: [a]$$
$$(:) :: a \rightarrow [a] \rightarrow [a]$$

List by construction

$$
\begin{aligned}
[1, 2, 3] &= 1 : [2, 3] \\
&= 1 : 2 : [3] \\
&= 1 : 2 : 3 : [\,]
\end{aligned}
$$

Concatenation Operator ++

- Concatenation
  operator ++ connects two lists into one.

  $$(++) :: [a] \to [a] \to [a]$$

Concatenation Operator ++

- Concatenation
  operator ++ connects two lists into one.

$$(++) :: [a] \rightarrow [a] \rightarrow [a]$$

- Operator ++ is right associative and has associativity property.

Concatenation Operator ++

- Concatenation
  operator ++ connects two lists into one.

  $$(++) :: [a] \rightarrow [a] \rightarrow [a]$$

- Operator ++ is right associative and has associativity property.

Example of ++

```
? [1,2,3]++[4,5]
 [1,2,3,4,5]
? [1,2,3]++[ ]
 [1,2,3]
? [ ]++[4,5]
 [4,5]
? [ ]++[ ]
 [ ]
?
```

List Comprehension

- A list may be expressed as comprehension
  [E | GEN ...]
  - E: expression
  - GEN: generator

List Comprehension

- A list may be expressed as comprehension
  [E | GEN ...]
  - E: expression
  - GEN: generator
- Conditions may be expressed by Boolean expressions.

List comprehension

```
? [x+1 | x <- [1, 2, 3]]
 [2, 3, 4]
? [x+y | x<- [10, 20], y <- [4, 5]]
```

List Comprehension

- A list may be expressed as comprehension
  [E | GEN ...]
  - E: expression
  - GEN: generator
- Conditions may be expressed by Boolean expressions.

List comprehension

```
? [x+1 | x <- [1, 2, 3]]
 [2, 3, 4]
? [x+y | x<- [10, 20], y <- [4, 5]]
 [14, 15, 24, 25]
? [x-y | x <- [1, 2, 3], y <- [ ]]
```

List Comprehension

- A list may be expressed as comprehension
  [E | GEN ...]
  - E: expression
  - GEN: generator
- Conditions may be expressed by Boolean expressions.

List comprehension

```
? [x+1 | x <- [1, 2, 3]]
[2, 3, 4]
? [x+y | x<- [10, 20], y <- [4, 5]]
[14, 15, 24, 25]
? [x-y | x <- [1, 2, 3], y <- [ ]]
[ ]
?
```

List comprehension with condition

```
? [x | x <- [0..9], even x]
[0, 2, 4, 6, 8]
?
```

Strings

- A string is a list of characters of type `[Char]`.

Strings

- A string is a list of characters of type `[Char]`.
- A string constant may be expressed by enclosing characters with quotation marks `"`.

Strings

- A string is a list of characters of type `[Char]`.
- A string constant may be expressed by enclosing characters with quotation marks `"`.

Strings

```
? 'c':['a','l','c','u','l','a','t','i','o','n']
 calculation
? "calculation"
 calculation
?
```

- Strings can be manipulated in the same way as general lists.

Tuples

- A tuple is a combination of a fixed number of elements;

Tuples

- A tuple is a combination of a fixed number of elements; a pair is a 2-tuple,

Tuples

- A tuple is a combination of a fixed number of elements; a pair is a 2-tuple, a triple is a 3-tuple, and so on.

Tuples

- A tuple is a combination of a fixed number of elements; a pair is a 2-tuple, a triple is a 3-tuple, and so on.
- n-tuple of elements of type t_i has type $(t_1, t_2, \cdots, t_n)$

Tuples

- A tuple is a combination of a fixed number of elements; a pair is a 2-tuple, a triple is a 3-tuple, and so on.
- n-tuple of elements of type t_i has type $(t_1, t_2, \cdots, t_n)$
- Tuples are expressed by enumerating elements in (and).

Tuples

- A tuple is a combination of a fixed number of elements; a pair is a 2-tuple, a triple is a 3-tuple, and so on.
- n-tuple of elements of type t_i has type $(t_1, t_2, \cdots, t_n)$
- Tuples are expressed by enumerating elements in (and).

Tuples

(1, 2) :: (Int, Int)
(1, True, [2, 3, 4]) :: (Int, Bool, [Int])

Tuples

- A tuple is a combination of a fixed number of elements; a pair is a 2-tuple, a triple is a 3-tuple, and so on.
- n-tuple of elements of type t_i has type $(t_1, t_2, \cdots, t_n)$
- Tuples are expressed by enumerating elements in (and).

Tuples

(1, 2) :: (Int, Int)
(1, True, [2, 3, 4]) :: (Int, Bool, [Int])

- Elements of tuples may be any expressions.

Tuples

- A tuple is a combination of a fixed number of elements; a pair is a 2-tuple, a triple is a 3-tuple, and so on.
- n-tuple of elements of type t_i has type $(t_1, t_2, \cdots, t_n)$
- Tuples are expressed by enumerating elements in (and).

Tuples

(1, 2) :: (Int, Int)
(1, True, [2, 3, 4]) :: (Int, Bool, [Int])

- Elements of tuples may be any expressions.

Tuples in list comprehension

? [(x+y, x*y) | x<– [10, 20], y <– [4, 5]]

Tuples

- A tuple is a combination of a fixed number of elements; a pair is a 2-tuple, a triple is a 3-tuple, and so on.
- n-tuple of elements of type t_i has type $(t_1, t_2, \cdots, t_n)$
- Tuples are expressed by enumerating elements in (and).

Tuples

```
(1, 2) :: (Int, Int)
(1, True, [2, 3, 4]) :: (Int, Bool, [Int])
```

- Elements of tuples may be any expressions.

Tuples in list comprehension

```
? [(x+y, x*y) | x<- [10, 20], y <- [4, 5]]
 [(14, 40), (15, 50), (24, 80), (25, 100)]
?
```

Function Data Types

- A function with source type s and target type t is of the type $s->t$.

Function Data Types

- A function with source type s and target type t is of the type $s- > t$.
- A function with source type s and target type $s'- > t'$ is of the type $s- > (s'- > t')$.

Function Data Types

- A function with source type s and target type t is of the type $s- > t$.
- A function with source type s and target type $s'- > t'$ is of the type $s- > (s'- > t')$.
  - Such functions may take two arguments;

Function Data Types

- A function with source type s and target type t is of the type $s- > t$.
- A function with source type s and target type $s'- > t'$ is of the type $s- > (s'- > t')$.
  - Such functions may take two arguments; one of type s and one of type s' to get the result of type t'.

Function Data Types

- A function with source type s and target type t is of the type $s- > t$.
- A function with source type s and target type $s' - > t'$ is of the type $s- > (s' - > t')$.
  - Such functions may take two arguments; one of type s and one of type s' to get the result of type t'.
  - This type may be written as $s- > s' - > t'$.

Function Data Types

- A function with source type s and target type t is of the type $s- > t$.
- A function with source type s and target type $s'- > t'$ is of the type $s- > (s'- > t')$.
  - Such functions may take two arguments; one of type s and one of type s' to get the result of type t'.
  - This type may be written as $s- > s'- > t'$.

Functions with two parameters

f :: (Int, Int) –> Int
f (x,y) = x*x + y*y
f' :: Int –> Int –> Int
f' x y = x*x + y*y

Curry and Uncurry Functions

- Functions *curry* and *uncurry* convert functions with a single pair into one with two parameters, and vice versa.

Curry and Uncurry Functions

- Functions *curry* and *uncurry* convert functions with a single pair into one with two parameters, and vice versa.

Functions with two parameters

```
curry :: ((a, b) -> c) -> (a -> b -> c)
curry f x y = f (x, y)
uncurry :: (a -> b -> c) -> ((a, b) -> c)
uncurry f' (x, y) = f' x y
```

Curry and Uncurry Functions

- Functions *curry* and *uncurry* convert functions with a single pair into one with two parameters, and vice versa.

Functions with two parameters

```
curry :: ((a, b) -> c) -> (a -> b -> c)
curry f x y = f (x, y)
uncurry :: (a -> b -> c) -> ((a, b) -> c)
uncurry f' (x, y) = f' x y
```

Function uncurry

```
? sumsquares 3 4
 25
? uncurry sumsquares (3, 4)
 25
?
```

Algebraic Types

- Algebraic types are defined by construction;

Algebraic Types

- Algebraic types are defined by construction; constructors produce values from values of existent types.

Algebraic Types

- Algebraic types are defined by construction; constructors produce values from values of existent types.

Algebraic type by enumeration

```
data Day = Sun |Mon |Tue
           |Wed |Thu |Fri |Sat
```

- Constructor has a name with capital letter as its initial character.

Algebraic Types

- Algebraic types are defined by construction; constructors produce values from values of existent types.

Algebraic type by enumeration

```
data Day = Sun |Mon |Tue
           |Wed |Thu |Fri |Sat
```

- Constructor has a name with capital letter as its initial character.

Function on algebraic type

```
workday :: Day -> Bool
workday Sun = False
workday Mon = True
workday Tue = True
workday Wed = True
workday Thu = True
workday Fri = True
workday Sat = False
```

Type Classes and Instances

- Declaration of instances of type classes can make operations inherited from the class.

Type Classes and Instances

- Declaration of instances of type classes can make operations inherited from the class.

Instance declaration of type class Eq

```
instance Eq Day where Sun == Sun   = True
                      Mon == Mon   = True
                      Tue == Tue   = True
                      Wed == Wed   = True
                      Thu == Thu   = True
                      Fri == Fri   = True
                      Sat == Sat   = True
                      _  ==  _     = False
workday d | d==Sun || d==Sat = False
          | otherwise        = True
```

Algebraic Types by Construction

- Enumerating constructions produces new algebraic data types.

Algebraic Types by Construction

- Enumerating constructions produces new algebraic data types.

Figures of Circles and Rectangles

data Figure = Circle Float | Rectangle Float Float

- Circle with radius 5.2 is represented as *Circle* 5.2
- Rectabgles with sides 3.2 and 2.5 as *Rectangle* 3.2 2.5

Algebraic Types by Construction

- Enumerating constructions produces new algebraic data types.

Figures of Circles and Rectangles

data Figure = Circle Float | Rectangle Float Float

  - Circle with radius 5.2 is represented as *Circle* 5.2
  - Rectabgles with sides 3.2 and 2.5 as *Rectangle* 3.2 2.5

- Constructors of the defined type has functionality with this type as target.

Functionality of constructors

Circle :: Float −> Figure
Rectangle :: Float −> Float −> Figure

Algebraic Types by Recursive Construction

- Recursive algebraic types are defined by recursion in type definition.

Algebraic Types by Recursive Construction

- Recursive algebraic types are defined by recursion in type definition.

Peano numerals

data Nat = Zero | Succ Nat

  - *Zero* is interpreted as numeral 0,

Algebraic Types by Recursive Construction

- Recursive algebraic types are defined by recursion in type definition.

Peano numerals

data Nat = Zero | Succ Nat

  - *Zero* is interpreted as numeral 0, *Succ Zero* as 1,

Algebraic Types by Recursive Construction

- Recursive algebraic types are defined by recursion in type definition.

Peano numerals

data Nat = Zero | Succ Nat

- *Zero* is interpreted as numeral 0, *Succ Zero* as 1, *Succ*(*Succ Zero*) as 2, ...

Algebraic Types by Recursive Construction

- Recursive algebraic types are defined by recursion in type definition.

Peano numerals

data Nat = Zero | Succ Nat

  - *Zero* is interpreted as numeral 0, *Succ Zero* as 1, *Succ*(*Succ Zero*) as 2, ...

Redefinition of the list type

data List a = Nil | Cons a (List a)

  - Type *List a* has type *a* as a parameter for element type.
  - Haskell provides notational conventions for lists:

Algebraic Types by Recursive Construction

- Recursive algebraic types are defined by recursion in type definition.

Peano numerals

data Nat = Zero | Succ Nat

- *Zero* is interpreted as numeral 0, *Succ Zero* as 1, *Succ*(*Succ Zero*) as 2, ...

Redefinition of the list type

data List a = Nil | Cons a (List a)

- Type *List a* has type *a* as a parameter for element type.
- Haskell provides notational conventions for lists: [ ] for *Nil*, : for ‘*Cons*‘, and [*a*] for *List a*.

Patterns in Function Definitions

- Functions over algebraic data type can be defined by cases of patterns according to construction methods.

Patterns in Function Definitions

- Functions over algebraic data type can be defined by cases of patterns according to construction methods.
- Definition by case enumeration as in *workday* is an example.
- Parameters in pattern are bound to values when match found.

Areas of figures

```
area :: Figure -> Float
area (Circle r) = 3.14 * r * r
area (Rectangle x y) = x * y
```

Patterns for Lists and Tuples

- Algebraic types of lists and tuples are provided special notational conventions for patterns as well as constructions.

Patterns for Lists and Tuples

- Algebraic types of lists and tuples are provided special notational conventions for patterns as well as constructions.

Patterns for lists

```
head :: [a] -> a
head (x:xs) = x
tail :: [a] -> [a]
tail (x:xs) = xs
```

Patterns for Lists and Tuples

- Algebraic types of lists and tuples are provided special notational conventions for patterns as well as constructions.

Patterns for lists

```
head :: [a] -> a
head (x:xs) = x
tail :: [a] -> [a]
tail (x:xs) = xs
```

Patterns for pairs

```
fst :: (a, b) -> a
fst (x, y) = x
snd :: (a, b) -> b
snd (x, y) = y
```

- Wild character _ may be used for unreferenced parameters;

Patterns for Lists and Tuples

- Algebraic types of lists and tuples are provided special notational conventions for patterns as well as constructions.

Patterns for lists

```
head :: [a] -> a
head (x:xs) = x
tail :: [a] -> [a]
tail (x:xs) = xs
```

Patterns for pairs

```
fst :: (a, b) -> a
fst (x, y) = x
snd :: (a, b) -> b
snd (x, y) = y
```

- Wild character _ may be used for unreferenced parameters; $head(x : \_) = x$, $snd(\_, y) = y$.

Standard Functions for Lists

- Lists are most popular data structure in calculational programming.

Standard Functions for Lists

- Lists are most popular data structure in calculational programming.
- Only a few standard functions are enough even for description of advanced algorithms.

Map function

- The map function applies the given function to every element of the given list.

Map function

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

Map function

- The map function applies the given function to every element of the given list.

Map function

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

Map function

```
? map (1+) [1, 2, 3]
 [2, 3, 4]
?
```

Filter function

- The filter function selects elements from the given list according to the given predicate.

Filter function

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

Filter function

- The filter function selects elements from the given list according to the given predicate.

Filter function

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

Filter function

```
? filter (\ x -> x `rem` 2 == 0 ) [0..9]
 [0, 2, 4, 6, 8]
?
```

Fold functions

- The fold function inserts the given operator between elements of the given list to compute the result.

Fold functions

- The fold function inserts the given operator between elements of the given list to compute the result.
- There are two fold functions;

Fold functions

- The fold function inserts the given operator between elements of the given list to compute the result.
- There are two fold functions; *foldl* from left to right, and *foldr* from right to left.

Fold functions

- The fold function inserts the given operator between elements of the given list to compute the result.
- There are two fold functions; *foldl* from left to right, and *foldr* from right to left.

Foldl function

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

Fold functions

- The fold function inserts the given operator between elements of the given list to compute the result.
- There are two fold functions; *foldl* from left to right, and *foldr* from right to left.

Foldl function

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

Foldr function

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

Functions by Folding

- Many popular functions may be defined using fold functions.

Functions by Folding

- Many popular functions may be defined using fold functions.

Functions using folds

```
sum :: [Int] -> Int
sum = foldl (+) 0
product :: [Int] -> Int
product = foldl (*) 1
concat :: [ [a] ] -> [a]
concat = foldr (++) [ ]
```

Functions by Folding

- Many popular functions may be defined using fold functions.

Functions using folds

```
sum :: [Int] -> Int
sum = foldl (+) 0
product :: [Int] -> Int
product = foldl (*) 1
concat :: [ [a] ] -> [a]
concat = foldr (++) [ ]
```

Functions using folds

```
? sum [1..5]
15
? product [1..5]
120
? concat [ [1, 2], [3], [4, 5] ]
[1, 2, 3, 4, 5]
?
```

Functions by Folding

- Many popular functions may be defined using fold functions.

Functions using folds

```
sum :: [Int] -> Int
sum = foldl (+) 0
product :: [Int] -> Int
product = foldl (*) 1
concat :: [ [a] ] -> [a]
concat = foldr (++) [ ]
```

Functions using folds

```
? sum [1..5]
 15
? product [1..5]
 120
? concat [ [1, 2], [3], [4, 5] ]
 [1, 2, 3, 4, 5]
?
```

- Which of fold functions should we use?

Functions by Folding

- Many popular functions may be defined using fold functions.

Functions using folds

```
sum :: [Int] -> Int
sum = foldl (+) 0
product :: [Int] -> Int
product = foldl (*) 1
concat :: [ [a] ] -> [a]
concat = foldr (++) [ ]
```

Functions using folds

```
? sum [1..5]
15
? product [1..5]
120
? concat [ [1, 2], [3], [4, 5] ]
[1, 2, 3, 4, 5]
?
```

- Which of fold functions should we use? Are they always behave same?

Functions by Folding

- Many popular functions may be defined using fold functions.

Functions using folds

```
sum :: [Int] -> Int
sum = foldl (+) 0
product :: [Int] -> Int
product = foldl (*) 1
concat :: [ [a] ] -> [a]
concat = foldr (++) [ ]
```

Functions using folds

```
? sum [1..5]
 15
? product [1..5]
 120
? concat [ [1, 2], [3], [4, 5] ]
 [1, 2, 3, 4, 5]
?
```

- Which of fold functions should we use? Are they always behave same? If not, Why?

Towards Calculational Programming

Calculation needs insight in Mathematics!

Calculational programming requires deep insight into the structure of data and algorithms.