GRENOBLE 1
UNIVERSITE
JOSEPH FOURIER
SCIENCES.TECHNOLOGIE.SANTÉ

INF121:
Functional Algorithmic and Programming
Lecture 2: Identifiers and functions

Academic Year 2011 - 2012

Identifiers

The notion of identifier

A fundamental concept of programming languages: associating a value to a name (an identifier)

Remark “Close” to the notion of *variable* but has fundamental differences! ☐

Identifiers

The notion of identifier

A fundamental concept of programming languages: associating a value to a name (an identifier)

Remark “Close” to the notion of *variable* but has fundamental differences! □

Some rules when defining identifiers:

- Maximal length: 256 characters
- Must begin with a non-capital letter
- No blanks
- Case-sensitive
- Should have a meaningful name

Identifiers

The notion of identifier

A fundamental concept of programming languages: associating a value to a name (an identifier)

Remark “Close” to the notion of *variable* but has fundamental differences! ☐

Some rules when defining identifiers:

- Maximal length: 256 characters
- Must begin with a non-capital letter
- No blanks
- Case-sensitive
- Should have a meaningful name

Example (Identifiers (valid and unvalid))

- speed ✓
- Speed ✗
- average speed ✗
- average_speed ✓
- s ✓ ✗
- 3m ✗
- temporary3 ✓

Identifiers: Global definition

Syntax of a *global definition*

`let identifier = expression`

↪ the value of `expression` is bound/linked to `identifier`

Type of the identifier is the type of the evaluated expression

Identifiers: Global definition

Syntax of a *global definition*

```
let identifier = expression
```

↪ the value of `expression` is bound/linked to `identifier`

Type of the identifier is the type of the evaluated expression

Definition is global: it can be used

- in other definitions
- in the rest of the program

Identifiers: Global definition

Syntax of a *global definition*

```
let identifier = expression
```

↪ the value of `expression` is bound/linked to `identifier`

Type of the identifier is the type of the evaluated expression

Definition is global: it can be used

- in other definitions
- in the rest of the program

Simultaneous definitions:

```
let ident1 = expr1
and ident2 = expr2
...
and identn = exprn
```

Identifiers: Global definition

Syntax of a *global definition*

```
let identifier = expression
```

↪ the value of `expression` is bound/linked to `identifier`

Type of the identifier is the type of the evaluated expression

Definition is global: it can be used

- in other definitions
- in the rest of the program

Simultaneous definitions:

```
let ident1 = expr1
and ident2 = expr2
...
and identn = exprn
```

Example

- `let x = 1`
- `let y = 2`
- `let i = 1`
- `let i = i+1`

DEMO: global definitions

Identifiers: Local definition

Example (Motivating example)

How to compute `e`=(2∗3∗4)∗(2∗3∗4)+(2∗3∗4)+2 ?

↪ `prod`= (2∗3∗4)

↪ `e`= `prod` ∗ `prod` + `prod` + 2

↪ `prod` is *local* to `e`

Identifiers: Local definition

Example (Motivating example)

How to compute `e`=(2∗3∗4)∗(2∗3∗4)+(2∗3∗4)+2 ?

↪ `prod`= (2∗3∗4)

↪ `e`= `prod` ∗ `prod` + `prod` + 2

↪ `prod` is *local* to `e`

Syntax of a *local definition:*

```
let identifier = expression1 in expression2
```

↪ the value of `expression1` is **permanently** bound/linked to `identifier` (only) when evaluating `expression2`

Identifiers: Local definition

Example (Motivating example)

How to compute `e`=(2∗3∗4)∗(2∗3∗4)+(2∗3∗4)+2 ?

↪ `prod`= (2∗3∗4)

↪ `e`= `prod` ∗ `prod` + `prod` + 2

↪ `prod` is *local* to `e`

Syntax of a *local definition:*

```
let identifier = expression1 in expression2
```

↪ the value of `expression1` is **permanently** bound/linked to `identifier` (only) when evaluating `expression2`

Can be nested:

```
let id1=expr1 in
 let id2=expr2 in
   ...
   let idn = exprn ... in expr
```

Works with simultaneous definitions:

```
let id1=expr1
 and id2=expr2
   ...
   and idn = exprn ... in expr
```

DEMO: local definitions

Functions

Introduction

So far, we have considered:

- expressions
- pre-defined operators

Defining our own functions: a piece of code with a specific job

Functions

Introduction

So far, we have considered:

- expressions
- pre-defined operators

Defining our own functions: a piece of code with a specific job

Motivations:

- code readability
- its job can be more elaborated than the job of pre-defined functions
- being able to execute this code from several locations

Functions

Introduction

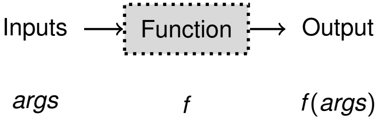
So far, we have considered:

- expressions
- pre-defined operators

Defining our own functions: a piece of code with a specific job

Motivations:

- code readability
- its job can be more elaborated than the job of pre-defined functions
- being able to execute this code from several locations



Inputs ⟶ Function ⟶ Output

args *f* $f(args)$

Functions

Introduction

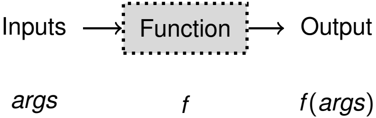
So far, we have considered:

- expressions
- pre-defined operators

Defining our own functions: a piece of code with a specific job

Motivations:

- code readability
- its job can be more elaborated than the job of pre-defined functions
- being able to execute this code from several locations



Inputs ⟶ Function ⟶ Output

args f $f(args)$

Functions in functional languages

- No side-effect (contrarily to C)
- Close to mathematical functions
- First-class objects: they are values ⇒ they have a *type*

Functions: functions with one argument

On an example

Example (Absolute value from a mathematical/abstract point of view)

$$\begin{array}{rcl} \mathbb{Z} & \rightarrow & \mathbb{N} \\ a & \mapsto & \text{if } a < 0 \text{ then } -a \text{ else } a \end{array}$$

Functions: functions with one argument

On an example

Example (Absolute value from a mathematical/abstract point of view)

$$\begin{array}{lcl} \mathbb{Z} & \rightarrow & \mathbb{N} \\ a & \mapsto & \text{if } a < 0 \text{ then } -a \text{ else } a \end{array}$$

Example (Absolute value in OCaml)

```
   fun a → if a < 0 then −a else a
or function a → if a < 0 then −a else a
or fun/function (a:int) → if a < 0 then −a else a
```

Functions: functions with one argument

On an example

Example (Absolute value from a mathematical/abstract point of view)

$$\begin{array}{lcl} \mathbb{Z} & \to & \mathbb{N} \\ a & \mapsto & \text{if } a < 0 \text{ then } -a \text{ else } a \end{array}$$

Example (Absolute value in OCaml)

`fun a → if a < 0 then −a else a`

or `function a → if a < 0 then −a else a`

or `fun/function (a:int) → if a < 0 then −a else a`

$$\begin{array}{lcccc} & \textit{keyword} & \textit{formal param.} & \textit{keyword} & \textit{function's body} \\ & \nwarrow & \uparrow & \uparrow & \uparrow \\ \text{Analysis:} & \texttt{fun} & \texttt{a} & \rightarrow & \texttt{if a < 0 then -a else a} \\ & & \downarrow & \downarrow & \downarrow \\ & \text{type:} & \texttt{int} & \texttt{->} & \texttt{int} \end{array}$$

Remark This function is anonymous, i.e., it does not have a name ☐

DEMO: anonymous functions

Functions

How to define them

Naming a function allows to reuse it

Example (Defining the function absolute value)

```
    let abs = fun (a:int) → if a < 0 then −a else a
or  let abs a = if a < 0 then −a else a
or  let abs (a:int) = if a < 0 then −a else a
or  let abs (a:int):int = if a < 0 then −a else a
```

DEMO: defining functions

Functions

How to define them

Naming a function allows to reuse it

Example (Defining the function absolute value)

```
    let abs = fun (a:int) → if a < 0 then −a else a
or  let abs a = if a < 0 then −a else a
or  let abs (a:int) = if a < 0 then −a else a
or  let abs (a:int):int = if a < 0 then −a else a
```

DEMO: defining functions

Exercise

Define the function `square`: `int` → `int`

Functions

How to use them

As in mathematics, the result of applying f to x is $f(x)$

Example

- `abs`(2)
- `abs`$(2 - 3)$
- `abs` 2 (parenthesis can be omitted)

Functions

How to use them

As in mathematics, the result of applying f to x is $f(x)$

Example

- `abs`(2)
- `abs`$(2 - 3)$
- `abs` 2 (parenthesis can be omitted)

Application of a function

`expr1 expr2`

Typing: if and `expr1` has type `t1->t2`, `expr2` has type `t1` } then `expr1 expr2` has type `t2`

Functions: Generalization to functions with several arguments

Example (Surface area of a rectangle)

- Needs 2 parameters: length and width (floats)
- definition:

```
let surface (x:float) (y:float):float = x *. y
let surface (length:float) (width:float):float = length *.
width
```

- usage: `surface` 2.3 1.2

Functions: Generalization to functions with several arguments

Example (Surface area of a rectangle)

- Needs 2 parameters: length and width (floats)
- definition:
  `let surface (x:float) (y:float):float = x *. y`
  `let surface (length:float) (width:float):float = length *. width`
- usage: `surface` 2.3 1.2

Definition of a Function with *n* parameters

`let fct_name (p1:t1) (p2:t2) ... (pn:tn) : t = expr`

- `p1`, ..., `pn` are *formal* parameters
- Type of `fct_name` is `t1 -> t2 -> ...  -> tn -> t`

Functions: Generalization to functions with several arguments

Example (Surface area of a rectangle)

- Needs 2 parameters: length and width (floats)
- definition:
  `let surface (x:float) (y:float):float = x *. y`
  `let surface (length:float) (width:float):float = length *. width`
- usage: `surface 2.3 1.2`

Definition of a Function with *n* parameters

`let fct_name (p1:t1) (p2:t2) ... (pn:tn) : t = expr`

- `p1, ..., pn` are *formal* parameters
- Type of `fct_name` is `t1 -> t2 -> ...  -> tn -> t`

Using a Function with *n* parameters

`fct_name e1 e2 ... en`

- `e1,...,en` are *effective* parameters
- Type of `fct_name e1 e2 ... en` is `t`
  if `ti` is the type of `ei` and `fct_name` is of type `t1 -> t2 -> ... -> tn -> t`

Functions: SPECIFICATION and IMPLEMENTATION

In this module (and in your future), it is very important to distinguish two concepts/stages about defining functions (and programs in general)

Functions: SPECIFICATION and IMPLEMENTATION

In this module (and in your future), it is very important to distinguish two concepts/stages about defining functions (and programs in general)

Specification:

A description of what it is expected to do/ the job

- at an abstract level
- should be precise
- close to maths description in fun programming
- illustrate the function with some *interesting* examples

A contract:

Inputs ⟶ Function ⟶ Output

Consists of:

- description
- signature
- examples

Functions: SPECIFICATION and IMPLEMENTATION

In this module (and in your future), it is very important to distinguish two concepts/stages about defining functions (and programs in general)

Specification:

A description of what it is expected to do/ the job

- at an abstract level
- should be precise
- close to maths description in fun programming
- illustrate the function with some *interesting* examples

Implementation:

The description of how it is done

- the OCaml code

A contract:

Consists of:

- description
- signature
- examples

Functions: SPECIFICATION and IMPLEMENTATION

In this module (and in your future), it is very important to distinguish two concepts/stages about defining functions (and programs in general)

Specification:

A description of what it is expected to do/ the job

- at an abstract level
- should be precise
- close to maths description in fun programming
- illustrate the function with some *interesting* examples

A contract:

Inputs ⟶ Function ⟶ Output

Consists of:

- description
- signature
- examples

Implementation:

The description of how it is done

- the OCaml code

Inputs ⟶ Function ⟶ Output

Defining a function: Specification AND THEN Implementation

Functions: SPECIFICATION and IMPLEMENTATION

In this module (and in your future), it is very important to distinguish two concepts/stages about defining functions (and programs in general)

Specification:

A description of what it is expected to do/ the job

- at an abstract level
- should be precise
- close to maths description in fun programming
- illustrate the function with some *interesting* examples

A contract:

Consists of:

- description
- signature
- examples

Implementation:

The description of how it is done

- the OCaml code

Defining a function: Specification AND THEN Implementation

Has many advantages (how big software is developed):

- re-usability
- thinking before acting
- you will save a lot of time
- you will have a better grade

Defining functions: some examples

Example (Defining the function absolute value)

- Specification:
  - The function absolute value *abs* takes an integer n as a parameter and returns n if this integer is positive or $-n$ if this integer is negative. The function absolute value always returns a positive integer.
  - Signature: $\mathbb{Z} \rightarrow \mathbb{N}$
  - Example: $abs(1) = 1$, $abs(0) = 0$, $abs(-2) = 2$
- Implementation: `let abs (a:int) = if a < 0 then −a else a`

Defining functions: some examples

Example (Defining the function absolute value)

- Specification:
  - The function absolute value *abs* takes an integer n as a parameter and returns n if this integer is positive or $-n$ if this integer is negative. The function absolute value always returns a positive integer.
  - Signature: $\mathbb{Z} \to \mathbb{N}$
  - Example: $abs(1) = 1$, $abs(0) = 0$, $abs(-2) = 2$
- Implementation: `let abs (a:int) = if a < 0 then −a else a`

Example (Defining the function square)

- Specification:
  - The function square *sq* takes an integer n as a parameter and returns $n * n$.
  - Signature: $\mathbb{Z} \to \mathbb{N}$
  - Example: $sq(1) = 1$, $sq(0) = 0$, $sq(3) = 9$, $sq(-4) = 16$
- Implementation: `let sq (n:int) = n∗n`

Some exercises

A piece of algorithmic

Exercise

Define the function `my_max` returning the maximum of two integers

Some exercises

A piece of algorithmic

Exercise

Define the function `my_max` returning the maximum of two integers

Exercise

Define functions:

- `square`: `int` → `int`
- `sum_square`: `int` → `int` → `int`

s.t. `sum_square` computes the sum of the squares of two numbers

Some exercises

A piece of algorithmic

Exercise

Define the function `my_max` returning the maximum of two integers

Exercise

Define functions:

- `square`: `int` → `int`
- `sum_square`: `int` → `int` → `int`

s.t. `sum_square` computes the sum of the squares of two numbers

Problem: Olympic mean

Computing the mean of 4 grades (or values), by suppressing the highest and lowest one

1. Propose a type for the function `mean`
2. Propose an algorithm, by supposing that you have two functions `min4` and `max4`, which compute respectively the minimum and the maximum of 4 integers
3. Define functions `min4` et `max4`, using `min` and `max`