Declarations of global variables

A more detailed syntax for declarations

```
Dec ::= 'let' Def ('and' Def)*
Def ::= Pat '=' Exp | ID Pat+ '=' Exp
Pat ::= ID | '_' |  '(' Pat? ')' | Pat (',' Pat)+
```

Declarations of global variables

A simple example

```
# let x=2;;
val x : int = 2
# let y=x+40;;
val y : int = 42
# x+y;;
- : int = 44
```

Remarks

- variables are global because they are declared at the top level
- local variables can be declared as well at inner levels (see later)
- the content of variables cannot be changed, variables are constant
- there is no variable assignment

Declarations of global variables

More elaborate examples

```
# let x=2 and y=42;;
val x : int = 2
val y : int = 42
# x+y;;
- : int = 44
# let x=3;; (* previous declaration is shadowed *)
val x : int = 3
# x+y;;
- : int = 45
# let pair = 4,2;;
val pair : int * int = (4,2)
# let a,b = pair;;
val a : int = 4
val b : int = 2
```

Remark

a top-level declaration with the same name shadows the previous declaration

Declarations of global variables

Examples of global function declarations

```
# let x = 3;;
# let inc x = x+1 and add (x,y) = x+y and add2 x y = x+y;;
```

Remark

the declaration of parameter `x` shadows in the body of the functions the top-level declaration of `x`

A useful syntactic abbreviation

`let inc x = x+1` abbreviates `let inc = fun x->x+1`
`let add (x,y)= x+y` abbreviates `let add = fun (x,y)->x+y`
`let add2 x y = x+y` abbreviates `let add2 = fun x->fun y->x+y`

More in general:

let *id* $pat_1\ pat_2 \ldots pat_n$ = *exp* abbreviates

let *id* = **fun** pat_1 -> **fun** pat_2 -> ... **fun** pat_n -> *exp*

Boolean values

Syntax

```
Exp ::= BOOL | 'not' Exp | Exp '&&' Exp | Exp '||' Exp
Type ::= 'bool'
```

`BOOL` defined by the regular expression **`false|true`**

Standard syntactic rules

- left syntactic associativity for `&&` and `||`
- **`not`** higher precedence than `&&`
- `&&` higher precedence than `||`

Boolean values

Static semantics

- **`false`** and **`true`** are type correct and have type `bool`
- **`not`** e is type correct and has type `bool` if and only if e is type correct and has type `bool`
- e_1 `&&` e_2 and e_1 `||` e_2 are type correct and have type `bool` if and only if e_1 and e_2 are type correct and have type `bool`

Boolean values

Dynamic semantics

- operands of `&&` and `||` evaluated left-to-right with short circuit
- short circuit means that not always the second operand is evaluated
- if e_1 evaluates to `false` then e_1`&&`e_2 evaluates to `false` and e_2 is not evaluated
- if e_1 evaluates to `true` then e_1`&&`e_2 evaluates to the value of e_2
- if e_1 evaluates to `true` then e_1`||`e_2 evaluates to `true` and e_2 is not evaluated
- if e_1 evaluates to `false` then e_1`||`e_2 evaluates to the value of e_2

Example

```
# 1<0 && 0/0>0;;
- : bool = false
# 0/0>0 && 1<0;;
Exception: Division_by_zero.
```

Boolean values

Conditional expression

```
Exp ::= 'if' Exp 'then' Exp 'else' Exp
```

Conditional expression has precedence lower than all other operators

Static semantics

if e **then** e_1 **else** e_2 is type correct and has type t if and only if

- e is type correct and has type `bool`
- e_1 and e_2 are type correct and have the same type t

Dynamic semantics

- if e evaluates to `true`, then **if** e **then** e_1 **else** e_2 evaluates to the value of e_1; hence, e_2 is not evaluated
- if e evaluates to `false`, then **if** e **then** e_1 **else** e_2 evaluates to the value of e_2; hence, e_1 is not evaluated

Boolean values

Statements versus expressions

- statement:
  - its execution is expected to change the status of the program (memory, I/O)
  - no associated value is expected
- expression:
  - an associated value is expected to be computed

Purely functional programming

- the core of OCaml is purely functional
- there are no statements
  - **if** e **then** e_1 **else** e_2 is an expression
  - the **else** branch cannot be omitted, otherwise the value of the expression would be undefined when e is false

Recursive declarations

Syntax for recursive declarations

```
Dec ::= 'let' 'rec'? Def ('and' Def)*
Def ::= Pat '=' Exp | ID Pat+ '=' Exp
Pat ::= ID | '_' |  '(' Pat? ')' | Pat (',' Pat)+
```

Remark

- the optional `'rec'` keyword means that the declaration is allowed to be recursive
- the use of `'rec'`, `'and'` keywords supports mutually recursive declarations
- recursive declarations allowed only for function types and other particular types
- for simplicity we consider only recursive declarations of functions

Recursive declarations of functions

Examples

```
(* addition of square numbers *)
let sumsquare n = (*sumsquare cannot be used on the right-hand side*)
    if n<0 then 0 else n*n+sumsquare(n-1);;
Error:  Unbound value sumsquare

let rec sumsquare n = (*sumsquare can be used on the right-hand side*)
    if n<0 then 0 else n*n+sumsquare(n-1);;
```

Curried functions and generic programming

Example 1: addition of square numbers

```
let rec sumsquare n =
    if n<0 then 0 else n*n+sumsquare(n-1);;
```

Example 2: addition of cube numbers

```
let rec sumcube n =
    if n<0 then 0 else n*n*n+sumcube(n-1);;
```

Remarks

- the two declarations above are almost identical!
- can we improve code reuse and maintenance?

Solution: use a curried function with an argument of type function

Curried functions and generic programming

Solution

```
(* computes f 0 + f 1 + ... + f n *)
let rec gen_sum f n = (* (int -> int) -> int -> int *)
   if n<0 then 0 else f n+gen_sum f (n-1);;

let sumsquare = gen_sum (fun x->x*x);; (* int -> int *)
let sumcube = gen_sum (fun x->x*x*x);; (* int -> int *)
```

Remarks

`gen_sum` can be specialized because

- it is curried
- its argument is the function `f` rather than the number `n`

Declarations of local variables

Syntax for declarations of local variables

```
Dec ::= 'let' 'rec'? Def ('and' Def)* 'in' Exp
Def ::= Pat '=' Exp | ID Pat+ '=' Exp
Pat ::= ID | '_' |  '(' Pat? ')' | Pat (',' Pat)+
```

Example

```
# let inc x=x+1 and
      v=41 in inc v;; (* inc and v can only be used here *)
- : int = 42
# let x=1 in let x=x*2 in x*x (* nested declarations *)
- : int = 4
```

Remark

A nested declaration with the same name shadows outer declarations

Global variables in function bodies

Example

```
# let v=40;;
val v : int = 40
# let mul x = x*v;; (* v refers to the declaration above *)
val mul : int -> int = <fun>
# mul 3;;
- : int = 120
# let v=2;; (* previous declaration of v shadowed *)
val v : int = 2
# mul 3;; (* mul still refers to the shadowed variable v *)
- : int = 120
```

We say that in OCaml the scope of declarations is static

Curried functions and generic programming (revisited)

Observation

```
(* computes f 0 + f 1 + ... + f n *)
let rec gen_sum f n = (* (int -> int) -> int -> int *)
   if n<0 then 0 else f n+gen_sum f (n-1);;
```

function `f` must be passed as argument of the recursive application

A better solution with a nested declaration

```
let gen_sum f = (* (int -> int) -> int -> int *)
    let rec aux n = if n<0 then 0 else f n+aux (n-1) (* int -> int *)
    in aux;;
```

Remark

we do not have to pass argument `f` to the recursive function `aux`

Lists

Lists are built-in values in OCaml

Some examples of built-in composite list types:

- `int list`
- `(int * int)list`
- `(int -> int)list`
- `int list list`

List constructors

- Syntax: `Exp ::= '[' ']'| Exp '::'Exp`
- `[]` is the empty list constructor
- `::` is the non-empty list constructor: h`::`t is the list with head h and tail t

Lists

Examples

```
# let l=1::2::3::[];;
val l : int list = [1; 2; 3]
# let l2=0::l;;
val l2 : int list = [0; 1; 2; 3]
# let pl = (1,2)::(3,4)::[];;
val pl : (int * int) list = [(1, 2); (3, 4)]
# let fl=(fun x->x+1)::(fun x->x*2)::[];;
val fl : (int -> int) list = [<fun>; <fun>]
# let ll=(1::[])::(2::3::[])::[];;
val ll : int list list = [[1]; [2; 3]]
```

Syntactic rules for lists

The usual properties of constructors hold

- $[\,] \neq h::t \qquad h \neq h::t \qquad t \neq h::t$
- $h_1::t_1 = l$ if and only if $l = h_2::t_2$, $h_1 = h_2$ and $t_1 = t_2$

Non-empty list constructor `::`

- right syntactic associativity holds
  $h_1::h_2::t$ is equivalent to $h_1::(h_2::t)$
  this is the only sensible choice (see later on)
- `::` has lower precedence than unary and binary infix operators
- `::` has higher precedence than
  - the tuple constructor
  - anonymous function expression (**`fun ... -> ...`**)
  - conditional expression (**`if ... then ... else ...`**)

Syntactic rules for lists

A useful shorthand notation

$[e_1; e_2; \ldots; e_n]$ is equivalent to $e_1 :: e_2 :: \ldots :: e_n :: []$

Examples

```
# 1::[];;
- : int list = [1]
# 1::2::3::[];;
- : int list = [1; 2; 3]
# (1,true)::[];;
- : (int * bool) list = [(1, true)]
# (1,true::[]);;
- : int * bool list = (1, [true])
```

Warning

Use parentheses if you mix lists and tuples together!