Pattern matching

New productions for `Pat`

```
Pat ::= '[' ']' | Pat '::' Pat | '[' Pat (';' Pat)* ']'
```

Remark

- all variables in a pattern **must be distinct**
  - this makes pattern matching more efficient
- patterns are built with constructors, but **not** with operators
  - `x::y` is a valid pattern, `x@y` or `x+y` **is not**
  - constructors guarantee unique decomposition of values

Pattern matching

What is pattern matching?

Examples:

```
let add (x,y) = x+y;;
add (3,5);; (* does (3,5) match with pattern (x,y)? *)
```

- `(3,5)` matches with pattern `(x,y)` if and only if `x=3` and `y=5`
- if we substitute `x` and `y` in `(x,y)` with `3` and `5`, respectively, then we obtain the value `(3,5)`

```
let hd (h::t) = h;; (* returns the head of the list *)
hd [3;5];; (* does [3;5] match with pattern h::t? *)
```

- `[3;5]` matches with pattern `(h::t)` if and only if `h=3` and `t=[5]`
- if we substitute `h` and `t` in `(h::t)` with `3` and `[5]`, respectively, then we obtain the value `(3::[5])=[3;5]`

Pattern matching

Do we always need to use a single pattern to match all values?

```
let hd (h::t) = h;; (* returns the head of the list *)

hd [];; // does [] match with pattern h::t?
```

- `[]` does not match with `(h::t)` for any value associated with `h` and `t`
- indeed `[]` $\neq$ `(h::t)` for all possible values associated with `h` and `t`
- this is correct, because the head of a list is undefined for the empty list

Functions that cannot be directly defined with a single pattern

- the length of a list
- the sum of all the elements of a list
- the list with the first two elements swapped

Pattern matching

An expression to match values with multiple patterns

```
Exp ::= 'match' Exp 'with' Pat '->' Exp ('|' Pat '->' Exp)*
```

Examples

```
(* defined with two patterns *)
let rec length l = match l with
    [] -> 0
  | hd::tl -> 1+length tl;;

let rec sum l = match l with
    [] -> 0
  | hd::tl -> hd+sum tl;; (* hd and tl are local variables *)
(* defined with more than two patterns *)
let swap l = match l with
    [] -> []
  | [x] -> [x] (* x is a local variable *)
  | x::y::l -> y::x::l;; (* x, y and l are local variables *)
```

Pattern matching

match e **with** p_1 `->` e_1 `| ... |` p_n `->` e_n

Static semantics

- the expression e and all patterns $p_1 \ldots p_n$ must have the same type
- all expressions $e_1 \ldots e_n$ must have the same type

Dynamic semantics

- e is evaluated
- all patterns $p_1 \ldots p_n$ tried from left to right, top to bottom
- at the first match with p_i, the expression e_i is evaluated, with variables defined by the match with p_i
- if there is no match, then `Match_failure` is raised

Pattern matching

Static semantics: further checks

A warning is reported if:

- patterns are not exhaustive, that is, some case is missing
- a pattern is unused

Example

```
# let head (hd::tl) = hd;;
           ^^^^^^^^^^^^^
Warning 8: this pattern-matching is not exhaustive.
Here is an example of a case that is not matched:
[]

# let rec length l = match l with
    hd::tl -> 1+length tl
  | [x] -> 1
    ^^^
  | [] -> 0;;

Warning 11: this match case is unused.
```

Pattern matching

Unique decomposition

Constructors ensure that if there is a match with *p*, then there exists a unique substitution for the variables in *p*

Counter-example

```
# fun ls -> match ls with l1@l2 -> l1;; (* @ is not a constructor! *)
                             ^
Error: Syntax error
```

If `ls` is `[1;2;3]` what would be the value of `l1`???

`[]`???

`[1]`???

`[1;2]`???

`[1;2;3]`???

Pattern matching

Constructors for primitive types

All *literals* (=tokens that represent values) are constant constructors

Example of pattern matching with primitive types

```
let mynot b = match b with false -> true | true -> false;;

let iszero i = match i with 0 -> true | any -> false;;
(* a variable is needed for the second pattern *)
```

Remark

Pattern matching with primitive types is seldom used

Pattern matching

Shorthand notation

- the *wildcard* _ is the pattern which matches all values when no variable is needed
- **function** p_1 `->` e_1 `|` ... `|` p_n `->` e_n is a shorthand for
  fun *var* `->` **match** *var* **with** p_1 `->` e_1 `|` ... `|` p_n `->` e_n
- p **as** *id*: a pattern (or sub-pattern) p can be associated with an *id* to refer to the matched value more directly

Pattern matching

Examples

```
let mynot = function false -> true | _ -> false;;

let iszero = function 0 -> true | _ -> false;;

let rec length = function _::tl -> 1+length tl | _ -> 0;;

let rec sum = function hd::tl -> hd+sum tl | _ -> 0;;

let swap = function x::y::l -> y::x::l | other -> other;;

let ord_swap = function
    x::y::tl as l -> if x>y then y::x::tl else l
  | other -> other;;
```