# Basic types

| value | type |
|---|---|
| () | unit |
| true, false | bool |
| 1,2, $\cdots$ | int |
| 1.0,1.,2.5, $\cdots$ | float |
| ’a’,\verb’b’+ | char |
| "hi" | string |

# Expressions

| expression | result | type |
|---|---|---|
| 2 * 3 | 6 | int |
| 2.0 *. 3.0 | 6. | float |
| fun n -> n*2 | <fun> | int -> int |
| (fun n -> n*2) 2 | 4 | int |
| ( * ) | <fun> | int -> int -> int |
| ( * ) 2 | <fun> | int -> int |
| ( * ) 2 3 | 6 | int |

# Tuples

| value | type |
|---|---|
| () | unit |
| (1) $\equiv$ 1 | int |
| (1,1) | int * int |
| (1,true,’a’) | int * bool * char |

# Algebraic Data Types

- Definition

```
type 'a mylist = Nil | Cons of 'a * 'a mylist
type 'a tree = Leaf | Node of 'a * 'a tree * 'a tree

type ('a, 'b) union = A of 'a | B of 'b
```

- Formally: $\texttt{type}\ ('a_1, \cdots, 'a_n)\ \texttt{name}\ =\ tag_1 \mid \cdots \mid tag_N$, where
  - name must start with a lower case letter.
  - $tag_i$ is
    - An atomic constructor: `Name`
    - A non-atomic constructor with argument:`Name of` *type*, where *type* is a type expression in which $'a_1, \cdots, 'a_n$ may occur.

  where name must start with an upper case letter and

- Values

```
Nil,  Cons(1,Nil), Leaf, Node(1,Leaf,Leaf), ...
```

# Expressions (without regarding types)

Let $e_1$ and $e_2$ be expressions.

- If $c$ is a constant then $c$ is an expression.
- If *name* is defined then it is an expression.
- The application $e_1\ e_2$ is an expression.
- The sequential composition $e_1\ ;\ e_2$ is an expression.
- If *Name* is an atomic constructor then *Name* is an expression
- If *Name* is a non-atomic then *Name* $e_1$ is an expression.
  Note that in this case *Name* itself is not an expression.
- The abstraction `fun` *name* `->` $e_1$ is an expression.
- If $\diamond$ is an operator then ( $\diamond$ ) is an expression.
- If $\diamond$ is a unary then $\diamond e_1$ is an expression.
- If $\diamond$ is a binary then $e_1 \diamond e2$ is an expression.
- If $e_1, \cdots, e_n$ are expressions then $(e_1, \cdots, e_n)$ is an expression.

# Pattern matching

- A pattern is

  $$P ::= \_ \mid \mathit{ident} \mid \mathit{Atom} \mid \mathit{Cons}\ P \mid (P, \cdots, P) \mid \text{constant}$$

  where an identifier may not occur twice or more.
  E.g. `Cons(a,x)` is a pattern and `Cons(x,x)` is not.
  See reference manual for other possibilities.

- If $p_1, \cdots, p_n$ are patterns and $e$, $c_1, \cdots, c_n$, $e_1, \cdots e_n$ are expressions then

  $$\begin{array}{l} \texttt{match}\ e\ \texttt{with} \\ \quad \mid\ p_1\ \texttt{when}\ c_1\ \longrightarrow\ e_1 \\ \qquad\qquad \vdots \\ \quad \mid\ p_n\ \texttt{when}\ c_n\ \longrightarrow\ e_n \end{array}$$

  is an expression.

# Definitions

- If $e_1$ and $e_2$ are expressions then

  $$\texttt{let}\ \mathit{name} = e_1\ \texttt{in}\ e_2$$

  is an expression.
- If $e$ is an expression and $e_1, \cdots, e_n$ are value expressions (functions or constructors or constants) then

  $$\texttt{let rec}\ \mathit{name}_1 = e_1\ \texttt{and} \cdots \texttt{and}\ \mathit{name}_n = e_n\ \texttt{in}\ e$$

  is an expression.

# Definitions

- At top level, we write

  $$\texttt{let}\ \mathit{name} = e_1\texttt{;;}$$
  $$e_2\texttt{;;}$$

  and

  $$\texttt{let rec}\ \mathit{name}_1 = e_1\ \texttt{and} \cdots \texttt{and}\ \mathit{name}_n = e_n\texttt{;;}$$
  $$e\texttt{;;}$$

  to enable separate compilation and reuse of definitions

# Abbreviations

| short | long |
|---|---|
| $\mid p \longrightarrow e$ | $\mid p\ \texttt{when true} \longrightarrow e$ |
| $f\ x_1 \cdots x_n = e$ | $f = \texttt{fun}\ x_1 \longrightarrow \cdots \texttt{fun}\ x_n \longrightarrow e$ |
| $\texttt{fun}\ p \longrightarrow e$ | $\texttt{fun}\ x \longrightarrow \texttt{match}\ x\ \texttt{with} \mid p \longrightarrow e$ |
| $\texttt{let}\ p = e_1\ \texttt{in}\ e_2$ | $\texttt{match}\ e_1\ \texttt{with} \mid p \longrightarrow e_2$ |
| $\texttt{if}\ c\ \texttt{then}\ e_1\ \texttt{else}\ e_2$ | $\texttt{match}\ c\ \texttt{with} \mid \texttt{true} \longrightarrow e_1 \mid \texttt{false} \longrightarrow e_2$ |