Introduction to ML

Expressions

- Basic notion in a functional language is an *expression*
- Expression: Function applied to a value
- Functions and values have *types*
- ML uses "eager evaluation": First, evaluate the arguments of a function, then the function itself

Types

- Basic types

  `unit, bool, real, char, string, int`

- Tuples: constructed from other types
- Basic types
  - `unit`: Single value `()`, used for expressions that do not return a value
  - `bool`: Values `true` and `false`
  - `int`: Integers, positive and negative. Note that `~3` is -3. This is actually an operator, that negates the integer. Other operators on integers are
  - `*`: Multiplication
  - `+`: Addition
  - `-`: Subtraction (binary)

Examples

```
poly
1;
```

What do you expect as result?

Example

```
val it = 1: int

1=3;
```

Example

```
val it = false: bool

1.0;
```

Example

```
val it = false: bool

1.0=2.0;
```

Example

```
poly: : error: Type error in function application.
   Function: = : ''a * ''a -> bool
   Argument: (1.0, 2.0) : real * real
   Reason: Can't unify ''a to real (Requires equality type)
Found near 1.0 = 2.0
Static Errors
```

You don't have to understand all this now! We'll return to it later

More examples

```
> ();
val it = (): unit

> 2=3;
val it = false: bool

> ~3 * ~ 6;
val it = 18: int

> 1/2;
poly: : error: Type error in function application.
   Function: / : real * real -> real
   Argument: (1, 2) : int * int
   Reason:
      Can't unify int (*In Basis*) with real (*In Basis*)
         (Different type constructors)
Found near 1 / 2
Static Errors
```

Never mind the details for now. It's saying that division is only allowed for reals

Types

- `real`. Either 3.14 or `314e~2`
- Special values
  - `NaN`. Not a Number
  - `inf`. Infinity
- Reals can be compared with < etc, resulting in boolean values
- `char`: Characters. Written `#"a"`
- `string`: Sequences of characters
  - Concatenation: `"Hello, " ^ "world"` is equal to `Hello, world`

Examples

```
> 1.0/0.0;
val it = inf: real

> 5 < 6;
val it = true: bool
> 5.0 < 6.0;
val it = true: bool
> 5 < 6.0;
poly: : error: Type error in function application.
   Function: < : int * int -> bool
   Argument: (5, 6.0) : int * real
   Reason:
      Can't unify int (*In Basis*) with real (*In Basis*)
         (Different type constructors)
Found near 5 < 6.0
Static Errors
```

Types

- No automatic conversion of types
  - `5+7` and `5.0+7.0` are correct, resulting in `int` and `real`
  - `5+7.0` is wrong
  - Conversion between types
    - `ord`: character to integer
    - `chr` reverse direction

Conditionals

- Syntax

  ```
  if <p> then <exp1> else <exp2>;
  ```

- This is an *expression*. Therefore
  - `else` is required
  - Both parts must have values
- Example. Maximum

  ```
  if a>b then a else b;
  ```

Example

```
> if 5<6 then 5 else 6;
val it = 5: int
> if 5<6 then 5;
poly: : error: else expected but ; was found
poly: : error: Expression expected but ; was found
Static Errors
```

Variable

- Environment: Set of pairs of identifiers and value
- Environment is modified by assignment statements

```
val <name> = <value>;
val <name>:<type> = <value>;
val <name> = <expression>;
```

- Example

```
val v = 10.0/2.0;
```

(/ is integer division)

Examples

```
> val a = 5;
val a = 5: int
> val a = 5.0;
val a = 5.0: real

> val v = 10.0/2.0;
val v = 5.0: real
```

Variables

- Variables cannot be modified!
- `val` creates an association between a name and a variable
- The statements

  ```
  val pi = 3.14;
  val pi = 3.1415
  ```

  create two variables with name `pi`, where the second hides the first
- `val` statements change the environment, not the variables themselves

Functions

- In ML, just another type of value
- Represented by *parametrized expressions*
- Calculates a value based on parameters
  - No collateral effects
- Syntax `fn` (corresponds with λ in the λ-calculus, that we study later)

```
fn <param> => <expression>;
```

- Example

```
fn n => n+1;
```

Examples

```
> fn n => n+1;
val it = fn: int -> int
```

Not very useful without a name. But we can still apply it

```
> ( fn n => n+1) 5;
val it = 6: int
```

Functions

- Applying a function to a parameter

```
(fn n => n+1 ) 5;
```

- Value 5 is associated to formal parameter `n`, and then the function is evaluated
- Functions can be associated to names, just like values

```
val increment = fn n => n+1;
```

- Note the type of `increment`
- We can write

```
increment 5;
```

Question

- What is the difference between

  ```
  ( fn x => x+1) (( fn x=> x+1) 2 );
  ```

  and

  ```
  increment (increment 2);
  ```

Examples

```
> ( fn x => x+1) (( fn x=> x+1) 2 );
val it = 4: int

> val increment = fn n => n+1;
val increment = fn: int -> int

> increment (increment 2);
val it = 4: int
```

Cases

- Syntax

```
fn x => case x of
   <pattern_1> => <expression_1>
 | <pattern_2> => <expression_2>
...
 | <pattern_n> => <expression_n>
```

- Remember that this is an *expression*, so every `x` must satisfy one case
- Example

```
val day = fn n => case n of
   1 => "Monday"
|  2 => "Tuesday"
|  _ => "Other";
```

Examples

```
val day = fn n => case n of
     1 => "Monday"
  |  2 => "Tuesday"
  |  _ => "Other";

> val day = fn: int -> string

> day 1;
val it = "Monday": string
> day 4;
val it = "Other": string
```

Example

What if we omit the default case?

```
>val day = fn n => case n of
     1 => "Monday"
  |  2 => "Tuesday";

poly: : warning: Matches are not exhaustive.
Found near case n of 1 => "Monday" | 2 => "Tuesday"
val day = fn: int -> string

> day 1;
val it = "Monday": string
> day 4;
Exception- Match raised
```

Cases

- The pattern does not have to be a constant value, as in most programming languages
- ML uses a mechanism of *pattern matching*
- Example

```
val f = fn a => case a of
   0 => 1000.0
|  x => 1.0/real x;
```

- Another example of pattern matching

```
val sum = fn (a,b) => a+b;
```

Example

```
> val f = fn a => case a of
|  x => 1.0/real x;
val f = fn: int -> real
> f 0;
val it = 1000.0: real
> f 1;
val it = 1.0: real
> f 11;
val it = 0.09090909091: real

> val sum = fn (a,b) => a+b;
val sum = fn: int * int -> int
> sum (5,6);
val it = 11: int
> sum (5.1,6.2);
poly: : error: Type error in function application.
   Function: sum : int * int -> int
   Argument: (5.1, 6.2) : real * real
   Reason:
      Can't unify int (*In Basis*) with real (*In Basis*)
         (Different type constructors)
Found near sum (5.1, 6.2)
Static Errors
```

Pattern matching

- Case statements can be replaced by pattern matching
- Example

```
val day = fn   1 => "Monday"
             |  2 => "Tuesday"
             |  _ => "Other";
```

- Another example of pattern matching

```
val (x,y) = (4,5);
```

  Assigns two variables with a single statement

Example

```
> val day = fn   1 => "Monday"
              |  2 => "Tuesday"
              |  _ => "Other";
val day = fn: int -> string
> val (x,y) = (4,5);
val x = 4: int
val y = 5: int
>
```

Pattern matching

- Pattern
  - A constant value, that matches itself
  - A variable pattern `<var>:<type>`, of type `<type>`, that matches any value of type `<type>`, after having created a binding between `<var>` and its value
  - A tuple pattern `<pattern1>,<pattern2>,...,<patternn>` of type `<type1>*<type2>*.. * <typen>`. Each pattern must match the corresponding component of the tuple
  - The wildcard pattern `_`, that matches any value

  Note that the value `()` is an example of a tuple pattern, the empty tuple

Example

```
> val x:int = 1;
val x = 1: int

> val x:real = 1;
poly: : error: Pattern and expression have incompatible types.
   Pattern: x : real : real
   Expression: 1 : int
   Reason:
      Can't unify int (*In Basis*) with real (*In Basis*)
         (Different type constructors)
Found near val x : real = 1
Static Errors

> val x:real = 1.0;
val x = 1.0: real

> val x = "ab": string
val x:char = 'a;

> val x:bool = true;
val x = true: bool
```

Exercise

Write a case statement to simulate the if-then-else clause

Solution

```
case booleanExpr of
    true => expr1
  | false => expr2;
```

For example

```
case 1<2 of
   true => 1;
  | false => 2;
```

Exercise

Write a function to return 1 if the parameter is 1 and "`anything else`" otherwise

Solution

```
val f = fn
   1 => "one"
 | _ => "anything else";

> f 1;
val it = "one": string
> f 2;
val it = "anything else": string
```

Why is this wrong?

```
val f = fn
  _ => "anything else";
|
   1 => "one"
```

Recursion

- Functional programming languages use recursion where an imperative language would use iteration
- The problem is that we cannot normally use a name before it has been defined

```
val fact = fn n => if n = 0 then 1 else n * fact (n - 1);
```

- The keyword `rec` creates an association even before the function has been defined

```
val rec fact = fn n => if n = 0 then 1 else n * fact (n - 1);
```

Examples

```
> val fact = fn n => if n = 0 then 1 else n * fact (n - 1);
poly: : error: Value or constructor (fact) has not been declared
Found near if n = 0 then 1 else n * fact (n - 1)
Static Errors
> val rec fact = fn n => if n = 0 then 1 else n * fact (n - 1);
val fact = fn: int -> int
> fact 0;
val it = 1: int
> fact 3;
val it = 6: int
> fact 100;
Exception- Overflow raised
```

Recursive functions

- The keyword `fun` can be used instead of `val rec`

```
fun fact n = if n = 0 then 1 else n * fact (n - 1);
```

- What is the difference between

```
val f = fn n => if n = 0 then 1 else n * f(n - 1);
```

and

```
val rec f = fn n => if n = 0 then 1 else n * f(n - 1);
```

Example

```
> val f = fn n => if n = 0 then 1 else n * f(n - 1);
poly: : error: Type error in function application.
   Function: * : int * int -> int
   Argument: (n, f (n - 1)) : int * string
   Reason:
      Can't unify int (*In Basis*) with string (*In Basis*)
         (Different type constructors)
Found near if n = 0 then 1 else n * f (n - 1)
Static Errors
> val rec f = fn n => if n = 0 then 1 else n * f(n - 1);
val f = fn: int -> int
>
```

Exercise

Define a function which computes the product of all integers between m and n (with $n \geq m$ inclusive. Use this function to define the function $C(n, k)$, the number of combinations of n elements taken k by k, which is defined by

$$C(n, k) = n!/(k! * (n - k)!)$$

Equivalently,

$$C(n, k) = (n - k + 1) \cdot (n - k + 2) \cdots n/1 \cdot 2 \cdots k$$

Use `div` for integer division

Solution

```
>    fun prod(m,n) = if n <= m then m else n * prod(m,n-1);
val prod = fn: int * int -> int
>    fun C(n,k) = prod(n-k+1,n) div prod(1,k);
val C = fn: int * int -> int

> C(7,5);
val it = 21: int
>
```