λ

Expressions, commands and data types

Programmazione Funzionale

2024/2025

Università di Trento

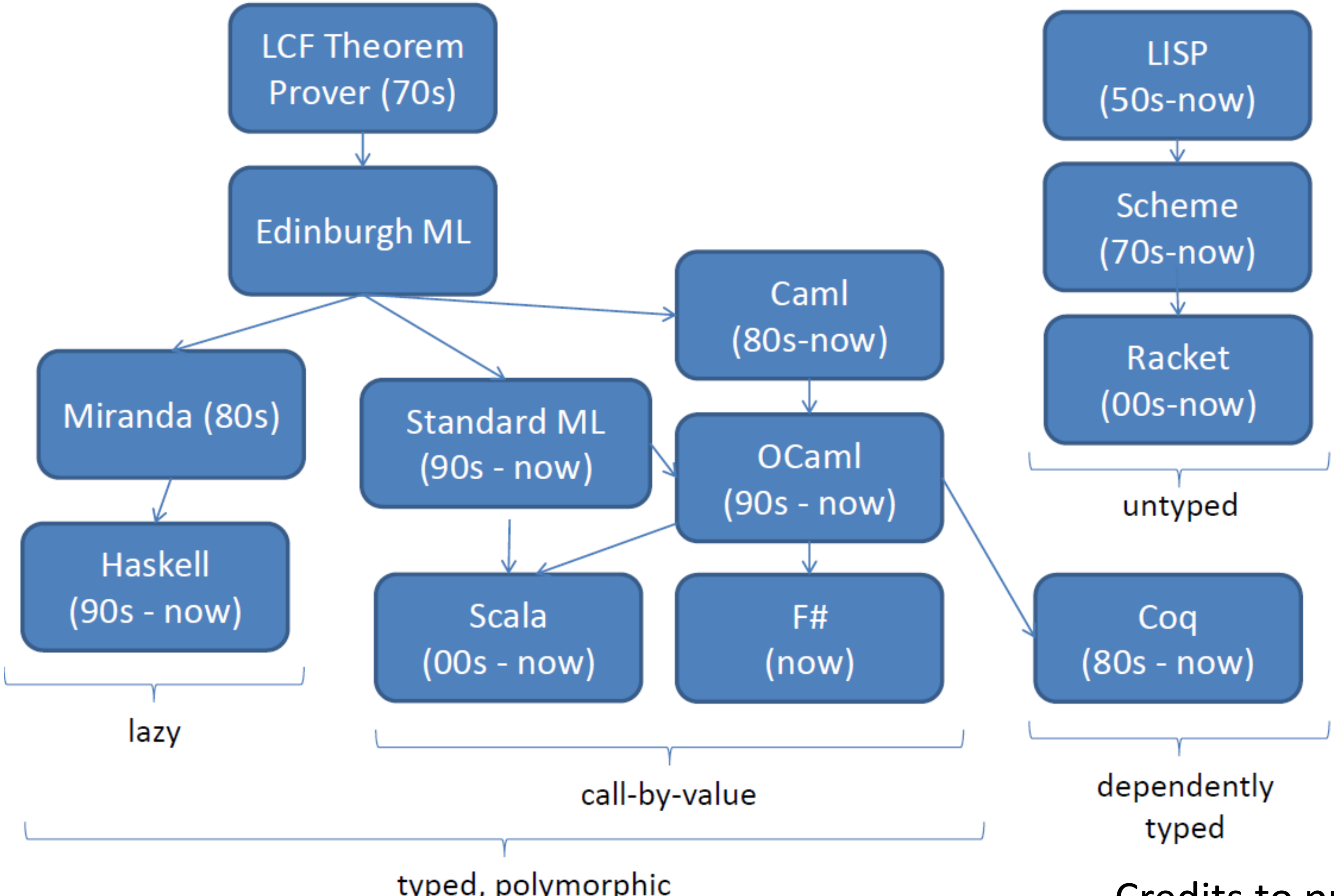
Chiara Di Francescomarino

Today

- Introduction to ML
- Expressions and commands
- Data types and type systems
- Expressions, types and operators in ML

Introduction to ML

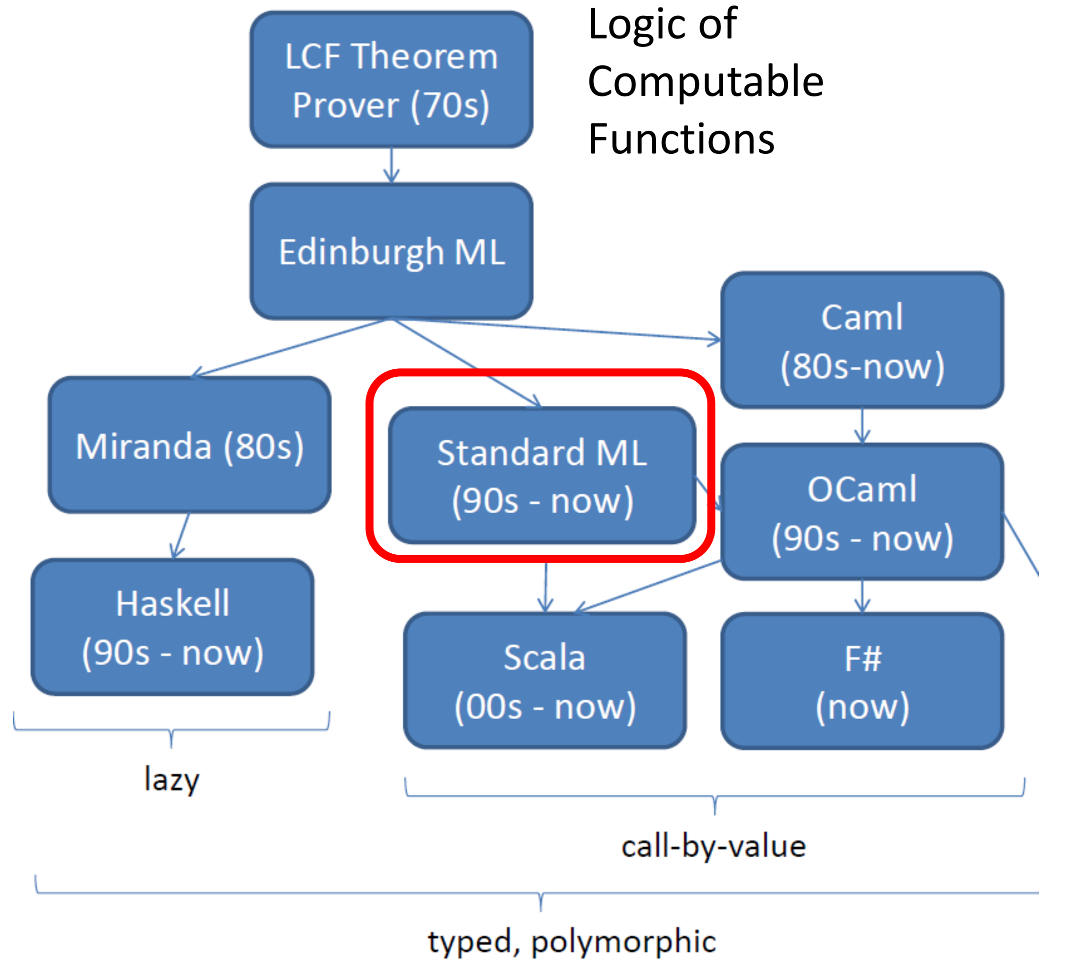
Families



Credits to prof. David Walker

The ML (Meta Language) Programming Language

- General purpose programming language
- It is a meta language for verification purposes
- Different dialects



Standard ML

Robin Milner, Mads Tofte, & Robert Harper
Standard ML
1980's

Why studying ML?

- Functional Programming will make you think differently about programming
  - Mainstream languages are all about states
  - Functional programming is about values
- ML is a practical (small) Programming Language
- New ideas can help make you a better programmer in any language

ML main FP characteristics

1. ML is a functional language:
   - Basic mode of computation: definition and application of functions
   - Functions can be considered as code
   - But they can also be considered as values, i.e., as parameters of other functions
   - Higher-order functions
     - Functions that take functions as values are supported
     - Other languages like C usually only have limited support for this

ML main FP characteristics

2. Recursion
   - Strongly encouraged in ML in preference for while-loops
   - Can be implemented efficiently
   - Iterative constructs are available in ML, when more appropriate
3. Rule-based programming
   - `If-then-else` rules implemented via pattern matching

ML main FP characteristics

4. No side effects

- Computation is by evaluation of expressions, not by assigning values to variables
- In C, `a=b+c` modifies the value of `a` (side effect)
- In ML, `b+c` creates a new element associated with the result
- If needed, side effects are allowed (printing output), but are not the main means of computation

ML characteristics

5. Strong typing
   - All values and variables have types that can be determined at "compile-time"
   - `4` value of integer type, `4.0` value of real type
   - Valuable debugging aid
   - Variable declarations usually not needed

ML characteristics

6. Polymorphism
   - A function can have arguments of different types
   - In C, we may have to create different programs for sorting arrays of strings or reals
   - In ML we can define one program that work for any type
7. Abstract Data Types (Structures)
   - Elegant type system
   - Ability to construct new types

Running ML

- Three ways:
  - Command line poly/interface poly
    - In order to enter through command line type `poly`
    - We see **>** once we are inside the Poly environment
  - If `foo` is a file name, there are two options

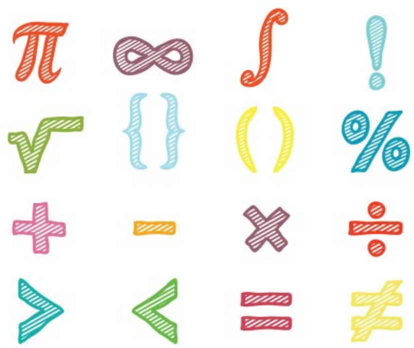
    `poly < foo`

    `use "foo";` (inside the Poly environment)

- Inside the Poly environment
  - `To quit: CTRL+D`
  - `To interrupt: CTRL+C`

Rlwrap poly

In Linux you can find the command "rlwrap poly" that allows you to freely move the cursor within the shell and to enable the history of the commands recently used within the shell environment.



Expressions and commands

What is the difference between expressions and commands?

Expressions

- Syntactic entities whose evaluation either produces a value or does not terminate (undefined expression)
- Usually composed of:
  - A single entity (constant, variable)
  - An operator applied to a number of arguments (which are also expressions)
- Three notational systems:
  - Infix: a + b
  - Prefix (Polish): + a b
  - Suffix (reverse Polish): a b +

brackets and precedence rules to avoid ambiguity

If arity of operators is known –no need of parenthesis and precedence rules

Expression evaluation

- Two evaluation strategies:
  - eager evaluation: first evaluating all the operands and then applying the operator to the values
  - lazy evaluation: operands are evaluated only when needed
- Lazy evaluation could solve the problem of undefined operators

`a == 0 ? b : b/a`	This expression in C demands for lazy evaluation because b/a would be evaluated even when a is equal to 0

Short-circuit evaluation

- Lazy evaluation of boolean expressions is often called short-circuit evaluation
- We arrive at the final value before knowing the value of all of the operands.

`a == 0 || b/a >2`	This expression in C with lazy/ short-circuit evaluation has value true With eager evaluation, we may get an error

- In ML → eager evaluation except for some constructs `andalso`, `orelse` and `if-then-else` that use the lazy evaluation

Commands

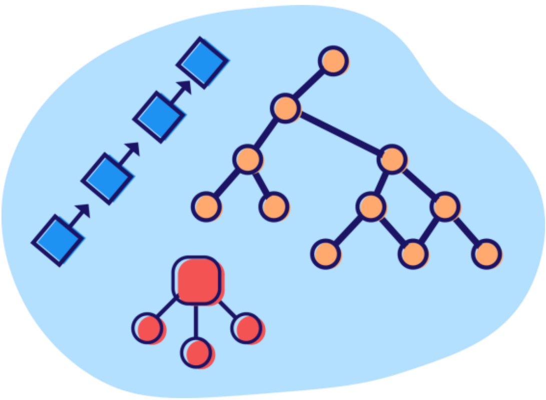
- Command: a syntactic entity whose evaluation does not necessarily return a value but can have a side effect
- Commands
  - Typical of the imperative paradigm
  - Not present in the functional and logical paradigms
  - In some cases yield a result (e.g., an assignment in C)
- The purpose of a command is the modification of the state

Side effects

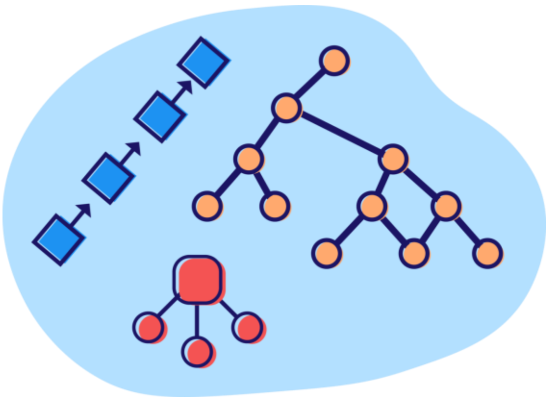
- Side effect: any change in the system state that occurs outside the immediate (local) context of the computation
- In imperative languages expression evaluation can modify the value of variables

`(a+f(b))*(c+f(b))`	if `f` modifies the value of its operand as side effect, the result of the first call to `f` may differ from the second

- Languages follow various approaches:
  - Pure declarative: do not allow side effects
  - Others: forbid the use in expressions of functions that cause side effects
  - Others (e.g., Java): specifies the order of evaluation, from left to right



Data types and type systems



Data types

Data types

- Data type: a collection of values (homogeneous and effectively described) together with a set of operators on these values
- What a type is depends on the specific programming language

What are types used for?

- Design level (conceptual level)
  - organize the information
- Program level (correctness)
  - identify and avoid errors
- Implementation level
  - permit certain optimizations

Conceptual organization

- Different types for different concepts (e.g., price and room)
- Design and documentation purposes
  - “Comments” for the intended use of identifiers but effectively controllable

Correctness

- Types (differently from comments) can be automatically verified
- Every programming language has its own type-checking rules
  - x:=exp they need to have compatible types
  - 3+”pippo” forbidden
  - Call to an object that is not a function or procedure forbidden
- Violation of a type constraint is a possible semantic error (minimal correctness)
- Type checker of the compiler: type constraints must be satisfied before the execution of a program
- Sometimes type rules even too restrictive
  - A subprogram that sorts a vector: it could require different implementations for integers, characters, …

Implementation

- Sources of information
  - Amount of memory to be allocated
    - A Boolean needs fewer bits than a real
    - Precalculate the offset for a record/struct

```
struct Professor{
    char Name[20];
    int Course_code;
}
```

Knowing the type allows us to access p.Course_code, through the offsets from the start address of p in memory

Type systems

Type systems

- The type system of a language:
  1. Predefined types
  2. Mechanisms to define new types
  3. Control mechanisms
     - Equivalence
     - Compatibility
     - Inference
  4. specification of whether types are statically or dynamically checked

Simple and composite types

- Simple (or scalar) types: types whose values are not composed of aggregations of other values
- Composite types: obtained by combining other types using appropriate constructors, e.g., records, vectors, sets, pointers
- We define new types for example with

```
type newtype = expression;
```

Static and dynamic type checking

Static type checking

- At compilation time (C, Java, Haskell, ML)
- Pros
  - At compilation time, before going to the user
  - It is efficient at runtime
- Cons
  - Design is more complex
  - Compilation takes longer
  - More conservative: static type errors are not runtime errors

```
int x;
if (0==1) x="pippo";
```

Dynamic type checking

- During code execution (Python, Javascript, Scheme)
- Pros
  - it locates type errors
- Cons
  - It is not efficient
  - The type error is identified only at runtime with the user

Strongly and weakly typed languages

Strongly typed

- informally, languages that are strict about types
- the type of a value does not change in an unexpected way (e.g., implicit type conversions are not allowed)

ML

```
> 4+"2";
poly: : error: Type error in function application.
   Function: + : int * int -> int
   Argument: (4, "2") : int * string
   Reason:
      Can't unify int (*In Basis*) with string (*In Basis*)
         (Different type constructors)
Found near 4 + "2"
Static Errors
```

Weakly typed

- informally, languages that are more relaxed about types
- the type of a value can change in an unexpected way (e.g., implicit conversions are allowed)

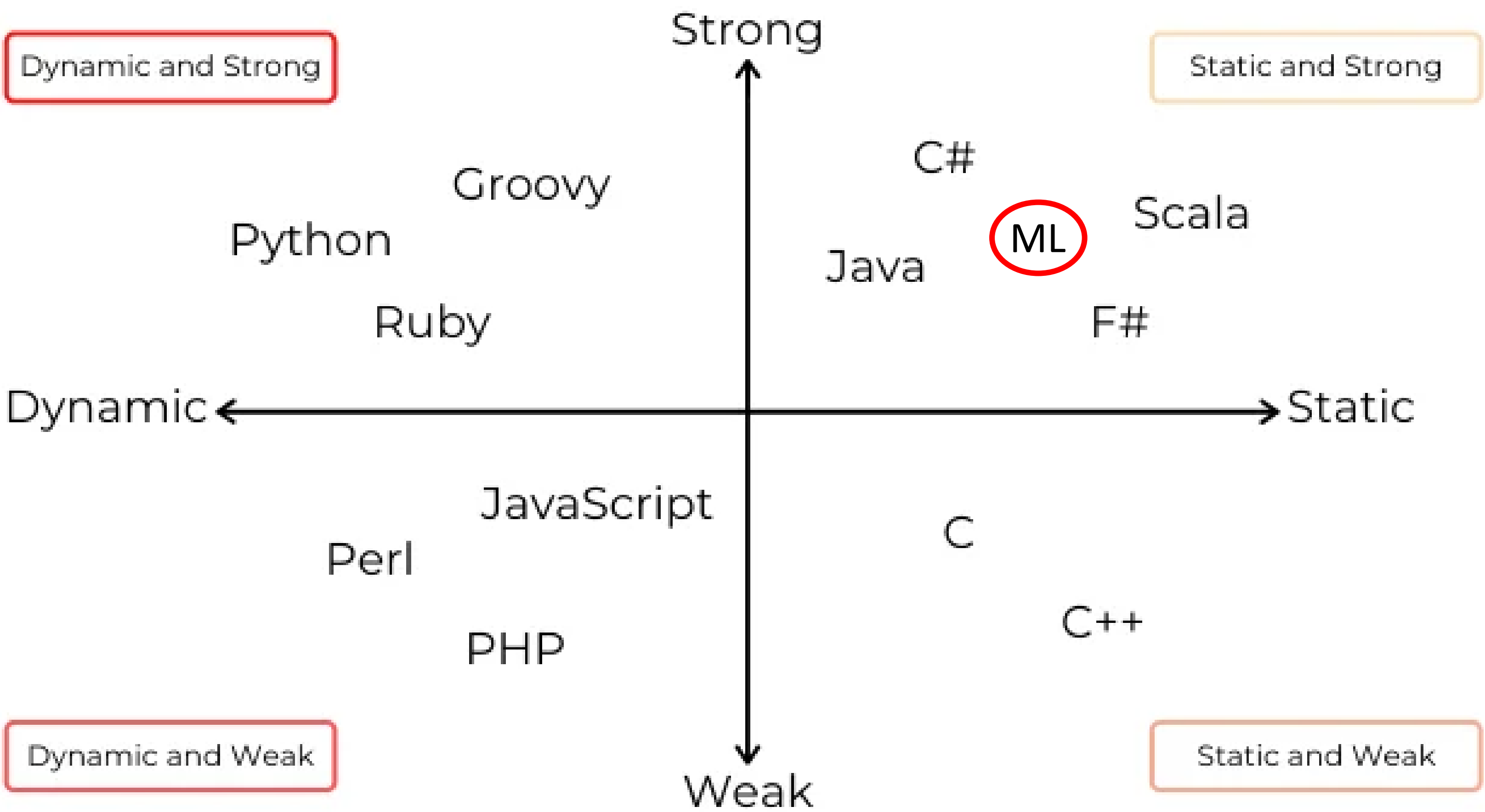
Javascript

```
4 + ‘2’
> ‘42’
```

Php

```
<?php $str = "candies";
$str = $str + 10;
echo ($str); ?>
> 10
```

Strong, weak, dynamic and static



Strong typing is an aspect of type safety

[Prepinsta.com]

Type safety

- A type system is type safe if
  - no program can have undetected errors deriving from type errors
- Type safety features ensure that the code does not perform any invalid operation on the underlying object
  - type error checking can be carried out at compile time or at runtime
- A strongly typed language has a high degree of type safety

C, C++ are type unsafe

- For instance, C and C++ do their best for accommodating casting from one type to another.

```
void func(char* char_ptr) {
    double* d_ptr = (double*) char_ptr;
    (*d_ptr) = 5.0;
    cout << "Value of pointer after cast in func(): " << *d_prt <<
endl; }
```

- The program will claim memory for a double rather than for a char.
- Similarly when allowing the programmer having control over memory allocations

```
int buf[4];
buf[5] = 3; /* overwrites memory */
```

+

x * 4

div

2+3

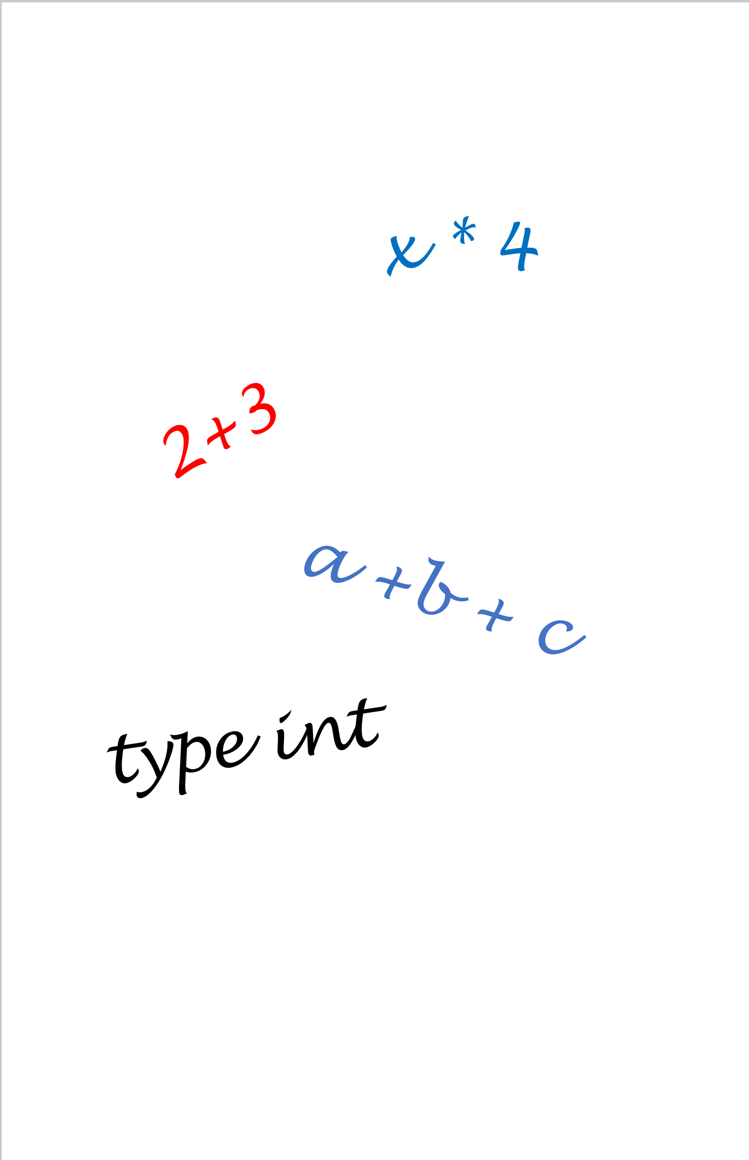
>=

a+b+c

type int

mod

Expressions, types and operators in ML



Expressions and types in ML

Expressions

- Basic notion in a functional language
- Expressions can be values or functions applied to values (to be evaluated)

- Functions and values have types
- ML uses "eager evaluation": first, evaluate the arguments of a function, then the function itself
  - Except for few constructs (e.g., `andalso`, `orelse` and `if then else`)

Example of expressions

```
> 1+2*3;
val it = 7: int
```

Any expression ends with ;

- `val`: value of the expression (7)
- `it`: name of the result
- `int`: type (inferred automatically) of the result

Types

- Basic types

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

  - `int`: Integers, positive and negative. Note that ~3 is -3. This is actually an operator, that negates the integer.
  - `unit`: Single value `()`, used for expressions that do not return a value

- Complex types: constructed starting from other types

We will see complex types in ML in the next lecture

Integers

- Integers
  - Positive integers, e.g.: 0, 1234
  - Negative integers, e.g.: ~1234 (not -1234)
  - Hexadecimals

```
> 0x124;
val it = 292: int
> ~0x124;
val it = ~292: int
```

Reals

- Reals
  - One of more digits
  - At least one of
    - Decimal point and more digits
    - The letter `E` or `e`, and one or more digits
- Examples

```
> ~123.0;
val it = ~123.0: real
> 3E~3;
val it = 0.003: real
> 3.14e12;
val it = 3.14E12: real
```

Booleans

- Booleans
  - `true` and `false` (only lower-case)
- Examples

```
> true;
val it = true: bool
> 1=3;
val it = false: bool
```

Remember that keywords have to be written with the correct case … ML is case sensitive

Characters

- Single character #"a"

```
> #"a";
val it = #"a": char
```

Strings

- Double quotes `"foo"`

```
>"foo";
val it = "foo": string
```

- Special characters for strings
  - `\n`: newline
  - `\t`: tab
  - `\\`: Backslash
  - `\"`: Double-quote
  - `\xyz`: ASCII character with this code (three digits)

^

+

div

>=

*

mod

Operators in ML

Arithmetic operators

- Arithmetic operators
  - `+` and `-`
  - `*`, `/` (division of reals), `div` (division of integers, rounding down), `mod` (remainder of integer division)
  - `~` unary minus
- Usual precedence rules (from left to right): expressions in brackets, unary operators, product and division, addition and subtraction

```
> 3.0 - 4.5 + 6.7;
val it = 5.2: real
> 3 – 4 * 2;
val it = ~5: int
> 43 div (8 mod 3) * 5;
val it = 105: int
> 3 + 4.0;
poly: : error: Type error in function application.
Function: + : int * int -> int
Argument: (3, 4.0) : int * real
Reason: Can't unify int (*In Basis*) with real (*In Basis*)
(Different type constructors)
Found near 3 + 4.0 Static Errors
```

We will be back to this later

Infix operators

- Note the line

```
Function: + : int * int -> int
```

- Operators in ML are prefix form (+(1,2)) with infix notation allowed for some operators for convenience
- + has only one argument and this argument is a tuple of two numbers

```
> (1,2);
val it = (1, 2): int * int
> (1.0,2.0);
val it = (1.0, 2.0): real * real
> (1,2.0);
val it = (1, 2.0): int * real
```

We will see tuples in the next lecture

- In the ML type system, * is used for tuples

String operators

- ^: Concatenation
- Example

```
> "house" ^ "cat";
val it = "housecat": string
> "house" ^ "";
val it = "house": string
```

Comparisons

- `=, <, >, <=, >=, <>`
- Can be used to compare integers, reals, characters, strings (lexicographical order), but:
  - Equality (and non-equality) comparisons of reals is not allowed

Examples

```
> 1 < 2;
val it = true: bool
> 1.0 < 2.0;
val it = true: bool
> 1 = 2;
val it = false: bool
> 1.0 = 2.0;
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
```

We will be back to this later on

Examples

```
> #"Z" < #"a";
val it = true: bool
> "abc" <= "ab";
val it = false: bool
> "abc" <= "acc";
val it = true: bool
```

Lexicographical order: strings are compared lexicographically and, in case of the same prefix (e.g., “abc” and “ab”), the shorter string “precedes” the longer one.

Logical operations

- Operations on booleans:
  - `not`
  - `andalso`: if the first operand is `false`, then it will even not evaluate the second operator
  - `orelse`: if the first operand is `true`, then it will even not evaluate the second operator

Lazy evaluation

- Examples

```
> 1<2 andalso 3>4;
val it = false: bool
> 1<2 orelse 3>4;
val it = true: bool
```

- Take care of the precedence rules: `not` precedence over the others, `andalso` precedence over `orelse`

```
> not 1<2;
poly: : error: Type error in function application.
 Function: < : 'a * 'a -> bool
 Argument: (not 1, 2) : bool * 'a
> not (1<2);
val it = false: bool
```

If-then-else

- Syntax

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

- This (like everything in ML) is an expression. Therefore
  - `else` is required
  - Both parts must have values and the resulting expression a well-defined type

- Example

```
> if 1<2 then 3+4 else 5+6;
val it = 7: int
```

Examples

```
> 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
```

Both then and else parts should be there

```
> if 5<6 then 5 else 6.0;
poly: : error: Type mismatch between then-part and else-part.
   Then: 5 : int
   Else: 6.0 : real
   Reason: Can't unify int to real (Incompatible overloadings)
Found near if 5 < 6 then 5 else 6.0
Static Errors
```

Well-defined type required

What happens if … (without trying it ☺)

- we ask for

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

- we ask for

```
> 1=8;
val it = false: bool
```

- we ask for

```
> 1.0;
val it = 1.0: real
```

Some more … without trying it ☺

How to participate?

1 Go to wooclap.com

2 Enter the event code in the top banner

Event code
BTYJOH

Type systems

- The type system of a language:
    1. Predefined types
    2. Mechanisms to define new types
    3. Control mechanisms
        - Equivalence
        - Compatibility
        - Inference
    4. specification of whether types are statically or dynamically checked

Rules on type correctness

Type equivalence

Type equivalence

- Two types `T` and `S` are equivalent if every object of type `T` is also of type `S`, and vice versa
- Two rules for type equivalence
  - Equivalence by name: the definition of a type is opaque
  - Structural equivalence: the definition is transparent

Equivalence by name

- Two types are equivalent if they have the same name
  - Used Java
  - Too restrictive
  - None of the four

```
type T1 = 1..10;
type T2 = 1..10;
type T3 = int;
type T4 = int;
```

- Loose or weak equivalence by name: Pascal
  - A declaration of an alias of a type generates a new name, not a new type
  - T3 and T4 are names of the same type
- Defined with reference to a specific program, not in general

Structural Equivalence

- Two types are structurally equivalent if they have the same structure: substituting names for the relevant definitions, identical types are obtained.
- Structural equivalence between types is the minimal equivalence relation that satisfies:
  - A type name is equivalent to itself
  - If `T` is defined as `type T = expression`, then `T` is equivalent to `expression`
  - Two types constructed using the same type constructor applied to equivalent types, are equivalent

Structural Equivalence Examples

```
type T1 = int;
type T2 = char;
type T3 = struct{
    T1 a;
    T2 b;
}
type T4 = struct{
    int a;
    char b;
}
```

```
type S = struct{
    int a;
    int b;
}
type T = struct{
    int n;
    int m;
}
type U = struct{
    int m;
    int n;
}
```

- Some aspects are clear
  - T3 and T4 are structurally equivalent
- Other aspects are less clear
  - S, T and U have field names or order that are different: are they equivalent?
  - Usually no, yes for ML.
- Defined in general – not specifically for a program
  - Two equivalent types can be substituted without altering the meaning
  - Referential transparency

Type equivalence in languages

- Combination or variant of the two equivalence rules
  - Pascal → weak equivalence by name
  - Java → equivalence by name – except for arrays with structural equivalence
  - C → structural equivalence for arrays and types defined with typedef but equivalence by name for records and unions
  - ML → structural equivalence except for types defined with `datatype`

Compatibility and conversion

Compatibility

- `T` is compatible with `S` if objects of type `T` can be used in contexts where objects of type `S` are expected
- Example: `int n; float r; r=r+n` in some languages
- In many languages compatibility is used for checking the correctness of:
  - Assignments (right-hand type compatible with left-hand),
  - parameter passing (actual parameter type compatible with formal one), …
- Compatibility is reflexive and transitive but it is not symmetric
  - E.g., compatibility between `int` and `float` but not viceversa in some languages

Compatibility

- The definition depends on the language.
- `T` can be compatible with `S` if
  - `T` and `S` are equivalent
  - The values of `T` are a subset of the values of `S` (interval)
  - All the operations on values of `S` can be performed on values of `T` (extension of record) – sort of subtype
  - There is a natural correspondence between values of `T` and values of `S` (`int` to `float`)
  - The values of `T` can be made to correspond to some values of `S` (`float` to `int` with truncating, rounding)

Type conversion

- If `T` is compatible with `S`, there is some type conversion mechanism.
- The main ones are:
  - Implicit conversion, also called coercion. The language implementation does the conversion, with no mention at the language level
  - Explicit conversion, or cast, when the conversion is mentioned in the program

Coercion

- Coercion indicates, in a case of compatibility, how the conversion should be done
- Three possibilities for realizing the type conversion. The types are different, but
  - Same values and same storage representation for values of $T \subseteq S$. E.g., types that are structurally the same, but have different names
    - Conversion only at compile time → no code to be generated
  - Different values, but the common values have the same representation. E.g., integer interval and integer
    - Code for dynamic control when there is an intersection
  - Different representations for the values. E.g., reals and integers
    - Code for the conversion

Cast

- In certain cases, the programmer must insert explicit type conversion (C, Java: cast)

  `S s = (S) T`

  For example `r = (float) n` and `n=(int) r`

- Cases similar to coercion
- Not every explicit conversion is allowed
  - Only when the language knows how to do the conversion
  - Can always be done when types are compatible (useful for documentation)
- Modern languages prefer cast to coercion.

Summary

- Introduction to ML
- Expressions and commands
- Data types and type systems
- Expressions, types and operators in ML

Readings

- Chapter 6, 7 and 8 of the reference book
  - Maurizio Gabbrielli and Simone Martini "Linguaggi di Programmazione - Principi e Paradigmi", McGraw-Hill

Next time

- Type errors and conversions in ML
- Variables and functions in ML