λ

Data Structures and Abstraction

Programmazione Funzionale

2023/2024

Università di Trento

Chiara Di Francescomarino

Today

- Recap
- Data Types and Type Systems
- Rules on type correctness
- Abstraction of data types

Recap

Functions as parameters

- A function is passed as a parameter to another function and then called through the actual parameter
- Call by name is a special case of functions as parameters
  - Use a function without arguments

```
{int x = 1;
 int f(int y){
    return x+y;
 }
 void g (int h(int b)){
    int x = 2;
    return h(3) + x;
 }
 ...
 {int x = 4;
  int z = g(f);
 }
}
```

Binding policy and scope policy

- Binding policy is independent from scope policy
- Static scoping
  - Deep binding
- Dynamic scoping
  - Deep binding
  - Shallow binding

```
{int x = 1;
 int f(int y){
     return x+y;
 }
void g (int h(int b)){
     int x = 2;
     return h(3) + x;
}
...
 {int x = 4;
  int z = g(f);
 }
}
```

Static and deep	Dyn and deep	Dyn and shallow
x is 1	x is 4	x is 2
h(3) returns 4	h(3) returns 7	h(3) returns 5
z is 6	z is 9	z is 7

Deep vs shallow binding

- Dynamic scope
  - Possible with deep binding
    - Implementation with closure
  - Or shallow binding
    - No special implementation needed
- Static scope
  - Always uses deep binding
    - Implemented with closure

Functions as results

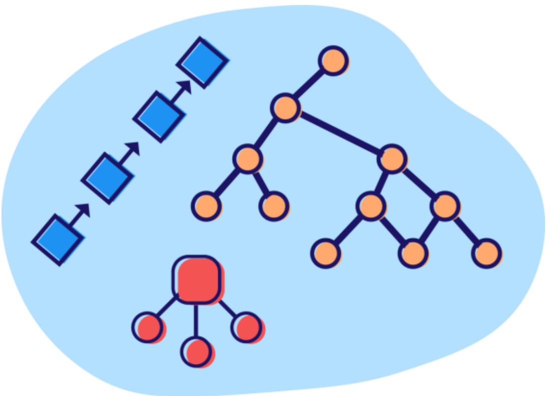
- Generating functions as the result of other functions allows the dynamic creation of functions at runtime

```
{int x = 1;
 void->int F () {
     int g () {
          return x+1;
     }
     return g;
 }
 void->int gg = F();
 int z = gg();
}
```

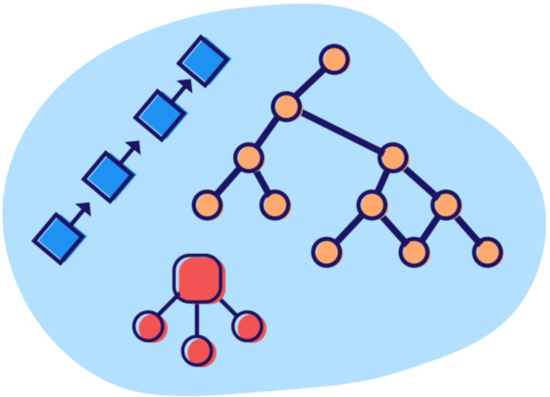
- `void-> int` denotes the type of the functions that take no argument and return an int
- `void->int F()` is the declaration of a function which returns a function of no argument and return value int
- `return g` returns the function and not its application
- gg is dynamically associated with the result of the evaluation of `F`
- The function gg returns the successor of the value of `x`

Returning a function

- A function returned as a result requires besides its code the environment in which the function will be evaluated.
- When a function returns a function as result, the result is a closure
- Applying constraints or implementing a stack in the heap
  - No automatic deallocation
  - Activation record on heap
  - Static or dynamic chain connects the record
  - Call garbage collector when needed



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?

- Project 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 with

```
type newtype = expression;
```

Static and dynamic type checking

Static type checking

- At compilation time (C, Java, Hashkell)
- 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)

Python

4 + ‘2’

Traceback (most recent call last):

File "<string>", line 4, in <module>

ERROR!

TypeError: unsupported operand type(s) for +: 'int' and 'str'

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

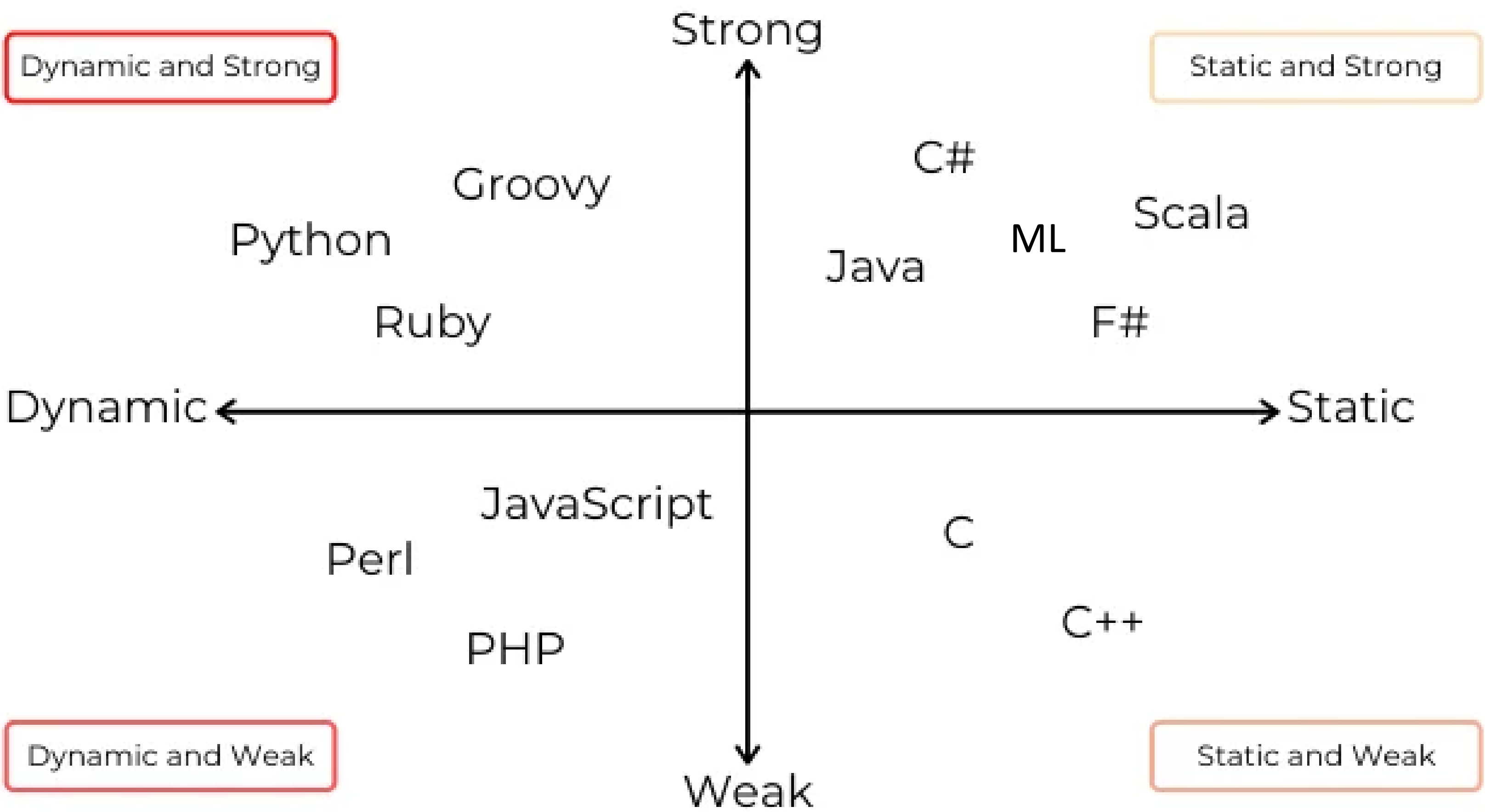
4 + ‘2’

> ‘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 */
```

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

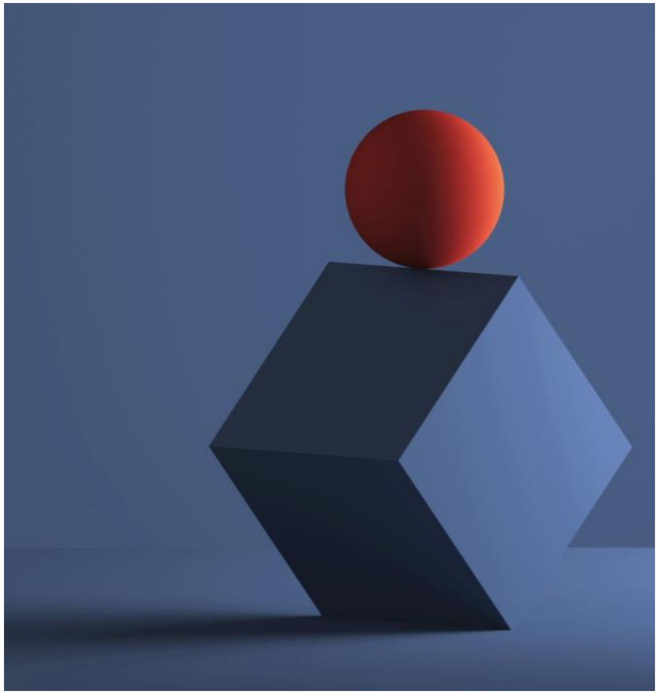
Cast

- 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.



Polymorphism

Polymorphism

- A single value has multiple types
  - Conventional languages allows for some polymorphism
    - Example: +: `int x int→ int` and `float x float → float`, the value `null` has type `T*`,
  - But the user cannot define polymorphic objects
    - E.g., in Pascal, we have different functions for different types `void int sort (int A[])`, `void char sort (char C[])`, …
  - In a polymorphic language
    - `void sort (<T> A[])`
- Three forms of polymorphism
  - Ad hoc polymorphism (overloading)
  - Universal polymorphism
    - Parametric polymorphism (explicit or implicit)
    - Subtype or inclusion polymorphism

Ad-hoc polymorphism: overloading

- The same symbol has different meanings and the context information used to determine the correct one
- Examples
  - +: both integer and real addition (as well as string concatenation)
  - More than one function or constructor with the same name and different parameters
- The compiler translates them in different ways
- Overloading is usually resolved at compile time, after type inference
- Overloading is different from coercion

```
a. 1 + 2
b. 1.0 + 2.0
c. 1 + 2.0
d. 1.0 + 2
```

Depending on the language, we can have

- overloaded with 4 meanings
- overloaded with two meanings (a and b) + coercion in c and d
- only real addition (b) and coercion for b, c and d

Parametric polymorphism

- A value has parametrized universal polymorphism when the value has an infinite number of possible types, obtained by instantiation of a general type schema
- Polymorphic function is a single definition that is applied uniformly to all instances of a general type.
- By denoting with <T> a type variable/ a sort of parameter
  - `null` which is of type `<T>*`
  - `ide(x)=x;` of type `<T> -> <T>`
  - `sort(v);` of type `<T>[] -> void`
  - `swap(x, y);` of type `<T>x<T> -> void`

Polymorphic object instantiations

- A polymorphic object can be instantiated to a specific type
  - Simplest way: directly by the compiler

```
int* k = null;
char v,w;
int i,j;
...
swap(v,w);
swap(i,j);
```

Assignment on a variable of type int* → the type checker instantiates the type of null to int*

Variables of type char → swap instantiated to character

Variables of type int → swap instantiated to int

- General and flexible. Two types:
  - Explicit: explicit annotation (<T>) indicating the types to be considered as parameters (e.g., C++ *template* and Java *generic*)
  - Implicit: the type checker tries to determine for each object the most general type from which the others can be obtained (e.g., ML)

```
fun Comp(f,g,x){return f(g(x));}
```

The most general type is

```
(<S>-><T>)*(<R>-><S>) *<R> -> <T>
```

Subtype polymorphism

- Similar to explicit polymorphism, but not all types can be used to instantiate the general one – instantiation is limited by the structural compatibility between types
- Suppose T is a subtype of S, written `T <: S`
- A value has subtype (or limited) polymorphism if it has an infinite number of possible types, obtained by substituting for a parameter all the subtypes of the given type
- A polymorphic function:

```
∀T<:D.T-> void
```

can be applied uniformly to all values (any legal instances) of any subtype of D

Exercise 5.1

- Given the following type definitions in a programming language which uses structural type equivalence:
- In the scope of the declarations `T3 a` and `T4 b`, is the assignment `a = b` permitted? Why?

```
type T1 = struct{
     int a;
     bool b;
}
type T2 = struct{
     int a;
     bool b;
}
type T3 = struct{
     T2 u;
     T1 v;
}
type T4 = struct{
     T1 u;
     T2 v;
}
```

Solution exercise 5.1

- Given the following type definitions in a programming language which uses structural type equivalence:
- In the scope of the declarations `T3 a` and `T4 b`, is the assignment `a = b` permitted? Why?

```
type T1 = struct{
     int a;
     bool b;
}
type T2 = struct{
     int a;
     bool b;
}
type T3 = struct{
     T2 u;
     T1 v;
}
type T4 = struct{
     T1 u;
     T2 v;
}
```

Solution

- In some implementations this structure could be illegal
- In others, it may be permitted since `T1` and `T2` are structurally equivalent, so `T4` may be considered to be compatible with type `T3`

Exercise 5.2

- Which type is assigned to each of the following functions using polymorphic type inference?

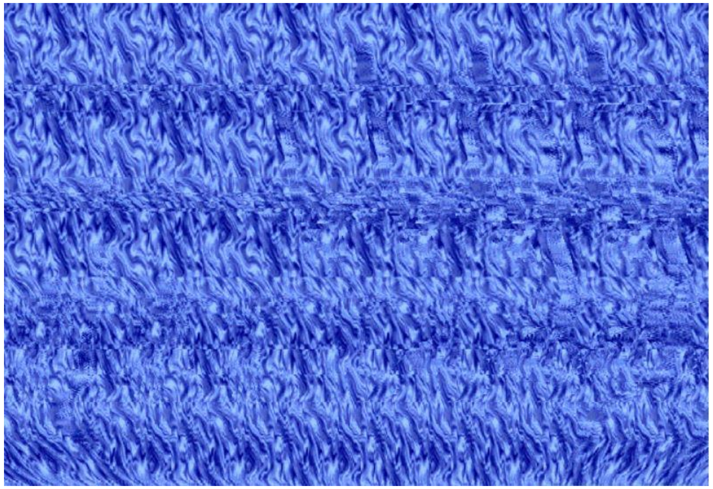
```
fun G(f,x){return f(f(x));}
fun H(t,x,y){
    if (t(x))
        return x;
    else return y;}
fun K(x,y){return x;}
```

Solution exercise 5.2

- Which type is assigned to each of the following functions using polymorphic type inference?

```
fun G(f,x){return f(f(x));}
fun H(t,x,y){
    if (t(x))
        return x;
    else return y;}
fun K(x,y){return x;}
```

```
G:('a-> 'a)*'a->'a
H : ('a -> bool)* 'a*'a->'a
K: 'a*'b ->'a
```



Data abstraction

Data types

- Data type is a high-level concept: it allows for abstracting from pure bits
- Data types: specify the values (of sequences of bits) and operations allowed on those values
  - `integer` consists of values `[-maxint .. maxint]` and operations `{+, -, *, div, mod}`
  - These operations are the only way to manipulate integers
  - Each value is wrapped in an encapsulation (its type)

Defining new data types

- When defining new data types, a user can only use existing capsules and a new type does not allow the user to define types at the same level of abstraction of the predefined types
  - It is possible to define new values
  - But the internal structure and operations are still accessible to the programmer

```
type Int_Stack = struct{
    int P[100]; // the stack proper
    int top; // first readable element
}
Int_Stack create_stack(){
    Int_Stack s = new Int_Stack();
    s.top = 0;
    return s;
}
Int_Stack push(Int_Stack s, int k){
    if (s.top == 100) error;
    s.P[s.top] = k;
    s.top = s.top + 1;
    return s;
}
int top(Int_Stack s){
    return s.P[s.top];
}
Int_Stack pop(Int_Stack s){
    if (s.top == 0) error;
    s.top = s.top - 1;
    return s;
}
bool empty(Int_Stack s){
    return (s.top == 0);
}
```

An example

Even in case of equivalence by name, we can access the stack in its representation as an array

```
int second_from_top()(Int_Stack c){
    return c.P[s.top - 1];
    }
```

We would need … linguistic support for abstraction

- Abstraction of control
  - Hide the implementation of procedure bodies
- Data abstraction
  - Hide decisions about the representation of the data structures and the implementation of the operations
  - Example: a stack implemented via
    - A vector
    - A linked list

Abstract Data Types

- One of the major contributions of the 1970s
- Basic idea: separate the interface from the implementation
  - Interface: types and operations that are accessible to the user
  - Implementation: internal data structures and operations acting on the data types
  - Example
    - Sets have operations as `empty`, `union`, `insert`, `is_member?`
    - Sets can be implemented as vectors, lists etc.

Abstract Data Types characteristics

1. A name for the type
2. An implementation or representation for the type (concrete type)
3. Names denoting the operations for manipulating the values of the type with their types
4. For every operation, an implementation that uses the concrete type representation
5. A security capsule which separates the name of the type and those of the operations from their implementations

An example

```
abstype Int_Stack{
        type Int_Stack = struct{
                int P[100];
                int n;
                int top;
        }
        signature
        Int_Stack create_stack();
        Int_Stack push(Int_Stack s, int k);
        int top(Int_Stack s);
        Int_Stack pop(Int_Stack s);
        bool empty(Int_Stack s);
        operations
        Int_Stack create_stack(){
                Int_Stack s = new Int_Stack();
                s.n = 0;
                s.top = 0;
                return s;
        }
        Int_Stack push(Int_Stack s, int k){
                if (s.n == 100) error;
                s.n = s.n + 1;
                s.P[s.top] = k;
                s.top = s.top + 1;
                return s;
        }
        int top(Int_Stack s){
                return s.P[s.top];
        }
        Int_Stack pop(Int_Stack s){
                if (s.n == 0) error;
                s.n = s.n - 1;
                s.top = s.top - 1;
                return s;
        }
        bool empty(Int_Stack s){
                return (s.n == 0);
        }
}
```

Name of the abstract data type `Int_Stack`

Representation or *concrete type*

Names and types of the operations

Implementation of the operations

- Inside, `Int_Stack` is a synonym of the concrete representation
- Outside, no relation between an `Int_Stack` and its concrete type
- No way to manipulate `Int_Stack` e.g., through:

```
int second_from_top()(Int_Stack c){
     return c.P[s.top - 1];
     }
```

Concrete languages

- Different languages have different levels of support for ADT
- C:
  - Header file (.h) containing the interface/signature
  - Implementation in separate .c files
- Java, C++:
  - Object-orientation – through classes
    - Methods implementing the interface are public
    - Internal representation private
- ML:
  - Signatures and structures

Summary

- Data Types and Type Systems
- Rules on type correctness
- Abstraction of data types

Readings

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

Next time

- Logic paradigm