LTP

Unit 1 – Programming languages

Type: set of values that can be assigned to a variable or expression (to avoid errors, give structure to the information, handle data structures). Languages can be typed or untyped (all vars have a universal type).

Type systems:

- Value and variable type must coincide and/or must be compatible
- Static typing: type does not change during execution, Dynamic typing: can change
- Explicing typing: types are part of the syntax, Implicit typing: types do not need to be part of the syntax
- Monomorphic types: type expression contains no type variable (Int -> Int)
- Polymorphic types: type expression contains type variables ([a] -> Int)

Polymorphism: permits the use of values of different types under an uniform interface

- A function can be polymorphic (operators such as +, -)
- A data type can be polymorphic (a list or array)

Polymorphism types:

- Overloading: several functions share a single name (arithmetic operations, overloaded functions)
- Coercion: type conversion of arguments
    - Implicit conversion (in java, sum of double + int -> double)
    - Explicit conversion (in java, casts [(int) 2.5])
- Genericity/Parametric: function definition has a common structure for any set of types (in java, generic types, an example, a dictionary)
- Inclusion/Inheritance: definition of a function based on inheritance
    - Abstract methods have no implementation. Subclasses of abstract classes must implement each abstract method.
    - If Circle inherits Shape, Shape s = new Circle() is correct, but List<Shape> = new List<Circle> is not

Reflection: enabled a program to see and modify its own structure

- Advanced but simple feature, specially in functional languages due to data/program duality (homoiconicity)
- Reflection in Java allows inspection of classes, interfaces and attributes/methods without knowing their names

Procedures and control flow:

- Parameter passing: when a method is called there is a change in the execution context
    - Call by value: values are computed and copied into the function
    - Call by reference: a reference is passed so the function works with the same memory objects
    - Call by need: expressions passed are evaluated only when used in the function
- Variable scope: if a variable can be reached from a specific point
    - Static scope: defined in compilation time (modern programming languages)
    - Dynamic scope: defined during the execution

Memory management: the compiled program, temporary values, call and return operations, buffers, and some insertion/destruction operations require storage during program execution. Some approaches:

- Static allocation: computed and assigned in compilation time (efficient but incompatible with dynamic data structures)
- Dynamic allocation: computed and assigned during execution
    - Stack-based: set up as a sequential block in memory
    - Heap-based: region of storage where subblocks can be allocated and removed at arbitrary times (explicit/implicit [where variable can't be reached anymore]), for example the Java Garbage Collector (it identifies unreachable elements)

Programming paradigms:

- Imperative: sequence of commands that change the state of the machine. Instructions are sequentially processed, and the program builds the sequence of machine states that lead to the solution (Pascal).
    - Main point: how to solve a problem
    - Destructive assignment: understanding code is harder
    - Programmer responsible for control issues
- Declarative: program describes the properties of the desired solution (PROGRAM = LOGIC + CONTROL). (Haskell)
    - Functional paradigm: data structures and functions defined via equations
    - Logic paradigm: relations among objects defined by means of rules
    - Specification of solution to a given problem
    - Easy to verify and modify, efficient as Java
- Object-Oriented: object = state + operations. Concepts such as class, subclass, inheritance, instance, abstraction, encapsulation, modularity, hierarchy
- Concurrent: simultaneous execution of multiple interactive tasks. Comes from the concurrency problems such as data corruption and indeterminism. Dijkstra introduced

the first abstractions (semaphores) and Hoare introduced the notion of critical region. In java, we have threads (using inheritance or interfaces).

  - Threads are like processes, but all resources belong to a root program
  - In java undesirable behaviours are left to the programmer
  - Parallel programming: use of more than one processor core to split execution to make it faster

- Interaction-based paradigm: programming as calculation. Collection of entities that interact according to some rules, for instance, event-driven programming, embedded systems, client/server architectures.
  - Control flow is defined by events
  - Not bound to any programming paradigm, as it must only catch signals and manage an event queue
  - Very used in GUI development, but difficulty to extend and error prone
- Bio-computation: inspired in biology
- Quantum computing: replaces classical circuits by others that take benefits from quantum effects

Some multiparadigm languages: Python, Scala

Unit 2 – Foundations of programming languages

Syntax and static/dynamic semantics

Syntax: allowed character sequences of a program

Semantics: the computational meaning of a program

BNF Notation:

- <name>: we give name to an expression defined by rules
- ::== (assignment), | (or)
- [name]: optional item
- {name} / name*: sequence of 0 or more items
- name+ sequence of 1 or more items
- Example: <while_statement> ::== while (<expression>) <statement>

How the program is compiled:

1- Lexical analyser: decomposes characters into a sequence of tokens

2- Syntactic analyser: recognizes sequence of tokens and builds a sequence of instructions (parse tree)

3- Semantic analyser: variables declared, type compatibility, function profiles…

4- Optimization of the code

5- Generation of object code

6- Link of the code to other libraries to obtain an executable file

Static/Dynamic semantics:

- Static semantics: can be checked during compilation time
- Dynamic semantics: can only be checked during execution

Operational Semantics

State of the machine (s): mapping of variables {X → 3, Y → -4} WARNING: $\mapsto$

Machine configuration: pair consisting of the current state and instruction to evaluate <i, s>

Program execution is formalized via an arrow (→)

Transition rules: $\frac{premise}{<i,s> \rightarrow <i',s'>}$, <I', s'> is obtained if the premise is satisfied

- Evaluation of arithmetic instructions: <exp, s> ⇒ n
- Direct computation of a final state: <i, s> ⇓ s'

Small step semantics (relates to instructions and states)

Sequence: $\frac{<i1,s> \rightarrow <i1',s'>}{<i1;i2,s> \rightarrow <i1';i2,s'>}$

Assignment: $\frac{<a,s> \Rightarrow n}{<x:=a,s> \rightarrow <skip,s[x \rightarrow n]>}$

Big step semantics (relates to configurations and state)

Sequence: $\frac{<i1,s> \Downarrow s1 \; <i2,s1> \Downarrow s'}{<i1;i2,s> \Downarrow s'}$

Assignment: $\frac{<a,s> \Rightarrow n}{<x:=a,s> \Downarrow s[x \rightarrow n]}$

Axiomatic Semantics

Hoare triple: {P} S {Q}, if S satisfies P, the final state S' also satisfies Q

- P restricts input states to S
- S is the instruction set
- Q is the desired output states

Weakest precondition: wp(i, Q)

- For any state before i, Q should hold

Example (assignment): wp(x:=a,Q) = Q[x→a]

Example 2: wp(x:=x-1,x>0) = (x-1>0) = x>1

Semantic properties

Two programs P and P' are equivalent if S(P) = S(P')

Small step and big step semantics are not usually equal (different number of steps)

Implementation of programming languages

- Compiled languages:
    - Source program → *Compiler* → Object program
    - Input → *Object program* → Output
- Interpreted languages:
    - Source program & Input → *Interpreter* → Output

Pure translation and interpretation are rare. Usually a mixed interpretation is used: source program is translated into a more executable format, which is then executed by an interpreter