Programming Paradigms

Overview

Keith Mannock

Department of Computer Science and Information Systems
Birkbeck, University of London

† Thanks to Sven Helmer for the basis for this slide deck.

Elements of Programming Languages

- We are going to have a quick look at the following concepts
  - Compiled/Interpreted
  - Syntax
  - Semantics
  - Typing

Compiled vs. Interpreted Languages

- Compiled languages are translated into a form that can be run directly on a computer's processor
  - Usually the whole program is translated before it is run
- Interpreted languages are processed by a higher-level virtual machine
  - Usually a program is translated on the fly, i.e., a statement is translated and then immediately executed

Compiled Languages

Source code

↓

Lexical/Syntactical Analysis

↓

Type Checker

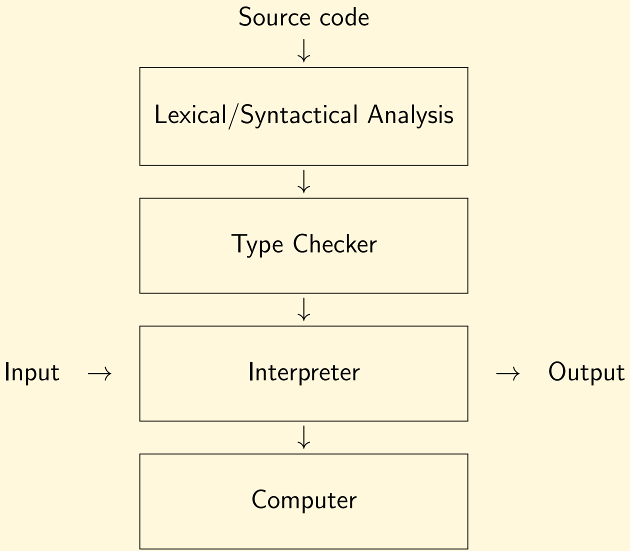
↓

Code generation/optimization

↓

Input → Computer → Output

Interpreted Languages



Syntax I

- The syntax of a language describes how well-formed expressions should look like
  - This includes putting together symbols to form valid tokens
  - As well as stringing together tokens to form valid expressions
- For example, the following (English) sentence is not correct:

  "Furiously slqxp ideas grn colorless."

- While

  "Colorless green ideas sleep furiously."

  is syntactically correct (but it does not make any sense).

Syntax II

- The syntax of a programming language is usually described by a formalism called *grammar*
- More details on this can be found on an appropriate compilers course (e.g., see `Coursera`)

Semantics I

- Semantics is concerned with the meaning of (programming) languages
  - Usually much more difficult to define than syntax
- A programmer should be able to anticipate what will happen before actually running a program
- An accurate description of the meaning of language constructs has to be worked out

Semantics II

- There are different ways of describing semantics of programming languages
- Main approaches are:
  - Operational semantics
  - Axiomatic semantics
  - Denotational semantics

Operational Semantics

- In operational semantics the behaviour is formally defined by an interpreter
  - This can be an abstract machine, a formal automaton, a transition system, etc.
  - In the extreme case, a specific implementation on a certain machine (1950s: first version of Fortran on an IBM 709)

Axiomatic Semantics I

- Axiomatic semantics uses logic inference to define a language
- An example is Hoare logic
  - $\{P\}C\{Q\}$; if precondition P is true, then execution of command C will lead to postcondition Q
- Axiomatic semantics does have some limitations:
  - Side effects are disallowed in expressions;
  - the `goto` command is difficult to specify;
  - aliasing is not allowed; and
  - scope rules are difficult to describe unless we require all identifier names to be unique.

Axiomatic Semantics II

- Despite these limitations, axiomatic semantics is an attractive technique because of its potential effect on software development:
  - The development of *bug free* algorithms that have been proved correct.
  - The automatic generation of program code based on specifications.

Denotational Semantics

- Denotational semantics defines the meaning of each phrase by translating it into a phrase in another language
  - Clearly, assumes that we know the semantics of this target language
- Target language is often a mathematical formalism

Typing

- A programming language needs to organise data in some way
- The constructs and mechanisms to do this are called type system
- Types help in
  - designing programs
  - checking correctness
  - determining storage requirements

Type System

The type system of a language usually includes

- a set of predefined data types (e.g. integer, string)
- a mechanism to create new types (e.g. typedef)
- mechanisms for controlling types:
  - equivalence rules: when are two types the same?
  - compatibility rules: when can one type be substituted for another?
  - inference rules: how is a type assigned to a complex expression?
- rules for checking types (e.g. static vs. dynamic)

Data Types

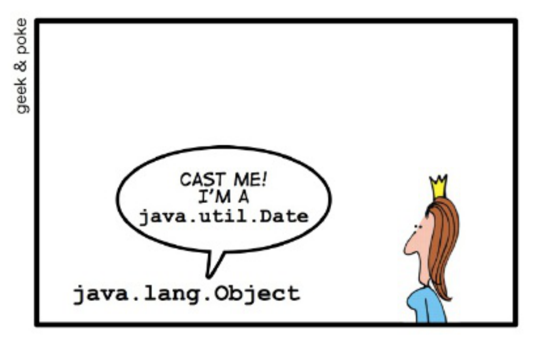
- A language is *typed* if it specifies for every operation to which data it can be applied
- Languages such as assembly or machine languages can be *untyped*
  - Assembler: all data is represented by bitstrings (to which all operations can be applied)
- Languages such as markup or scripting languages can have very few types
  - XML with DTDs: elements can contain other elements or parsed character data (#PCDATA)

Type Checking I

- There is a distinction between *weak typing* and *strong typing*
- In *weak typing* one type can be interpreted as another
  - For example a string representing a number "3.4028E+12" is treated as a number
- In *strong typing* applying the wrong operation to typed data will raise an error
  - Languages supporting strong typing are also called *type-safe*

Type Checking II

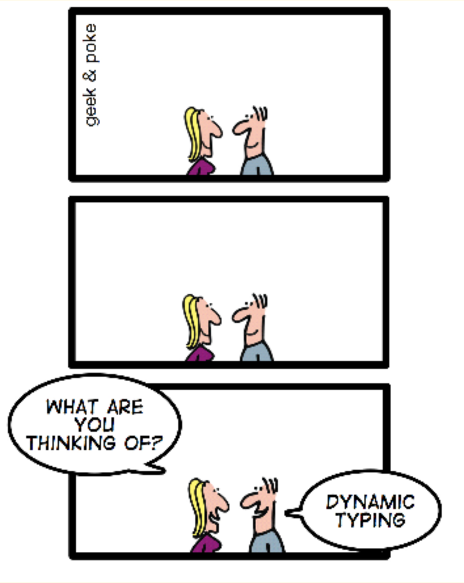
- In some languages it is possible to bypass typing by casting one type into another



Type Checking III

- We also distinguish between languages depending on *when* they check typing constraints
- In *static typing* we check the types and their constraints *before* executing the program
  - Can be done during the compilation of a program
- When using *dynamic typing*, we check the typing *during* program execution

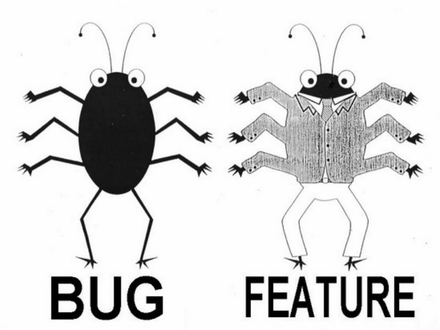
Static vs. Dynamic Typing I



Static vs. Dynamic Typing II

- Although some people feel quite strongly about this, each approach has pros and cons
- Static typing:
  - \+ less error-prone
  - \- sometimes too restrictive
- Dynamic typing:
  - \+ more flexible
  - \- harder to debug (if things go wrong)

Bugs or Features?

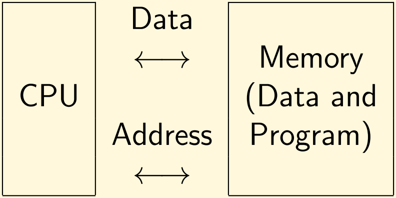


Some paradigms

- After this brief introduction we are now going to have a (brief) look at some programming paradigms and consider their characteristics, specifically
    - imperative (procedural)
    - functional
    - logic programming
    - object-oriented
    - concurrent

Imperative Paradigm I

- The *imperative paradigm* is one of the oldest and is based on the von Neumann architecture



Imperative Paradigm II

Characteristics

- Discipline and idea — Digital hardware technology and the ideas of Von Neumann
- Incremental change of the program state as a function of time.
- Execution of computational steps in an order governed by control structures

  We call the steps commands
- Straightforward abstractions of the way a traditional Von Neumann computer works

Imperative Paradigm III

- Similar to descriptions of everyday routines, such as food recipes and car repair
- Typical commands offered by imperative languages assignment, IO, procedure calls
- Example languages — Fortran, Algol, Pascal, Basic, C
- The natural abstraction is the procedure
  - abstracts one or more actions to a procedure, which can be called as a single command
  - coined the phrase, Procedural Programming

Imperative Paradigm IV

- Example of computing the factorial of a number:

```
unsigned int n = 5;
unsigned int result = 1;
while(n > 1) {
  result *= n;
  n--;
}
```

Imperative Paradigm V

Procedures can be used the same way that built-in commands are used (allows re-usability)

- Some state changes are localised in this way
- Creating a procedure from the previous example:

```
int factorial(unsigned int n) {
    unsigned int result = 1;
    while(n > 1) {
        result *= n;
        n--;
    }
    return result;
}
```

Functional Paradigm I

Evaluate an expression and use the resulting value for something
Characteristics:

- Discipline and idea
  Mathematics and the theory of functions
- The values produced are non-mutable
  - Impossible to change any constituent of a composite value
  - As a remedy, it is possible to make a revised copy of composite value
- Atemporal
  Time only plays a minor role compared to the imperative paradigm
- Applicative
  All computations are done by applying (calling) functions

Functional Paradigm II

- The natural abstraction is the function
  Abstracts a single expression to a function which can be evaluated as an expression
- Functions are first class values
  Functions are full-fledged data just like numbers, lists, ...
- Fits well with computations driven by needs
  Opens a new world of possibilities

Logic Paradigm I

Answer a question via search for a solution
Characteristics:

- Discipline and idea
- Automatic proofs within artificial intelligence
- Based on axioms, inference rules, and queries.
- Program execution becomes a systematic search in a set of facts, making use of a set of inference rules

Object-Oriented Paradigm I

Send messages between objects to simulate the temporal evolution of a set of real world phenomena

Characteristics:

- Discipline and idea
- The theory of concepts, and models of human interaction with real world phenomena
- Data as well as operations are encapsulated in objects
- Information hiding is used to protect internal properties of an object
- Objects interact by means of message passing
- A metaphor for applying an operation on an object

Object-Oriented Paradigm II

- In most object-oriented languages objects are grouped in classes
- Objects in classes are similar enough to allow programming of the classes, as opposed to programming of the individual objects
- Classes represent concepts whereas objects represent phenomena
- Classes are organised in inheritance hierarchies
- Provides for class extension or specialisation

Concurrent Paradigm

Characteristics:

- Performance
- Throughput
- Utilisation of system resources

Concurrency or Parallelism, what's the difference?

Concurrency:

- Logically simultaneous processing.
- Does not require multiple processing elements
- Requires interleaved execution on a single processing element.

Parallelism:

- Physically simultaneous processing.
- It does involve several processing element

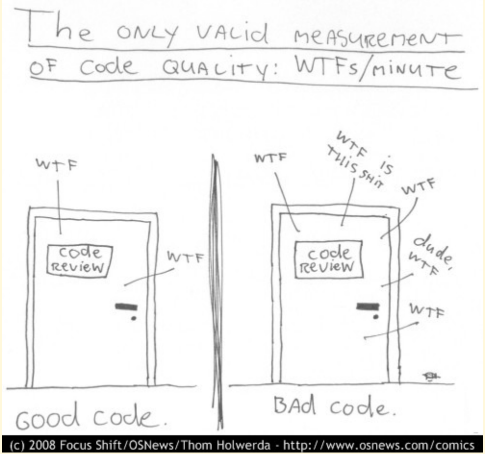
Both concurrency and parallelism require controlled access to shared resources.

In general people use the words concurrent and parallel interchangeably.

A concurrent program is. . .

. . . a program that has multiple threads or tasks of control, allowing it perform multiple computations in parallel and to control multiple external activities that occur at the same time.

Questions thus far...



and onto something we *sort of know* ... Objects!