Programming Paradigms

Unit 1 — Introduction and Basic Concepts

J. Gamper

Free University of Bozen-Bolzano
Faculty of Computer Science
IDSE

Acknowledgements: I am indebted to Sven Helmer for providing me the slides.

Outline

1. Introduction
2. Basics of Programming Languages

Outline

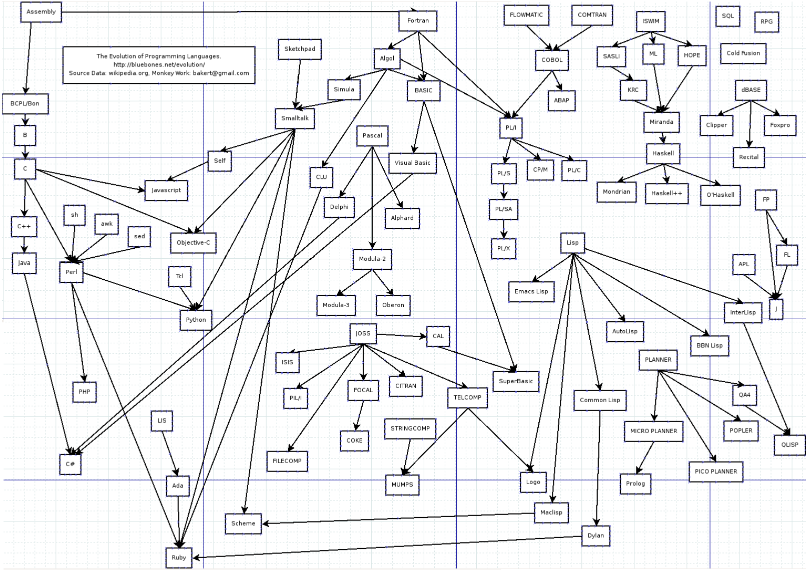
Programming Languages and HCI

- A programming language is an artificial language designed to communicate instructions to a machine, e.g., computer
- The earliest programming languages preceded the invention of the computer
    - e.g., used to direct the behavior of machines such as Jacquard looms and player pianos.
- *"Programming languages are the least usable, but most powerful human-computer interfaces ever invented"*
  Any Ko, http://bit.ly/1iVxF3A
    - The difference between a programming language and a GUI comes down to just a few key differences

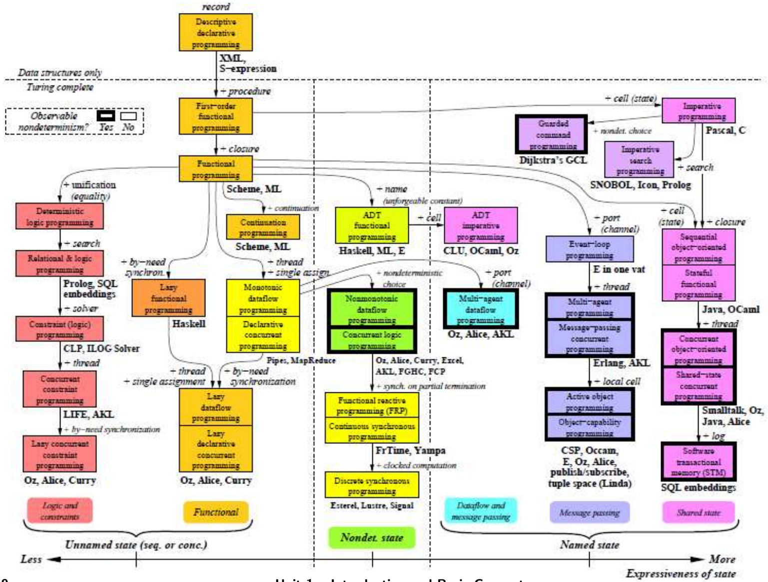
How Many Programming Languages Exist?/1

- There are thousands of programming languages
  - The encyclopaedia britannica mentiones over 2,000 computer languages
  - As of May 2006 Diarmuid Pigott's Encyclopedia of Computer Languages hosted at Murdoch University, Australia lists over 8,000 computer languages
  - Still many new ones are created every year
  - (there are approximately 7,000 current human languages)
- Few languages ever become sufficiently popular that they are used by more than a few people
- But professional programmers may use dozens of languages in a career

How Many Programming Languages Exist?/2



How Many Programming Languages Exist?/3



Programming Languages and Abstraction

- Programming languages provide an abstraction from a computer's instruction set architecture
- Low-level programming languages provide little or no abstraction, e.g., machine code and assembly language
  - Difficult to use
  - Allows to program efficiently and with a low memory footprint
- High-level programming languages isolate the execution semantics of a computer architecture from the specification of the program
  - Simplifies program development

Machine code

```
8B542408 83FA0077 06B80000 0000C383
C9010000 008D0419 83FA0376 078BD98B
B84AEBF1 5BC3
```

Assembly language

```
mov edx, [esp+8]
cmp edx, 0
ja @f
mov eax, 0
ret
```

High-level language

```
unsigned int fib(unsigned int n) {
    if (n <= 0)
      return 0;
    else if (n <= 2)
      return 1;
    else
      ...
}
```

Programming Paradigms/1

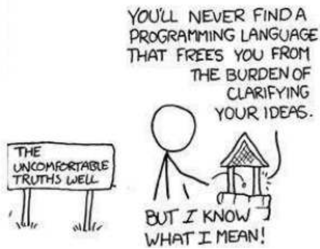
- Programming languages can be categorized into programming paradigms
- Meaning of the word 'paradigm'
  - *"An example that serves as pattern or model"*
    The American Heritage Dictionary of the English Language
  - *"Paradigms emerge as the result of social processes in which people develop ideas and create principles and practices that embody those ideas"*
    Thomas Kuhn, "The Structure of Scientific Revolutions"
- Programming paradigms are the result of people's ideas about how computer programs should be constructed
  - Patterns that serves as a "school of thoughts" for programming of computers

Programming Paradigms/2

Language	Intended use	Paradigm(s)	Standardized?
ActionScript 3.0	Application, client-side, Web	event-driven, imperative, object-oriented	1996, ECMA
Ada	Application, embedded, realtime, system	concurrent,[2] distributed,[3] generic,[4] imperative object-oriented,[5] procedural,[6]	1983, ANSI, ISO, GOST 27831-88[7]
Aldor	Highly domain-specific, symbolic computing	imperative, functional, object-oriented	No
ALGOL 58	Application	imperative	No
ALGOL 60	Application	imperative	1960, IFIP WG 2.1, ISO[8]
ALGOL 68	Application	concurrent, imperative	1968, IFIP WG 2.1, GOST 27974-88,[9]
Ateji PX	Parallel application	object-oriented, pi calculus	No
APL	Application, data processing	array-oriented, tacit	1989, ISO
Assembly language	General	any, imperative	No
AutoHotkey	GUI automation (macros), highly domain-specific	imperative	No
AutoIt	GUI automation (macros), highly domain-specific	event-driven, imperative, procedural	No
BASIC	Application, education	imperative, procedural	1983, ANSI, ISO
BBj	Application, business, Web	object-oriented, procedural	No
BeanShell	Application, scripting	functional, imperative, object-oriented, reflective	In progress, JCP[10]
BLISS	System	procedural	No
BlitzMax	Application, game	imperative, object-oriented, procedural	No
Boo	Application		No
C	System[11]	imperative, procedural	1989, ANSI C89, ISO C90/C99
C++	Application, system	generic, imperative, object-oriented, procedural	2011, ISO
C#	Application, business, client-side, general, server-side, Web	functional,[12] generic, imperative, object-oriented, reflective	2000, ECMA, ISO[13]
Clarion	General, business, Web	functional,[14] imperative, object-oriented	Unknown
Clean	General	functional, generic	No

Programming Paradigms/3

- Once you have understood the general concepts of programming paradigms, it becomes easier to learn new programming languages
- However, this does not mean that by just picking the right paradigm all problems vanish into thin air



- Or put more elegantly:

"There does not now, nor will there ever exist, a programming language in which it is the least bit hard to write bad programs."

L. Flon

Principal Programming Paradigms

- Imperative / Procedural
- Functional
- Object-Oriented
- Concurrent
- Logic
- Scripting

- In reality, very few languages are "pure"
  - Most combine features of different paradigms

Brief Overview of the Course Content

- Brief recapitulation
  - Elements of programming languages
  - Imperative/procedural paradigm
- Paradigms and languages
  - Object-oriented: Ruby
  - Logic programming: Prolog
  - Functional: Haskell
  - Concurrent: Erlang

- We will highlight strengths and weaknesses of each paradigm
- This will be done in a practical way using concrete languages:

> *"Learning to program is like learning to swim. No amount of theory is a substitute for diving into the pool."*
>
> *Joe Armstrong*

Books/Literature

- The main book used for this lecture is
  - Bruce A. Tate: Seven Languages in Seven Weeks, Pragmatic Bookshelf, 2010
- Additional material taken from
  - Maurizio Gabrielli, Simone Martini: Programming Languages: Principles and Paradigms, Springer, 2010 (also available in Italian)
  - Allen B. Tucker, Robert E. Noonan: Programming Languages – Principles and Paradigms (2nd ed.), McGraw-Hill, 2007

Outline

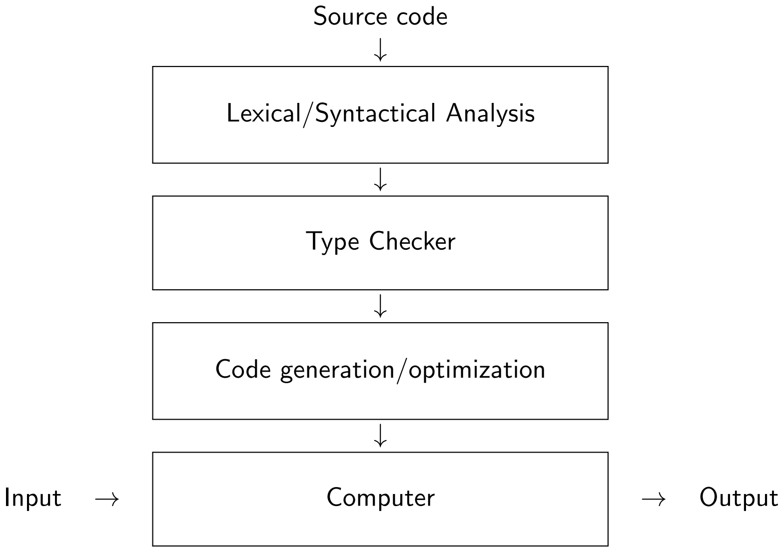
Elements of Programming Languages

- Programming languages have many similarities with natural languages
  - e.g., they conform to rules for syntax and semantics, there are many dialects, etc.
- We are going to have a quick look at the following concepts
  - Compiled/Interpreted
  - Syntax
  - Semantics
  - Typing

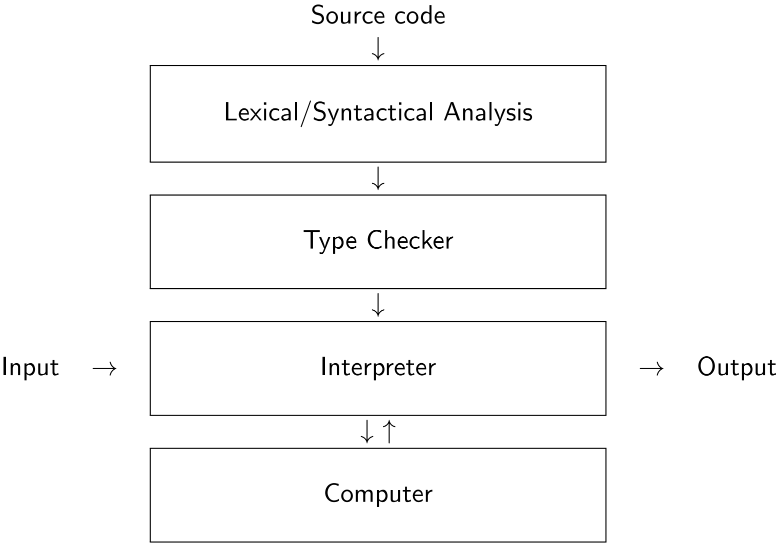
Compiled vs. Interpreted Languages

- Compiled languages are translated into machine code 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



Interpreted Languages



Syntax/1

- 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."

- In contrast, the sentence

  "Colorless green ideas sleep furiously."

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

Syntax/2

- The syntax of a programming language is usually described by a formalism called grammar
- The following very simple grammar recognizes arithmetic expressions

```
<exp> ::= <exp> "+" <exp>
<exp> ::= <exp> "*" <exp>
<exp> ::= "(" <exp> ")"
<exp> ::= "a"
<exp> ::= "b"
<exp> ::= "c"
```

- A program in this language is the product or the sum of 'a', 'b' and 'c'
  - e.g., `a * (b + c)`
- More details on this in the Compiler module

Semantics

- 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 is needed
- 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 behavior 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

- Axiomatic semantics uses logical inference to define a language
- An example is Hoare logic (named after the British computer scientist and logician C. A. R. Hoare)
  - Hoare triple: $\{P\}C\{Q\}$;
    - Describes how the execution of a piece of code changes the state of the computation
    - If precondition P is true, then the execution of command C will lead to postcondition Q
  - Hoare logic provides axioms and inference rules for all constructs of a simple imperative programming language
  - Some examples of rules:
    - An axiomatic rule: $\dfrac{}{\{P\}\ \text{skip}\ \{P\}}$
    - Composition rule: $\dfrac{\{P\}S\{Q\},\{Q\}T\{R\}}{\{P\}S;T\{R\}}$

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 organize 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

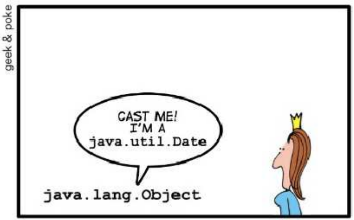
- A language is typed if it specifies for every operation to which data it can be applied
- Languages such as assembly or machine languages can be untyped
  - Assembly language: all data is represented by bitstrings (to which all operations can be applied)
- Languages such as markup or scripting languages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

Strong and Weak Typing

- There is a distinction between weak typing and strong typing
- In strongly typed languages, applying the wrong operation to typed data will raise an error
  - Languages supporting strong typing are also called type-safe
- Weakly typed languages perform implicit type conversion if data do not prefectly match, i.e., one type can be interpreted as another
  - e.g., the string “3.4028E+12” representing a number might be treated as a number
  - May produce unpredictable results

Type Casting

- In some languages it is possible to bypass implicit type conversion done by the compiler
- Type casting is an explicit type conversion defined within a program



- Example of type casting

```
double da = 3.3;
double db = 3.3;
double dc = 3.4;
int result1 = (int)da + (int)db + (int)dc; //result == 9
```

- Implicit type conversion gives a different result (conversion is after addition)

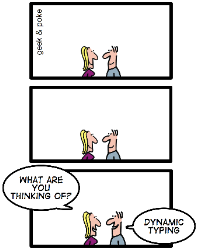
```
int result2 = da + db + dc; //result == 10
```

Static vs. Dynamic Type Checking/1

- 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 Type Checking/2

- 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)



Summary

- Programming languages are artificial languages designed to communicate with computers
    - Provide most powerful human-computer interface
- There are thousands of different languages, which are more or less appropriate for different problems
- Can be classified according to programming paradigms and abstraction level
- There are many similarities to natural languages, e.g., syntax, semantics
    - Syntax determines whether a programm is well-formed
    - Semantic determines the meaning of lanugage concepts/programs, and can be defined in different ways (operational, aximoatic, denotational semantics)
- Type system in a programming language is needed to organize data and helps to check the correctness of programs
- Different forms of type checking, all having pros and cons
    - Weak typing vs. strong typing
    - Static vs. dynamic type checking
    - Type casting