Concepts of programming languages

Lecture 1

Wouter Swierstra

Universiteit Utrecht

Faculty of Science
Information and Computing Sciences

Programming languages

By now, each and every one of you will have experience programming in several different languages

- C# - Imperative programming / Game programming
- Haskell - Functional programming
- SQL - Databases
- Javascript - software project
- PHP & Python - part-time job
- Go - hobby project
- …

Programming languages

What is the programming language that you will spend most of your time developing in?

Programming languages

What is the programming language that you will spend most of your time developing in?

Chances are it hasn't been invented yet!

Programming languages

What is the programming language that you will spend most of your time developing in?

Chances are it hasn't been invented yet!

How much of the technology you are using today was around five years ago?

Teaching programming languages

We want to try and prepare you for tomorrow's programming languages by teaching *concepts* of program design and programming languages - the ideas and principles that can be found across languages over and over again.

Programming languages

Bash shell scripts and C# share many features:

- you can organize code in functions or methods;
- you can iterate using a loop or recursion;
- you can conditionally execute code using if-statements;
- …

Yet these are two different languages are used for very different purposes!

Concepts of programming languages

The aim of this course is to familiarize you with several more advanced concepts of programming languages:

- domain specific languages;
- concurrency and parallelism;
- metaprogramming;
- semantics of programming languages.

We will study several different programming languages (including Erlang, Haskell, and Racket), highlighting how each of these languages addresses these different areas.

Flipping the classroom

Part of the course will be given by you!

There will be presentations on specific languages or language features, including:

1. Template metaprogramming in C++
2. The Elm architecture
3. OCaml's module system
4. References and borrowing in Rust
5. Logic programming in Prolog
6. …

I've put a list of possible topics online. Form a team of 4 or 5 people and claim a topic by opening a pull request listing the names of those students in the schedule.

Projects

In addition to presenting a programming language and its key concepts, you will need to perform a small (research) project in the language of your choice.

1. Choose the language in which you would like to work this week;
2. In the coming three weeks, write a brief project proposal – more instructions online.
3. Submit a written report on what you've done before the end of term.

Question: I want to organize a demo afternoon or poster session, where you can present your work to one another. Would this appeal to you?

Exam

There will be an exam covering all the material covered during the lectures.

- My lectures on programming concepts;
- The presentations given by your fellow students;

Together with any other material and resources that we cover over the coming weeks.

Your final mark is determined by your project and presentation (50%) and exam (50%).

Why study concepts?

Thomas Ball and Benjamin Zorn are two distinguised academics, employed by Microsoft Research.

They wrote a column it the Communications of the ACM in May 2015.

Teach Foundational Language Principles

> *Industry is ready and waiting for more graduates educated in the principles of programming languages.*

Ball and Zorn (Communications of the ACM, 2015)

We should remember that programming languages continuously arise as the need to solve new problems emerges and that it is language principles that are lasting.

Ball and Zorn (Communications of the ACM, 2015)

We should remember that programming languages continuously arise as the need to solve new problems emerges and that it is language principles that are lasting.

As we discuss in this Viewpoint, language foundations serve an increasingly important and necessary role in the design and implementation of complex software systems in use by industry.

Ball and Zorn

The authors put forward several topics they give three central pieces of advice to aspiring computer scientists:

- ‘get a grounding in logic’
- ‘exposed to functional languages as early as possible’
- ‘study type systems in detail’

All three of these points will be touched on in this course (and various other courses in your degree).

When does a programming language exist?

The definition of a programming language

Any programming language definition consists of three parts:

- Syntax
- Static semantics
- Dynamic semantics

Syntax

If you have taken the course on Languages and Compilers, you will have learned that the *syntax* of a language can be described using a EBNF grammar.

The *syntax* specifies what strings of characters constitute valid language fragments…

Syntax

If you have taken the course on Languages and Compilers, you will have learned that the *syntax* of a language can be described using a EBNF grammar.

The *syntax* specifies what strings of characters constitute valid language fragments…

… but does not say anything about what these strings mean.

Brainfuck

We can specify the syntax of a toy programming language like Brainfuck easily enough:

```
C := '>' | '<' | '+' | '-' | '.' | ',' | '[' | ']'
S := C*
```

- `Hello world!` is not a Brainfuck program
- `><><><>+.,.,-` is a Brainfuck program – but we have no idea what it does.

A *parser* reads a string, determines whether it is a in a given language or not, and returns an *abstract syntax tree* representing the structure of our program.

Concrete vs abstract syntax

To illustrate the difference between concrete and abstract syntax, consider the following example in Haskell:

```
data Expr = Val Int | Add Expr Expr

c = "1    + (  2 + 3) /* Sums to six */ "
a = Add (Val 1) (Add (Val 2) (Val 3))
```

The abstract syntax tree lets us focus on the *structure* of our program, without having to worry about the unimportant details (like the number of spaces or comments in the program text).

Beyond syntax

Not every syntactically correct program is sensical.

The following C program is syntactically correct, but not accepted by the C compiler:

```
void main()
{
  x = 7; /* Use of undeclared variable */
  printf("Hello World\n");
}
```

Static semantics

The *static semantics* of a programming language determine which syntactically correct programs are well-formed.

- which variables are declared?
- which methods are defined?
- are all method calls supplied the correct number of arguments?
- are all these arguments of the correct type?

Different languages have a very different notion of ‘well-formed’.

Haskell vs Python

Python interpreter accepts the following definition:

```
def main():
  if True:
    print "hello"
  else:
    5
```

A Haskell interpreter rejects this definition:

```
main =
  if True then print "hello"
  else return 5
```

Which of these two interpreters is correct?

Both are correct

Haskell and Python have a very different design philosophy.

Both interpreters are a correct implementation of the language specification.

The point of this course is to teach you the different approaches and their relative merits.

Dynamic semantics

What is the length of the list `[2 + 0, 2 - 0, 2 * 0, 2/0]`?

Python

```
>>> len [2 + 0, 2 - 0, 2 * 0, 2/0]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ZeroDivisionError: integer division or modulo by zero
```

Haskell

```
> length [2 + 0, 2 - 0, 2 * 0, 2/0]
4
```

Which language is right?

Dynamic semantics

The dynamic semantics specify what the result of running a program.

Once again, different languages can make very different choices.

- Haskell is lazy – it can happily compute the length of the list *before* triggering a division by zero exception;
- Python is not – it evaluates all the elements of the list before computing its length.

Precisely specifying the static and dynamic semantics of even the simplest programming language is not at all easy!

Learning outcomes of this course - I

1. Formulate and design domain specific languages, either as embedded or stand-alone language, while understanding the relative merits of these two approaches.
2. Distinguish between the concepts of concurrency and parallelism and understand the language mechanisms that modern languages use to support both these issues.
3. Understand the concept of metaprogramming and how different languages implement this.

Learning outcomes of this course - II

4. Being able to formulate a simple language's syntax, static semantics, and dynamic semantics and understand the design choices involved.
5. Be able to learn new languages quickly and be able to identify how new languages relate to existing concepts and languages.

More generally

This course aims to study existing programming languages and programming concepts.

By doing so, you will be better positioned to adopt new technology in the future.

Next time

- Be sure to choose a subject and start to form teams.
- We'll start by covering basic concepts and terminology.