x = 1

let x = 1 in ...

x(1).

!x(1)

x.set(1)

Programming Language Theory

Introduction

Ralf Lämmel

≈77min

Motivation

```
constructorDeclaratorRest
    :   formalParameters ('throws' qualifiedNameList)? constructorBody
    ;

constantDeclarator
    :   Identifier constantDeclaratorRest
    ;

variableDeclarators
    :   variableDeclarator (',' variableDeclarator)*
    ;

variableDeclarator
    :   variableDeclaratorId ('=' variableInitializer)?
    ;

constantDeclaratorsRest
    :   constantDeclaratorRest (',' constantDeclarator)*
    ;

constantDeclaratorRest
    :   ('[' ']')* '=' variableInitializer
    ;

variableDeclaratorId
    :   Identifier ('[' ']')*
    ;

variableInitializer
    :   arrayInitializer
    |   expression
    ;

arrayInitializer
    :   '{' (variableInitializer (',' variableInitializer)* (',')? )? '}'
    ;

modifier
    :   annotation
    |   'public'
    |   'protected'
    |   'private'
    |   'static'
    |   'abstract'
    |   'final'
    |   'native'
    |   'synchronized'
    |   'transient'
    |   'volatile'
    |   'strictfp'
    ;

packageOrTypeName
    :   qualifiedName
    ;

enumConstantName
    :   Identifier
    ;

typeName
    :   qualifiedName
    ;
```

The Java grammar (or the C# or Cobol grammars for that matter) has hundreds of productions. How can we possibly abstract from this complexity and understand the essence of an OO programming language (likewise for other paradigms)?

What are the different programming paradigms anyhow? What multi-paradigmatic combinations are there?

What is a reasonable style to define the meaning of language concepts?

- Any way you like?
- Visitor style of OO programming?
- Functional programming?
- ...

What does it really mean to be statically or dynamically typed? How do semantics and type system interrelate?

What heavy lifting can formal treatment of programming languages provide?

Programming language theory = Formal semantics + Programming paradigms

Formal semantics [Wikipedia]:

In theoretical computer science, formal semantics is the field concerned with the **rigorous mathematical study of the meaning of programming languages and models of computation**. The formal semantics of a language is given by a mathematical model that describes the possible computations described by the language.

Formal semantics - why?

- Theory of programming languages is the mathematical study of the meaning of programs.
- The goal is to find ways to describe program behaviors that are both precise and abstract.
  - ***precise*** so that we can use mathematical tools to formalize and check interesting properties, and
  - ***abstract*** so that properties of interest can be discussed clearly, without getting bogged down in low-level details.

This slide is derived from [Odersky]; see resources for the course.

Formal semantics - why?

- ***To develop intuitions for informal reasoning about programs***.
- ***To understand language features and their interactions*** and thereby also develop principles for better language design (PL is the "materials science" of computer science...).
- ***To prove general facts about all the programs in a given programming language*** (e.g., safety or isolation properties).
- **To prove specific properties of particular programs** (i.e., program verification), which is important in some domains (safety-critical systems, hardware design, security protocols, inner loops of key algorithms, ...), but it is still quite difficult and expensive.

This slide is derived from [Nielson]; see resources for the course.

Application areas of formal semantics

- Language understanding
- Language prototyping
- Compiler construction
- Program verification
- Program analysis
- Program optimization
- Program translation
- ...

Classic approaches to formal semantics

- ***Denotational semantics*** and domain theory view programs as simple mathematical objects, abstracting away their flow of control and concentrating on their input-output behavior.
- ***Axiomatic semantics*** is based on program logics such as Hoare logic and dependent type theories focus on logical rules for reasoning about programs.
- ***Operational semantics*** describes program behaviors by means of abstract machines. This approach is somewhat lower-level than the others, but is extremely flexible.
- ***Process calculi*** focus on the communication and synchronization behaviors of complex concurrent systems.
- ***Type systems*** describe approximations of program behaviors, concentrating on the shapes of the values passed between different parts of the program.
- ...

Programming paradigm [Wikipedia]:

A programming paradigm is **a fundamental style of computer programming**. (Compare with a methodology, which is a style of solving specific software engineering problems). Paradigms differ in the concepts and abstractions used to represent the elements of a program (such as **objects, functions, variables, constraints**, etc.) and the steps that compose a computation (**assignation, evaluation, continuations, data flows**, etc.).

(Multi-) paradigms

- **Logic programming**
- **Functional programming**
- **Imperative programming**
- **Object-oriented programming**
- **Constraint-logic programming**
- **Concurrent programming**
- **Parallel programming**
- **...**

Programming paradigms (incl. "multi"-paradigms)

- **Logic programming**
- **Functional programming**
- **Imperative programming**
- **Object-oriented programming**
- **Constraint-logic programming**
- **Concurrent programming**
- **Parallel programming**
- **...**

(Multi-) paradigms in the course

- **Logic programming**
  - ✦ Prerequisite: foundations & primitive, if any, programming skills.
  - ✦ Become proficient as a Prolog programmer.
  - ✦ Use Prolog as a 1st class sandbox for formal semantics.
- **Functional programming**
- **Imperative programming**
- **Object-oriented programming**
- **Constraint-logic programming**
- **Concurrent programming**
- **Parallel programming**

(Multi-) paradigms in the course

- **Logic programming**
- **Functional programming**
  - Become a modest Haskell programmer.
  - Use Haskell as a "2nd class" sandbox for formal semantics.
  - Better understand the pros (and cons) of static typing.
- **Imperative programming**
- **Object-oriented programming**
- **Constraint-logic programming**
- **Concurrent programming**
- **Parallel programming**

(Multi-) paradigms in the course

- **Logic programming**
- **Functional programming**
- **Imperative programming**
  - ✦ Prerequisite: related OO programming skills.
  - ✦ Serves as a basic studying subject for formal semantics.
- **Object-oriented programming**
- **Constraint-logic programming**
- **Concurrent programming**
- **Parallel programming**

(Multi-) paradigms in the course

- **Logic programming**
- **Functional programming**
- **Imperative programming**
- **Object-oriented programming**
  - ✦ Prerequisite: modest OO programming skills.
  - ✦ Serves as a challenging studying subject for formal semantics.
- **Constraint-logic programming**
- **Concurrent programming**
- **Parallel programming**

(Multi-) paradigms in the course

- **Logic programming**
- **Functional programming**
- **Imperative programming**
- **Object-oriented programming**
- **Constraint-logic programming**
  - ✦ Constraints are a powerful programming concept.
  - ✦ CLP is a matured multi-paradigm language with constraints.
  - ✦ Let's study it a bit.
- **Concurrent programming**
- **Parallel programming**

(Multi-) paradigms in the course

- **Logic programming**
- **Functional programming**
- **Imperative programming**
- **Object-oriented programming**
- **Constraint-logic programming**
- **Concurrent programming**
  - ✦ Concurrency important for modern, complex systems.
  - ✦ Understand designated formal semantics: process calculi.
- **Parallel programming**

(Multi-) paradigms in the course

- **Logic programming**
- **Functional programming**
- **Imperative programming**
- **Object-oriented programming**
- **Constraint-logic programming**
- **Concurrent programming**
- **Parallel programming**
  - ✦ Parallelism important for current hardware trends.
  - ✦ We mention parallelism in the context of advanced FP.

Formal semantics: a short introduction

Formal semantics in context

- Theory of programming languages
  - ✦ Formal ***syntax***
    - ⋆ (E)BNF as a description formalism
    - ⋆ Parsing as an execution method of (E)BNF
  - ✦ Formal ***semantics***
    - ⋆ Assign meaning to syntax
      - Interpret syntax in a ***stepwise*** manner
      - Interpret syntax in a ***compositional*** manner

The formal semantics approach

concrete syntax	LL(1)/LALR(1) grammar concrete syntax trees
↓	
abstract syntax	syntactic categories abstract syntax trees
↓	
semantics	semantic categories semantic definitions

Syntactic categories of the *While* language

- numerals
  $n \in$ Num
- variables
  $x \in$ Var
- arithmetic expressions
  $a \in$ Aexp
  $a ::= n \mid x \mid a_1 + a_2 \mid a_1 * a_2 \mid a_1 - a_2$
- booleans expressions
  $b \in$ Bexp
  $b ::= \texttt{true} \mid \texttt{false} \mid a_1 = a_2 \mid a_1 \leq a_2 \mid \neg\, b \mid b_1 \wedge b_2$
- statements
  $S \in$ Stm
  $S ::= x := a \mid \texttt{skip} \mid S_1; S_2 \mid \texttt{if}\ b\ \texttt{then}\ S_1\ \texttt{else}\ S_2 \mid \texttt{while}\ b\ \texttt{do}\ S$

Operational semantics

$y := 1;$
`while` $\neg(x = 1)$ `do` $(y := x * y; x := x - 1)$

First we assign 1 to y, then we test whether x is 1 or not. If it is then we stop and otherwise we update y to be the product of x and the previous value of y and then we decrement x by one. Now we test whether the new value of x is 1 or not $\cdots$

Operational semantics

Operational semantics works with configurations of the form

$$\langle control, data \rangle.$$

Roughly:

control – "where are we", *data* – the values of program variables.

control may be absent (final configuration).

Structural Operational Semantics

Sequences of configurations, $conf_1 \Rightarrow conf_2 \Rightarrow \cdots$.

(Small step semantics.)

Natural Semantics (big step semantics)

$\langle control, data \rangle \rightarrow data'$ in one step.

Denotational semantics

$y := 1;$
`while` $\neg(x = 1)$ `do` $(y := x * y; x := x - 1)$

The program computes a partial function from states to states: the final state will be equal to the initial state except that the value of x will be 1 and the value of y will be equal to the factorial of the value of x in the initial state.

Two kinds of denotational semantics:

- Direct Style Semantics
- Continuation Style Semantics

Axiomatic semantics

$y := 1;$
`while` $\neg(x = 1)$ `do` $(y := x * y; x := x - 1)$

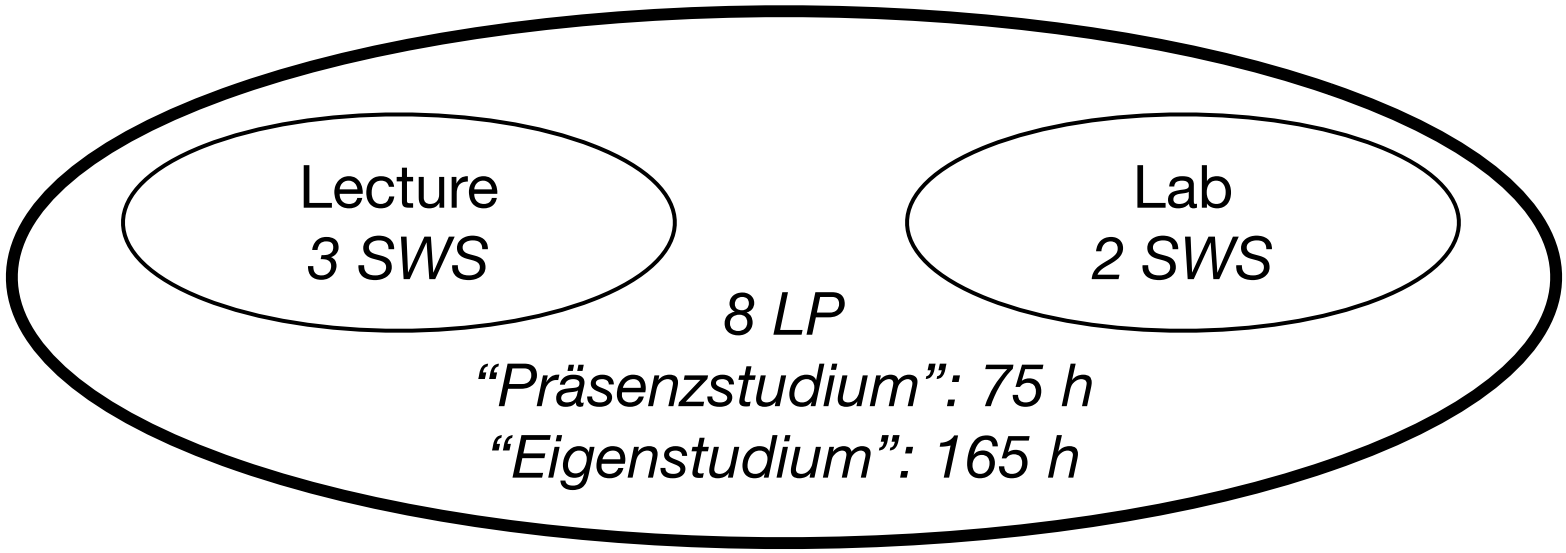
If $x = n$ holds before the program is executed then $y = n!$ will hold when the execution terminates (if it terminates)

Two kinds of axiomatic semantics:

- Partial Correctness
- Total Correctness

Course metadata

Course - size & hours & points



People

- Many motivated, committed, active, and smart **students**
- **Instructor**: Ralf Lämmel
- **Assistant**: Andrei Varanovich

Passing this course

- Exam in 2 parts
  - ✦ Midterm; 40 %; 20 Dec 2011
  - ✦ Final; 60 %; 08 Feb 2012
- Resit (1 part only; possible oral) not scheduled yet
- Student teams of 3 members
  - ✦ Send the following info to dotnetby@gmail.com
    - ★ Your chosen team name
    - ★ Member names and logins (@uni-koblenz.de)
    - ★ Students' register numbers ("MatrikelNr.")
    - ★ Curriculum (Inf/CV/?)

Deadline: 27 Oct EOD

- 8 assignments
  - ✦ Introduced in the lab
  - ✦ Submit solutions to svn by deadline (EOD Sunday)
  - ✦ 2 lab presentations per team *before* midterm
  - ✦ 2 lab presentations per team *after* midterm

Resources for this lecture

General book recommendations for the course.

[Nielson] Book/Slides by Nielson & Nielson: Semantics with applications (1999+)

[Hutton] Graham Hutton: Programming in Haskell, Cambridge University Press, 2007

[Pierce] B.C. Pierce: Types and Programming Languages, MIT Press, 2002

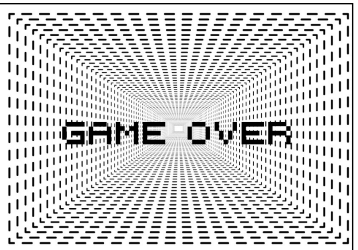
[Odersky] Slides by Martin Odersky (Benjamin Pierce): Foundations of Software (2008+)

[Drabent] Slides by Wlodzimierz Drabent: Programming Theory TDDA43 (2009+)

See the website
for all major resources for the course.

Leveling expectations: What not to expect from this course ...

- The most formal part of formal semantics.
- A classic logic programming course.
- A slow introduction to functional programming.



- **Summary**: *Tough content. Doable exam.*
- **Prepping**:
  - *Start reading Hutton or Pierce or Nielson[2]!*
  - *Re-animate or materialize your Prolog skills.*
- **Lab**: no lab this week yet
- **Outlook**: Prolog crash course