Programming Languages & Paradigms
PROP HT 2011

Lecture 1

Introduction to the course and the subject,
Syntax & Semantics

Beatrice Åkerblom
beatrice@dsv.su.se

Stockholm University

Why Paradigms?

Stockholm University

Languages are tools used for communication, where different languages are more or less efficient for communicating different kinds of situations and solving different kinds of problems.

Stockholm University

"A language that doesn't affect the way you think about programming is not worth knowing"

— Alan Perlis

Stockholm University

The Sapir-Whorf Hypothesis

- Edward Sapir and Benjamin Lee Whorf, two American linguists (early 20th century)
- *"Language influences how we see the world and behave in it"*
- Has not been completely disputed or proven
- Is sometimes used to claim that programmers good at a particular language may not have a deep understanding of some concepts of other languages. Think programming paradigms
- What do you think?

What is a Programming Language?

- A formal language for describing computation?
- A "user interface" to a computer?
- Syntax + semantics?
- Compiler, or interpreter, or translator?
- A tool to support a programming paradigm?

"A programming language is a notational system for describing computation in a machine-readable and human-readable form."

— Louden

Paradigms of Programming?

- There are several ways to think about computation:
  - a set of instructions to be executed
  - a set of expressions to be evaluated
  - a set of rules to be applied
  - a set of objects to be arranged
  - a set of messages to be sent and received

Course PM

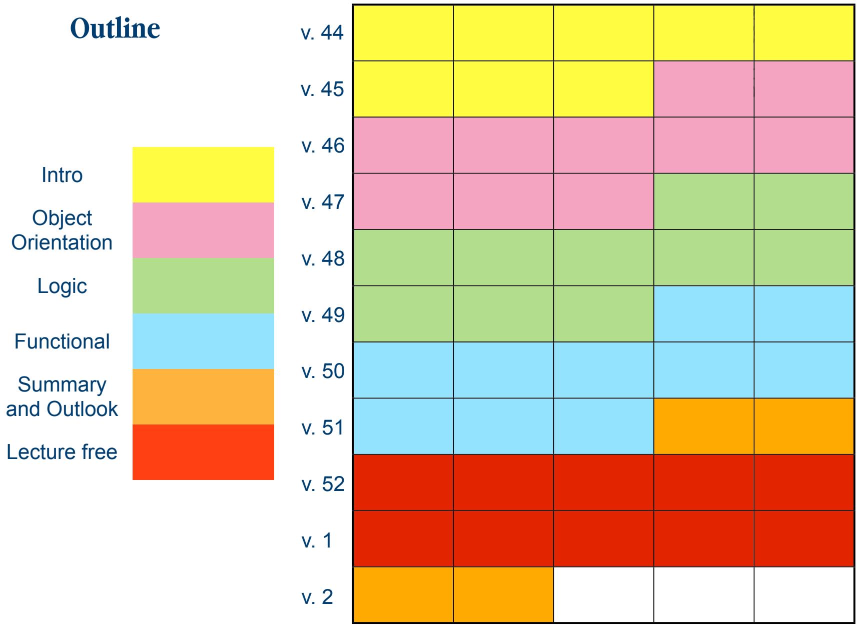
How to Study

- Research shows that these are the **worst** studying techniques:
  - Postponing reading literature to just before exam
  - Postponing doing assignment work to just before exam
- What we mean by **worst**
  - No real understanding (harder in real-life and subsequent courses that build on this knowledge)
  - Easy to forget (harder at the re-exam)
  - You wont pass the course

Our Pedagogic Approach

- Dividing the work up into several, small and manageable bits
- Working together with other people, discussing and reasoning
- What this means for you in practice
  - Several small assignments
  - Reading exercises to keep lectures and reading at same pace
  - Working both in small and very small groups

Outline



How to pass

- Individual tasks:
  - Take-Home exam, 3 hp
- Group tasks:
  - Reading assignment, in total 1,5 hp
  - Programming assignments, in total 3 hp

Stockholm University

Take-Home Exam

- Goal: test your understanding of the theoretical aspects—no programming
- Your answers should be in essay form
- Exam is handed out 2011-12-19, at the end of the seminar and is due back before 2012-01-09, 13:00
- Use the literature, follow the guidelines on the web and **answer everything yourselves—no co-operation is allowed**
- Sample questions can be found on the course web
- Must pass all questions to pass exam
- Let the formal requirements guide your answers

Stockholm University

Assignment 1: Study Group Assignment (1,5hp)

- Goal: keeping your literature studies in synch with the lectures
- You will:
  - divide yourselves into reading groups of four people
  - read and discuss the book
  - submit written answers to questions or solutions for small programming problems related to the literature
  - peer review another groups' answers
- Handed in at the end of each block

Stockholm University

Assignment 2: Programming Assignment (3hp)

- Goal: improve understanding of the theoretical concepts by practical use & have fun
- Done in groups of two
- Divided into four parts, handed in before 2012-01-10, 13:00

Stockholm University

Workload is ***untested***

- Immediate feedback from you – if workload is too high or too low
- Make use of the ***course council***

Outline

Intro
Object Orientation
Logic
Functional
Summary and Outlook
Lecture free

L = Lecture
S = Seminar
T = Tutoring

v. 44	L1			L2	L3
v. 45			S1	L4	L5
v. 46			S2(T)	L6	L7
v. 47			S3	L8	L9
v. 48			S4(T)	L10	L11
v. 49			S5	L12	L13
v. 50			S6(T)	L14	L15
v. 51	S7 Tenta ut		L16	L17	
v. 52					
v. 1					
v. 2	S Tenta in	S			

Lectures

Stockholm University

- During lectures, important content from the literature will be presented and discussed
- Sometimes content taken from other sources will be part of lectures, but there will always be references to the original in the lecture notes
- Lectures and lecture notes can *not* be used instead of the literature
- Lectures will not be recorded
- Lectures may contain discussions or other activities

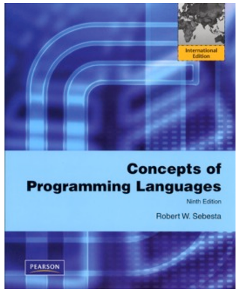
Seminars

Stockholm University

- Seminars contain:
  - presentations of more practical examples
  - introduction to assignments
  - tutoring
- Seminars are both "mini-lectures" and a place to start working on your assignments

Stockholm University

Course Book



- Concepts of Programming Languages by Robert W. Sebesta
- Available from Kårbokhandeln
- ISBN 9780132465588

The Importance of Reading

3 = Detailed understanding
2 = Understand
1 = Overview

1 Preliminaries	1	9 Subprograms	2
2 Evolution...	1	10 Implementing	3
3 Describing...	3	11 Abstract...	3
4 Lexical...	3	12 Support...	3
5 Names...	3	13 Concurrency	2
6 Data types (6.1-6.8)	1	14 Exception ...	2
6 Data types (6.9-6.13)	3	15 Functional	3
7 Expressions...	2	16 Logic	3
8 Statement...	2		

Articles

- Part of course literature for students taking the course at the advanced level (e.g. master students)
- List of articles can be found on the course web page

Code of Honour and Regulations

- All group members are responsible for group assignments
- Recount correctly any help received and sources used
- Do not copy the solutions of others
- Be prepared to present your solution
- Use attendance lists correctly

- **Teachers are obligated to report well founded suspicions of deception to the president and the disciplinary board.**

- Read more here:
  `http://dsv.su.se/en/education/regulations`

Course Council

- Four persons
- Council meetings Wednesdays after the lecture
- Meetings cancelled if not needed
- **The course can be changed while it runs**

Stockholm University

Programming Languages & Paradigms

Stockholm University

Programming Paradigms (Wikipedia)

Top level only...

- Agent-oriented
- Component-based
- Concatenative
- Concurrent computing
- Declarative (contrast: Imperative)
- Event-driven
- Feature-oriented
- Function-level (contrast: Value-level)
- Imperative (contrast: Declarative)
- Iterative (contrast: Recursive)
- Metaprogramming
- Modular
- Nondeterministic
- Parallel computing

Stockholm University

Classifying Languages

- Languages in different categories are fundamentally more alike than they are different
  - We tend to associate things that occur together in some early example of a language category. We tend to believe that these things must always come together
  - Categories are fuzzy. Difficult to decide which languages are or are not in any category
  - Languages frequently belong to more than one category. Sorting them into disjoint classes disguises real similarities among languages with different surface syntax

Stockholm University

“Fundamental” Paradigms

Imperative programming style:	program = algorithms + data *good for decomposition*
Functional programming style:	program = functions ∘ functions *good for reasoning*
Logic programming style:	program = facts + rules *good for searching*
Object-oriented programming style:	program = objects + messages *good for encapsulation*

Issues for all Languages

- Can it be understood by people and processed by machines?
  - although translation may be required
- Sufficient expressive power?
  - can we say what needs to be said, at an appropriate level of abstraction?

Translation

- Compilation
  - Translate into instructions suitable for some other (lower level) machine
  - During execution, that machine maintains program state information
- Interpretation
  - May involve some translation
  - Interpreter maintains program state

Trade-offs

- Compilation
  - lower level machine may be faster, so programs run faster
  - compilation can be expensive
  - examples: C (and Java?)
- Interpretation
  - more ability to perform diagnostics (or changes) at run-time
  - examples: Basic, UNIX shells, Lisp

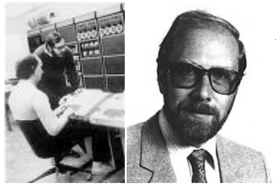
Chronological Classification of PLs

1940s Pre-lingual phase: First computer users wrote machine code by hand.

1950s Exploiting machine power: Early tools; macro assemblers and interpreters First generation optimising compilers. Assembler code, first version of Fortran, Lisp and COBOL.

Stockholm University

1960s Second generation tools -- optimising compilers, inspections. First really big projects. Commercial mainframes, software for big business. Increasing expressive power: Cobol, Lisp, Algol60, Basic, PL/1 ---but most "proper" programming still done in assembly language.



1970s Fighting the "software crisis" C, Prolog, Pascal, Algol68: Reducing machine dependency, Increasing program correctness - Structured Programming, modular programming and information hiding. Collaborative software tools; Unix, code repositories, make, etc. Minicomputers and small business software

Stockholm University

1980s: Personal computers and workstations. Emphasis on processes. The rise of consumer software. Reducing complexity Smalltalk, C++, Ada, Eiffel

1990s: Object-oriented programming and agile processes. Internet programming and software everywhere, parallel and distributed hardware, dynamic PLs Perl, Java

Stockholm University

Groups, Assignments, Submissions, etc.

Stockholm University

Isak and Tobias

Syntax & Semantics

Syntax vs. Semantics

```
def print_message():
   print("This is a message")

print_message()
```

```
def print_message
  p "This is a message"
end

print_message
```


Syntax vs. Semantics

```
def print_message():
   print("This is a message")

print_message()
```

```
def print_message
  p "This is a message"
end

print_message
```

```
class Print
{
    public static void main(String[] args)
    {
        printMessage();
    }
    static void printMessage()
    {
        System.out.println("This is my message");
    }
}
```


Syntax from two angles

- Specify structural rules for programing languages
  - for programmers, to make it possible for them to write valid programs
  - regular expressions, context-free grammars
- Figure out if a given program follows some given syntax rules
  - for compilers
  - scanners and parsers

Describing syntax

- Regular language
  - Concatenation
  - Alternation
  - “Kleene closures”

```
/hello/   /hello|goodbye/   /[hello]*/
```

- Context-free language
  - All of the above
  - Recursion

```
<Program>   ::= begin <stmt_list> end
<stmt_list> ::= <stmt> |
                <stmt> ; <stmt_list>
<stmt>      ::= <var> = <expr>
<var>       ::= A | B | C
<expr>      ::= <var> + <var> |
                <var> - <var> |
                <var>
```


Checking syntax

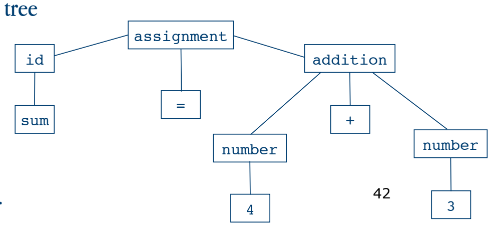
- For a given a string from some language, we can build a parse tree to represent its syntactic structure. This is typically done in two steps:

```
sum = 4 + 3;
```

  - A scanner chops the string up in tokens:

token type	id	assignment	number	addition	number
lexeme	sum	=	4	+	3

  - A parser builds a the parse tree



This is often enough, but...

Semantics

- The semantics of a string is directly connected to the parse tree, or we have the wrong grammar

Static and Dynamic Semantics

- Static semantics is used to describe properties that syntactically valid programs also must have to be semantically valid, e.g. that they are type correct
  - really more related to legal forms of programs rather than meaning
  - some cannot be described by BNF, some just very verbose
  - attribute grammars

- Dynamic semantics is used to describe how the meaning of valid programs should be interpreted

Static Semantics

- To each rule in the grammar we add a semantic clause
  - relating the semantics of the members of the right-hand side of the rule to the semantics of the entire rule
  - relating the semantics of the members of the entire rule to the semantics of the right-hand side of the rule
- Semantic information flowing down is called inherited: each rule inherits it from its parent in the tree
- Semantic information flowing up is called derived: each rule derives it from its children

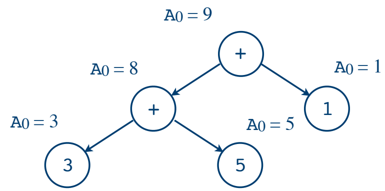
Attribute Grammars

```
Grammar:
SumS -> Digit
Sum -> Sum+Digit
Digit -> 0|1|...|9

Attribute grammar:
1.  SumS -> Digit         {A0:=A1}
2.  Sum -> Sum + Digit    {A0:=A1+A3}
3a. Digit -> 0            {A0:=0}
    ...                   ...
3j. Digit -> 9            {A0:=9}
```

```
String:
3+5+1 (semantics 9)

Attribute tree:
```



Dynamic Semantics

- Operational semantics
  - The meaning of a statement defined by describing the effect of running it on a machine

$a ::= n \mid X \mid a_0 + a_1 \mid a_0 - a_1 \mid a_0 * a_1$

$$\frac{\langle a_0, \sigma \rangle \rightarrow n_0 \quad \langle a_1, \sigma \rangle \rightarrow n_1}{\langle a_0 + a_1, \sigma \rangle \rightarrow n}$$

where n is the sum of n_0 and n_1

Dynamic Semantics, cont'd

- Denotational semantics
  - Mathematical denotation of the meaning of the program (typically, a function)
  - Facilitates reasoning about the program, but not always easy to find suitable semantic domains

Dynamic Semantics, cont'd

Stockholm University

- Axiomatic semantics
  - Program as a set of properties
  - good for proving theorems about programs, but somewhat distant from implementation

The End

Stockholm University

References

- Sebesta, R - Concepts of Programming Languages
- Louden, K - Programming Languages: Principle and Practice
- Scott, M - Programming Language Pragmatics
- Grune, D and Jacobs, C - Parsing Techniques, a Practical Guide
- Winskel, G - The Formal Semantics of Programming Languages