Lund University
Computer Science
Lennart Andersson

Course description
EDA145/DATN19
2009-03-16

Programming language theory

Introduction

Which value will the variable `x` have after the execution of these statements?

```
x = 1;
y = 2;
```

If the programming language is Java the answer is 1 (unless there are multiple threads using the same variables). If it is part of a C, C++ or Fortran program then there are two answers: The value is 1 or 2.

If you are familiar with some of those languages you will be able to construct a context where the latter alternative is true. Similar behavior will occur with Java when using objects. Define

```
class IntVar {
  int value;
  void assign(int value) {
    this.value = value;
  }
}
```

and suppose that `x` and `y` are instances of `IntVar` in

```
x.assign(1);
y.assign(2);
```

Here it is not certain that `x` has the value 1. This is a question of the *semantics* of the programming language, i.e. what the statements mean or accomplish.

Are languages with such semantics good?

Programming language theory

Courses in compiler construction are often very formal about describing the syntax of languages using regular expressions and context-free grammars but very informal when it comes to semantics. We will not repeat these parts in the current course, but start off where the compiler course becomes informal. This does not mean that a compiler construction course is a required prerequisite for this course. In fact, they can be read in any order. The interface between them is just an object-oriented model of a program that can be understood after a data structures course.

So the course is about formal semantics, i.e. different ways to describe the meaning of programming language constructs. Constructs with simple semantics are presumably easy to understand and use, while those with complicated semantics are error prone. The formal semantics for the

goto statement present in some languages is quite complicated. It is well known that undisciplined use of goto statements may lead to erroneous programs that are hard correct. Formal semantics provides an objective way to judge programming languages.

In introductory programming courses, semantics is described by explaining in what order the statements are executed and how they change the contents of the memory. This can be formalized and is called *operational* semantics.

The semantics for a programming language can also be given as a logical *theory* in the same way as mathematics with *axioms*, *theorems*, and *inference rules*. Proofs can be made so formal that a simple program can check them. In this context, axioms describe the properties of simple statements, and the inference rules describes the meaning of composed statements like an if-statement. This kind of semantics is called *axiomatic*. The axiomatic semantics emphasizes the task of the programmer; to construct a program that fulfills its specification.

Denotational semantics is more abstract. Every program construct is mapped onto a mathematical object, its *denotation*. The denotations are often mathematical functions. It may be convenient to define these functions using a functional programming language rather than with conventional mathematical notation. This will give an interpreter for language for free.

There are several reasons why a student in Computer science should know formal semantics:

- It provides exact definitions of the meaning of programming constructs which informal descriptions seldom do.
- It makes programming into a science rather than an engineering task or an art.
- It allows strict mathematical reasoning about programs.
- It gives new understandings of what a program is, different from understandings based on implementations on conventional computers.
- It is a support for program language designer by clearly distinguishing between simple and difficult constructs or between good and bad programming languages.
- It supports the programmer in constructing correct programs. A program constructed in a way that makes correctness proofs possible in principle are presumably better than programs that are difficult or impossible to prove correct.

The course will also introduce you to the abstractions and formalisms used throughout theoretical computer science, not just semantics. You will be able to read and understand many more scientific articles than prior to the course.

Lambda calculus

Conventional languages like C and Java, have complicated semantics. It is hard to explain the semantics to the novice without referring to how data is moved in the computer memory, how parameters are transferred, how inheritance is implemented, etc.

There are other kinds of programming languages that can be understood without knowing how a computer works. A very simple one is lambda calculus which is the basis for all functional programming languages. The lambda calculus has just three constructs, *identifiers*, *abstractions* that are simple function definitions, and *applications* where an abstraction is applied to one construct. This is all there is. There are no predefined data types or other operations apart from applications. Still, all computations that can be described using a conventional programming language can be implemented in lambda calculus.

Recursive definitions and domain theory

Both in programs and denotational semantics we use recursive definitions of functions and data types. Domain theory is about the meaning of such definitions. Defining something using the defined concept recursively may be problematic. Consider

Theorem 1 *Theorem 1 is false.*

Is Theorem 1 true or false? Both alternatives lead to contradictions. The problem with the theorem is that it refers to itself. Is it meaningful? And if so, what is the meaning? The domain theory gives semantics of recursive definitions.

Course prerequisites

The course belongs to the advanced level according to the Bologna classification. The formal requirements are a course in algorithms and data structures and some mathematics courses. What makes it advanced is all the abstractions. You should be interested in abstractions and logical reasoning to enjoy it.

Seminars

For every seminar there will be a set of problems which the students are supposed to solve at home and present at the seminar. The solutions should be written down, but they will usually not be collected nor checked. At the start of the seminar each student shall mark on a list all problems which he/she is willing to present a solution to at the blackboard. The marks will give a bonus on the exam in May 2007. A maximal bonus will increase the exam score by 20% of the maximal score.

Students are encouraged to cooperate when solving the problems. The bonus will not be reduced for erratic solutions. However marking a problem which the student hasn't solved completely in his/her mind is not acceptable.

The seminar problems will appear on the course home page.

Programming assignments

This year I will try to provide a number of voluntary programming assignments on a weekly basis. For those who complete the assignments there will either be a special exam or a bonus system. The details has not been settled yet. The regular exam will not presuppose that you have carried out the assignments.

You may use any program language for the implementations, but you are supposed to manage on your own. I believe that a functional programming language such as Haskell is best suited for the purpose, but any high level language with good data abstraction facilities should be fine.

Examination

There will be written exam in June. The text book and the lecture notes may be consulted during the exam.