

# Semantyka i weryfikacja programów

**Andrzej Tarlecki,  
Aleksy Schubert, Michał Skrzypczak**

Instytut Informatyki  
Wydział Matematyki, Informatyki i Mechaniki  
Uniwersytet Warszawski

<http://www.mimuw.edu.pl/~{tarlecki, alx, mskrzypczak}>  
[{tarlecki, alx, mskrzypczak}@mimuw.edu.pl](mailto:{tarlecki, alx, mskrzypczak}@mimuw.edu.pl)

Strona tego wykładu:

<http://www.mimuw.edu.pl/~tarlecki/teaching/semwer/>

# Program Semantics & Verification

**Andrzej Tarlecki,  
Aleksy Schubert, Michał Skrzypczak**

Institute of Informatics  
Faculty of Mathematics, Informatics and Mechanics  
University of Warsaw

<http://www.mimuw.edu.pl/~{tarlecki, alx, mskrzypczak}>  
[{tarlecki, alx, mskrzypczak}@mimuw.edu.pl](mailto:{tarlecki, alx, mskrzypczak}@mimuw.edu.pl)

This course:

<http://www.mimuw.edu.pl/~tarlecki/teaching/semwer/>

## Overall

- The aim of the course is to present the importance as well as basic problems, techniques and applications of formal description of programs.
- Various methods of defining program semantics are discussed, and their mathematical foundations as well as techniques are presented, with applications.
- The basic notions of program correctness are introduced together with methods, formalisms, and support tools for their derivation.
- The ideas of systematic development of correct programs are introduced.

2024/25:

## New course format

*lectures + laboratory classes*

### Lectures

To present various methods of defining program semantics, with their mathematical foundations and techniques, and notions of program correctness with their associated proof techniques.

### Lab classes

To illustrate the applications of such methods in programming language design, in programming and in program verification, with the use of available support tools.

## Lectures

- Formal description of programming languages
- Operational and denotational semantics of programming languages
- Semantics of typical programming constructs
- Partial and total correctness of programs, and their corresponding proof methods
- Hoare's logic and its formal properties
- Mathematical foundations of denotational semantics
- Basic concepts of universal algebra and their role in program semantics and development

## Laboratory

- *Practical* use of such methods in programming language design and implementation, and in program verification

*An opportunity to design and implement your own programming language*

- *Projects* (small groups):

*Your own programming language – overall idea and informal description*



*defining its semantics*



*implementing its interpreter*

- Program *verification* with the use of available support tools

## Prerequisites

- Wstęp do programowania (1000-211bWPI)
- Podstawy matematyki (1000-211bPM)
- Języki, automaty i obliczenia (1000-214bJAO)

## Literature

Rather random choice:

- M. Hennessy. *The Semantics of Programming Languages: An Elementary Introduction Using Structural Operational Semantics*. Wiley, 1990.
- M. Fernandez. *Programming Languages and Operational Semantics: A Consise Overview*. Springer, 2004.
- H. Riis Nielson, F. Nielson. *Semantics with Applications: An Appetizer*. Springer, 2007.
- M. Gordon. *Denotacyjny opis języków programowania*. WNT, 1983.
- D. Gries. *The Science of Programming*. Springer-Verlag, 1981.
- E. Dijkstra. *Umiejętność programowania*. WNT, 1978.
- K. R. Apt. *Ten Years of Hoare's Logic: A Survey – Part 1*, *ACM Toplas* 3(4), 1981, 431-483.
- A. Blikle, P. Chrzastowski-Wachtel. *Denotational Engineering of Programming Languages*. In preparation, 2021.



# Programs

```
D207 0C78 F0CE 00078 010D0      r := 0; q := 1;
D203 0048 F0D6 00048 01CD8      while q <= n do
8000 F0EA F0B3 010EC 00ED7        begin r := r + 1;
9C00 000C F0DA 0000C ...          q := q + 2 * r + 1 end
```

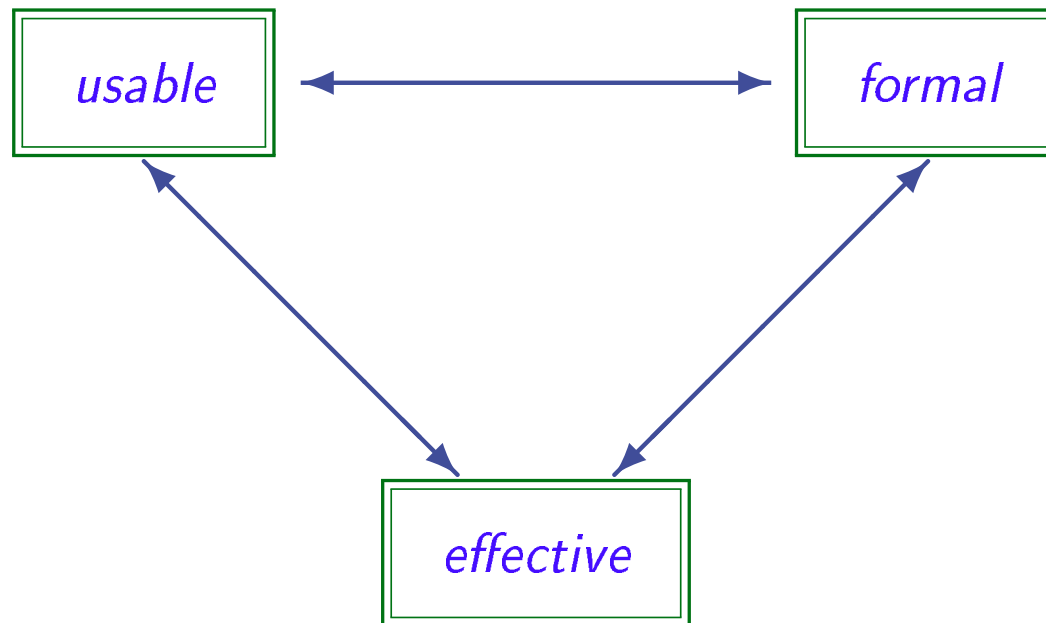
- a precise description of an *algorithm*, understandable for a human reader
- a precise prescription of *computations* to be performed by a computer

Programs should be:

- clear; efficient; robust; reliable; user friendly; well documented; ...
- but first of all, **CORRECT**
- don't forget though: also, *executable*...

# Programming languages

## Tensions



## Grand View

What we need for a good programming language:

- Syntax
- Semantics
- Logic
- Pragmatics/methodology
- Implementation
- Programming environment

# Syntax

To determine exactly the well-formed phrases of the language.

- *concrete syntax* (lexical analysis; often LL(k), LR(1), ...)
- *abstract syntax* (CF grammar, BNF notation, etc)
- *type checking* (context conditions, static analysis)

*It is standard by now to present it formally!*

One consequence is that excellent tools to support parsing are available.

# Syntax

There are standard ways to define a syntax for programming languages. The course to learn about this:

*Języki, automaty i obliczenia*

Basic concepts:

- *formal languages*
- (generative) *grammars*: regular (somewhat too weak), *context-free* (just about right), context-dependent (too powerful), ...

**BTW:** there are grammar-based mechanisms to define the semantics of programming languages: attribute grammars, perhaps also two-level grammars, see (or rather, go to)

*Metody realizacji języków programowania*

## Concrete syntax

*Concrete syntax* of a programming language is typically given by a (context-free) grammar detailing all the “commas and semicolons” that are necessary to write a string of characters that is a well-formed program.

$$rt := rt + 12$$

vs.

$$rt := rt + 1\%$$

vs.

$$\cancel{rt + 12} := rt$$

Typically, additional context-dependent conditions eliminate some of the strings permitted by the grammar (like “thou shalt not use an undeclared variable”).

Presenting a formal language syntax by an unambiguous context-free grammar gives a *structure* to the strings of the language: it shows how a well-formed string is build of its immediate components using some linguistic *construct* of the language.

## Abstract syntax

*Abstract syntax* presents the structure of the program phrases in terms of the linguistic constructs of the language, by indicating the *immediate components* of the phrase and the *construct* used to build it.

Think of abstract syntax as presenting each phrase of a language as a tree: the node is labelled by the top construct, the subtrees give the immediate components.

*Parsing* is the way to map concrete syntax to abstract syntax, by building the abstract syntax tree for each phrase of the language as defined by the concrete syntax.

$rt := rt + 1$

*parsing*

$ASSIGN(VarID(rt), SUM(VarID(rt), IntLiteral(12)))$

## At this course

We will not belabour the distinction between concrete and abstract syntax.

- concrete-like way of presenting the syntax will be used
- the phrases will be used as if they were given by an abstract syntax
- if doubts arise, parenthesis and indentation will be used to disambiguate the interpretation of a phrase as an abstract-syntax tree

*This is inappropriate for true programming languages  
but quite adequate to deal with our examples*

*This is not sufficient for your lab projects*



## Pragmatics

To indicate how to use the language well, to build *good* programs.

- user-oriented presentation of programming constructs
- hints on good/bad style of their use
  - intended application domains
  - programming patterns
  - naming conventions
  - modularisation techniques
  - ...

# Logic

To express and prove program properties.

- Partial correctness properties, based on first-order logic
- Hoare's logic to prove them
- Termination properties (total correctness)

Also:

- temporal logics
- other modal logics
- algebraic specifications
- abstract model specifications
- ...

*Other properties, e.g.:*

*interactive (infinite) behaviours, safety,  
use of resources, complexity, ...*

*Other verification methods:*

*proof systems, testing, model checking, ...*

*program verification*

vs.

*correct program development*

## Methodology

- specifications
- stepwise refinement
- designing the modular structure of the program
- coding individual modules

### Code development and maintenance

- various development styles (agile, eXtreme, ...)
- code refactoring ...

## Implementation

Compiler/interpreter, with:

- parsing
- static analysis and optimisations
- code generation

## Programming environment

So that we can actually do this:

- dedicated text/program editor
- compiler/interpreter
- code/module library
- version control system
- test bed
- debugger

BUT ALSO:

- support for
  - specification development
  - verification
  - architectural design
  - ...

# Semantics

To determine the meaning of the programs and all the phrases of the language.

*Informal description is often not good enough*

- **operational semantics** (small-step, big-step, machine-oriented): dealing with the notion of *computation*, thus indicating *how* the results are obtained
- **denotational semantics** (direct-style, continuation-style): dealing with the overall *meaning* of the language constructs, thus indicating the results without going into the details of how they are obtained
- **axiomatic semantics**: centred around the *properties* of the language constructs, perhaps ignoring some aspects of their meanings and the overall results

## Why formal semantics?

So that we can sleep at night. . .

- precise understanding of all language *constructs* and the underlying *concepts*
- independence of any particular implementation
- easy prototype implementations
- necessary basis for trustworthy reasoning, verification and optimisation

## Example 1

- Naive optimisation: replace

**if**  $f(x)$  **then**  $x := 555$  **else**  $x := 555$

by

$x := 555$

Are these two statements *equivalent*?

- Not-so-naive optimisation: replace

$x := 555; x := 555; x := 555$

by

$x := 555$

Are these two statements equivalent?

## Example 2

Recall:

```
r := 0; q := 1;
while q ≤ n do
  begin r := r + 1;
        q := q + 2 * r + 1
  end
```

Or better:

```
rt := 0; sqr := 1;
while sqr ≤ n do (rt := rt + 1;
                    sqr := sqr + 2 * rt + 1)
```



Well, this computes the integer square root of (nonnegative integer)  $n$ , doesn't it:

$\{n \geq 0\}$

$rt := 0; sqr := 1;$

$\{n \geq 0 \wedge rt = 0 \wedge sqr = 1\}$

**while**  $\{sqr = (rt + 1)^2 \wedge rt^2 \leq n\}$   $sqr \leq n$  **do**

$(rt := rt + 1;$

$\{sqr = rt^2 \wedge sqr \leq n\}$

$sqr := sqr + 2 * rt + 1)$

$\{rt^2 \leq n < (rt + 1)^2\}$

But how do we justify the implicit use of assertions and proof rules?

## Sample proof rule

For instance:

$$\{sqr = rt^2 \wedge sqr \leq n\} \text{ } sqr := sqr + 2 * rt + 1 \text{ } \{sqr = (rt + 1)^2 \wedge rt^2 \leq n\}$$

follows by:

$$\{\varphi[E/x]\} x := E \{\varphi\}$$

BUT: although correct *in principle*, this rule fails in quite a few ways for PASCAL (abnormal termination, looping, references and sharing, side effects, assignments to array components, etc)

*Be formal and precise!*

## Justification

- definition of program semantics
- definition of satisfaction for correctness statements
- proof rules for correctness statements
- proof of soundness of all the rules
- analysis of completeness of the system of rules