Semantics of programming languages

William Steingartner

william.steingartner@tuke.sk

Technical University of Košice, Slovakia
Department of Computers and Informatics,
Faculty of Electrical Engineering and Informatics

Introduction to Formal Semantics

Sets

A **set** A is a well defined collection of distinct objects. For example:

$$A = \{a_1, a_2, \ldots, a_n\}, \qquad B = \{b \mid P(b)\}.$$

Conventions:

- $\mathbb{N}$ set of natural numbers,
- $\mathbb{N}_0$ set of natural numbers with value 0,
- $\mathbb{Z}$ set of integers.

Basic operations:

$A \cup B = \{c \mid (c \in A) \vee (c \in B)\}$	union,
$A \cap B = \{c \mid (c \in A) \wedge (c \in B)\}$	intersection,
$A \times B = \{(a, b) \mid (a \in A) \wedge (b \in B)\}$	binary Cartesian product.

Relations

A **binary relation** R between arbitrary sets A (domain, the set of departure) and B (codomain, the set of destination) is a subset of the Cartesian product

$$R \subseteq A \times B.$$

The statement $a \in A$ is R-related to $b \in B$ and is denoted by aRb.

The relation $R \subseteq A \times A$ is called **preorder** on A, if

- $a_1 R a_1$ (reflexivity),
- if $a_1 R a_2$ and $a_2 R a_3$ then $a_1 R a_3$ (transitivity),

for all $a_1, a_2, a_3 \in A$.
The relation $R \subseteq A \times A$ over a set A which is reflexive, antisymmetric, and transitive, i.e., which satisfies for all $a_1, a_2, a_3 \in A$:

- $a_1 R a_1$ (reflexivity),
- if $a_1 R a_2$ and $a_2 R a_1$, then $a_1 = a_2$ (antisymmetry),
- if $a_1 R a_2$ and $a_2 R a_3$, then $a_1 R a_3$ (transitivity),

is called **partially ordered set** – a poset.

Functions

A **total function** from A to B

$$f : A \to B$$

is a function which is defined for all inputs of its domain. It is a relation in which for each $a \in A$ there exists **exactly one** $b \in B$ such that afb, denoted also by:

$$f(a) = b.$$

A is called **domain**, B is called **codomain** of the function f.

A function f is a relationship that assigns exactly one output $b \in B$ value for each input value $a \in A$, which can be denoted also

$$f : a \mapsto b$$

Partially defined function is not defined for all values of the domain. It is denoted by:

$$f : A \rightharpoonup B.$$

There can exits such $a \in A$, for which $f(a) = \bot$.

Composition, Currying

Composition of functions (or composite function) refers to the combining of functions in a manner where the output from one function becomes the input for the next function. Given two functions $f : A \to B$, $g : B \to C$, the composite function is a function from A to C:

$$g \circ f : A \to C \quad \text{such that for each } a \in A,$$
$$(g \circ f)(a) = g(f(a)).$$

Currying of functions. A function:

$$f : A \times B \to C$$

can be denoted also by:

$$f : A \to B \to C.$$

Firstly, f is applied to an argument $a \in A$, the result is a **function**:

$$f(a) : B \to C.$$

In the second step, the function $f(a)$ is applied to an argument $b \in B$ and the result is a value:

$$f(a)(b) = f(a, b) \in C.$$

Currying functions is **right associative** implicitly.

Semantics of programs

Formal semantics provides unambiguous meaning of programs written in programming languages.

Programming languages include various kinds of **constructs**: expressions, declarations, commands, etc.

For example:

- **imperative languages** have declarations, types, blocks, statements, formal arguments, . . .

The differences between languages are not merely syntactic: they reflect essential differences in the **computational meaning** of the various kinds of construct.

Each construct may involve both:

- **flow of control** and
- **flow of information**.

Flow of control and Flow of information

An **execution** of a program consists of a computation on some machine.

In general, the **computation** may be regarded as a combination of sub-computations for the constructs that occur in the program.

A **control**:

- **flows into** construct when its sub-computation starts, and
- **flows out** again when it finishes.

Information processing during the computation corresponds to flow of information between constructs.

Constructs can differ mostly in how they let

- control and
- information

flow to and from their sub-constructs.

Flow of control

Fundamental control-flow concepts include sequencing, interleaving, choice between alternatives, exception raising and handling, iteration, procedural abstraction and activation, concurrent processes, and synchronization.
The main features of the above control-flow concepts are as follows:

- **sequencing** involves letting control flow into sub-constructs from left to right, so long as each terminates normally,
- **interleaving** lets control move back and forth between two or more sub-constructs,
- **choice** between several alternatives usually lets control flow into only one of the alternatives, although when the chosen alternative fails, this may cause back-tracking, whereby another alternative is tried,
- **raising** or **throwing** an exception generally interrupts normal control flow (sequencing, etc.) and skips the rest of enclosing constructs until control reaches some enclosing construct that can handle or catch the exception, thereby resuming normal control flow,
- **iteration** lets control flow into the same sub-construct repeatedly,
- activation of a **procedural abstraction** lets control flow into a construct from different parts of the program, resuming from the point of activation when the activation terminates,
- **concurrent processes** – split the flow of control into separate threads, which may subsequently be subject to synchronization.

Flow of information

Fundamental information-flow concepts include computation of values, use of computed values, effects on storage, scopes of bindings, and message-passing.

- Obviously, **expressions** are **evaluated** primarily to compute values, a declaration computes a so-called environment (i.e., a set of bindings for identifiers); and a command, which conceptually computes no value at all, can be regarded as computing a fixed dummy value.
- **Computed values** correspond to information that flows out of constructs; the values may subsequently flow into other constructs. Computing a value always involves termination. Raising an exception may be seen as a combination of exceptional control flow and the computation of a value that identifies the exception.
- **Effects on storage** are stable: when a new cell has been added to the storage, or a value stored in a cell (overwriting any previous value), the cell remains in the same state until some new value is stored in it, or the cell is removed (explicitly, or by garbage collection).
- The **bindings computed** by one sub-construct may flow (often together with the current bindings) into another sub-construct, which is then the scope of the computed bindings.
- **Message-passing** by a construct corresponds to information flow both ways: the construct itself gets the information that its context accepts the message, and the context gets the information in the message itself.

Programming Paradigms

Programming languages can be classified according to which programming paradigm they support best.

Sometimes, a language intended for one paradigm can be used for other paradigms as well, with some extra effort.

Some widely-used paradigms:

- imperative programming,
- functional programming,
- concurrent programming,
- object-oriented programming,
- logic programming.

Imperative programming emphasizes sequencing, iteration, procedural abstraction and activation, and effects on storage.

Imperative programs:

- tend to involve numerous changes of stored values,
- the computations of the values themselves are usually rather simple,
- flow of control is expressed primarily by sequential, conditional, and iterative commands.

Typical languages: C, Pascal, Algol60 and many others.

Overview of other paradigms

- Functional programming emphasizes computation of values, scopes of bindings, and procedural abstraction and activation. Flow of control is usually expressed by activation of functions (i.e., procedural abstractions that compute values from argument values) giving them other functions as arguments. The scopes of bindings in declarations are often recursive.
- Concurrent programming emphasizes concurrent processes, choice between alternatives, and message-passing. Each process generally does a very limited amount of computation in between sending or receiving messages. Typically, choices between alternatives are decided by the kind of message next received, or by synchronization between two processes.

Overview of other paradigms

- Object-oriented programming may be regarded as imperative programming with a particular discipline for scopes of bindings. It is based on the use of bindings between objects and methods. Objects are essentially identifiable fragments of storage, created either by instantiating classes or by cloning existing objects. Methods are procedural abstractions that usually have direct effects on the objects to which they are bound.
- Logic programming emphasizes choice between alternatives with back-tracking, and procedural abstraction. The basic idea is that a logic program consists of a set of alternative procedure abstractions for each identifier. The choice between them is made by so-called unification of patterns between the parameters of the abstractions and the arguments of the activations. This unification generally binds identifiers to terms, and the bindings revert when a failure causes back-tracking.

Formal Language Descriptions

Descriptions are called **formal** when it is written in a notation that already has a precise meaning.

- Reference manuals and standards for programming languages generally provide some formal descriptions of program syntax.
- The description in the reference manual is generally completely informal, being expressed only in natural language which, even when used very pedantically, is inherently imprecise and open to misinterpretation.

We shall next consider how to express the computational meaning of constructs precisely, using **formal semantics**.

Formal definition of language

Formal definition of programming language is given by:

- formal definition of syntax:
    - Backus-Naur form,
    - inductive definition,
    - derivation rules, . . .
- formal definition of semantics by appropriate semantic method.

Syntax

Syntax of programming language determines the form and structure of programs written in given language.

Kinds of syntax:

- *concrete syntax* – deals with text and parsing,
- *abstract syntax* – deals only with structure,
- *context-free syntax* – assumes fixed grouping rules,
- *context-sensitive syntax* – can also deal with constraints.

From the point of formal definition of programming language, only **concrete** and **abstract syntax** are taken.

Concrete syntax

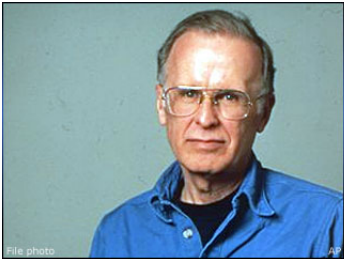
Concrete syntax deals with program source and parsing.

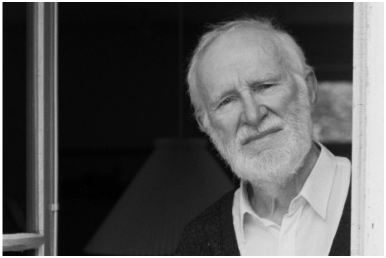
Concrete syntax:

- determines which text strings are accepted as programs and program constructs,
- provides a **parse tree** for each accepted program,
- is typically specified by **formal grammars**, with productions giving sets of alternatives for each non-terminal symbol,
- a grammar for concrete syntax should be unambiguous,
- an alternative approach is to let the grammar remain ambiguous, and provide precedence rules to select a unique parse tree for each program text.

Concrete syntax

Backus-Naur Form (BNF) – formal notation technique for context-free grammars, often used to describe syntax of languages used in computing.





John Backus Peter Naur

Abstract syntax

Abstract syntax deals only with a **structure** of programs and serves for formal semantics.

Whereas concrete syntax deals with the actual character strings used to write programs, abstract syntax is concerned only with the *deep structure* of programs, which is generally represented by trees – **abstract syntax trees** (AST).

The definition of abstract syntax consists of:

- syntactic domains, and
- abstract syntax tree.

In abstract syntax trees:

- each node is created by a particular constructor,
- leaves can be created by constant constructors with no arguments,
- each node has a branch for each argument of its constructor.

AST can be expressed also in linear form by using the rules in BNF.

For purpose of this course, we will use the following definition of abstract syntax.

Abstract syntax consists of:

- syntactic domains,
- one production rule for each syntactic domain.

  Every production rule has a form:

$$element\_of\_syntactic\_domain ::= form_1 \mid form_2 \mid \ldots$$

Example of abstract syntax

Language of untyped arithmetic expressions has abstract syntax defined as follows:

1. Syntactic domains:

$x \in \mathbf{Var}$	variables,
$n \in \mathbf{Num}$	strings of digits,
$e \in \mathbf{Expr}$	arithmetic expressions.

   The elements $n \in \mathbf{Num}$ and $x \in \mathbf{Var}$ have no internal structure from the semantic point of view.

2. We need only one production rule for syntactic domain $\mathbf{Expr}$ of arithmetic expressions:

$$e ::= n \mid x \mid e + e \mid e - e \mid e * e.$$

Semantics

Formal semantics of programming languages:

- provides abstract units which represent only essential features from all possible program executions, that are written in a given language,
- ignores details, that are not important from point of semantics.

Essential features are usually considered as follows:

- the relation between input and output,
- termination or non-termination of program execution.

Implementation details, like

- real memory addresses in computer,
- real time of program execution, or
- computer architecture,

are from the point of semantics **inessential** and they are ignored.

Semantic methods

Selected semantic methods:

- operational semantics:
    - natural semantics,
    - structural operational semantics,
- denotational semantics,
- axiomatic semantics,
- action semantics,
- attribute grammars,
- algebraic semantics,
- categorical semantics,
- game semantics.

Attribute grammars

- a formal way to define attributes for the productions of a formal grammar, associating these attributes to values,
- the evaluation occurs in the nodes of the abstract syntax tree, when the language is processed by some parser or compiler,
- they hang together with denotational semantics, an output language is filled with attribute annotations.

Algebraic semantics defines a model of specification (ADT) as many-sorted algebra.

Categorical semantics works with category theory and usually defines functor between categories as higher-order semantic mapping.

Game semantics with the concepts of validity on game theoretic concepts.

Axiomatic semantics works with logical formulas and **Action semantics** describes program as a sequence of computational actions.

Operational semantics

Operational semantics:

- the meaning of a construct is specified by the computation it induces when it is executed on a machine,
- in particular, it is of interest *how* the effect of a computation is produced and not merely what the results of execution are,
- the first version was formulated for Algol68,
- the meaning of program is described in terms of abstract machine,
- in this method we are interested in how the states are modified during the execution of the statement,
- the meaning of statements is specified by a transition system,
- we shall consider two different approaches to operational semantics:
  - natural semantics,
  - structural operational semantics.

Natural semantics

Formulated by **Gilles Kahn** in 1985.
Natural semantics

- formulates the relationship between the initial and the final state of an execution, therefore the transition relation will specify the relationship between the initial and the final state for each statement,
- allows to specify mane aspects of programming languages and specification languages: type systems, static analysis, dynamic semantics, etc.,
- allows to specify the parts of compiler and to translate them directly into executable code,
- is known also as the **big-steps semantics**.

Example in Natural semantics

We consider a trivial program:

$$z := x; x := y; y := z$$

We are interested into the resulting state after the program execution. In the natural semantics, the execution of the example program in the input state is represented by the following derivation tree:

$$\dfrac{\dfrac{\langle z := x, s_0\rangle \to s_1 \quad \langle x := y, s_1\rangle \to s_2}{\langle z := x; x := y, s_0\rangle \to s_2} \qquad \langle y := z, s_2\rangle \to s}{\langle z := x; x := y; y := z, s_0\rangle \to s}$$

Let $s_0 = [x \mapsto \mathbf{5}, y \mapsto \mathbf{7}, z \mapsto \mathbf{0}]$ be an initial state.

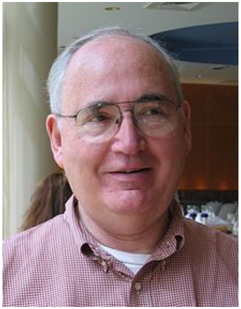
The resulting state s we obtain by applying particular steps:

$$\begin{array}{lcl} s_1 & = & s_0[z \mapsto \mathbf{5}] \\ s_2 & = & s_1[x \mapsto \mathbf{7}] \\ s & = & s_2[y \mapsto \mathbf{5}] \\ & = & [x \mapsto \mathbf{7}, y \mapsto \mathbf{5}, z \mapsto \mathbf{5}] \end{array}$$

Structural operational semantics

Formulated by **Dana Scott** (1970) and later by **Gordon Plotkin** (1981).

- the main idea is to formulate the behavior of program by its components,
- the emphasis is on the **individual steps** of the execution,
- provides an **inductive view** on operational semantics,
- also known as the **small-steps semantics**.



Example in Structural operational semantics

We consider again the trivial program:

$$z := x; x := y; y := z$$

and an initial state $s_0 = [x \mapsto \mathbf{5}, y \mapsto \mathbf{7}, z \mapsto \mathbf{0}]$.

We shall record the execution of the example program in a state s_0 by the following derivation sequence:

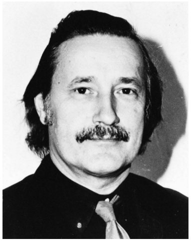
$$\begin{array}{ll}
\multicolumn{2}{l}{\langle z := x; x := y; y := z, [x \mapsto \mathbf{5}, y \mapsto \mathbf{7}, z \mapsto \mathbf{0}]\rangle} \\
\Rightarrow & \langle x := y; y := z,\ [x \mapsto \mathbf{5}, y \mapsto \mathbf{7}, z \mapsto \mathbf{5}]\rangle \\
\Rightarrow & \langle y := z,\ [x \mapsto \mathbf{7}, y \mapsto \mathbf{7}, z \mapsto \mathbf{5}]\rangle \\
\Rightarrow & \quad [x \mapsto \mathbf{7}, y \mapsto \mathbf{5}, z \mapsto \mathbf{5}]
\end{array}$$

The last element in the sequence is the resulting state.

Denotational semantics

Authors of denotational semantics are **Christopher Strachey** (1911-1975) and **Dana Scott**. They formulated denotational semantics in sixties of 20th Century.
Denotational semantics

- meaning of program is described by mathematical objects – the **denotations**,
- meaning of program is defined as semantic function from syntactic to semantic domain,
- is **fully compositional**, i.e. the denotation of compound constructions is defined only by the semantics of sub-constructs,
- we are merely interested in the **effect** of executing a program – an association between initial and final states,
- sometimes known also as **mathematical semantics**.



Example in denotational semantics

We consider again the trivial program:

$$z := x; x := y; y := z$$

and an initial state $s_0 = [x \mapsto \mathbf{5}, y \mapsto \mathbf{7}, z \mapsto \mathbf{0}]$.
Meaning of program is a composition of three semantic functions for assignment statements:

$$\mathscr{S}[\![z := x]\!], \quad \mathscr{S}[\![x := y]\!] \quad \text{and} \quad \mathscr{S}[\![y{:=}z]\!]$$
$$\mathscr{S}[\![z := x; x := y; y := z]\!] \;=\; \mathscr{S}[\![y := z]\!] \circ \mathscr{S}[\![x := y]\!] \circ \mathscr{S}[\![z{:=}\, x]\!].$$

The resulting state of the program execution from the initial state s_0 we obtain by application an initial state in this compound function:

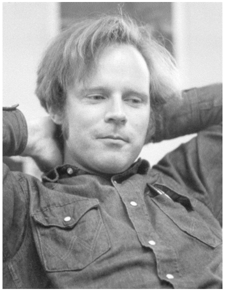
$$\begin{aligned}
&\mathscr{S}[\![z := x; x := y; y := z]\!]([x \mapsto \mathbf{5}, y \mapsto \mathbf{7}, z \mapsto \mathbf{0}])\\
=\;&(\mathscr{S}[\![y := z]\!] \circ \mathscr{S}[\![x := y]\!] \circ \mathscr{S}[\![z{:=}\, x]\!])([x \mapsto \mathbf{5}, y \mapsto \mathbf{7}, z \mapsto \mathbf{0}])\\
=\;&\mathscr{S}[\![y := z]\!]\;(\mathscr{S}[\![x := y]\!]\;(\mathscr{S}[\![z := x]\!]\;([x \mapsto \mathbf{5}, y \mapsto \mathbf{7}, z \mapsto \mathbf{0}])))\\
=\;&\mathscr{S}[\![y := z]\!]\;(\mathscr{S}[\![x := y]\!]\;([x \mapsto \mathbf{5}, y \mapsto \mathbf{7}, z \mapsto \mathbf{5}]))\\
=\;&\mathscr{S}[\![y := z]\!]\;([x \mapsto \mathbf{7}, y \mapsto \mathbf{7}, z \mapsto \mathbf{5}])\\
=\;&[x \mapsto \mathbf{7}, y \mapsto \mathbf{5}, z \mapsto \mathbf{5}]
\end{aligned}$$

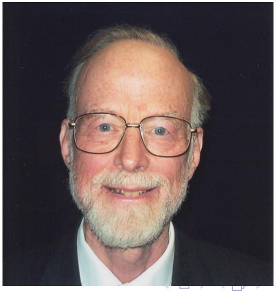
Axiomatic semantics

The first approach to axiomatic semantics formulated **Robert Floyd** (1967). The second one (and more famous) formulated **Sir Charles Antony Richard Hoare** (1969).

Axiomatic semantics

- the meaning of program is described by logical formulas (preconditions and postconditions),
- is used for the proof of partial correctness of programs,
- one of the first applications was axiomatic definition of programming language Pascal (1973).





Example in axiomatic semantics

We consider again the trivial program:

$$z := x; x := y; y := z$$

and an initial state $s_0 = [x \mapsto \mathbf{5}, y \mapsto \mathbf{7}, z \mapsto \mathbf{0}]$.
We put precondition at the beginning of program and the postcondition at the end of program:

$$\{x = \ \mathbf{n} \wedge y = \ \mathbf{m}\}\ z := x; x := y; y := z\ \{y = \ \mathbf{n} \wedge x = \ \mathbf{m}\}$$

An initial state $s_0 = [x \mapsto \mathbf{5}, y \mapsto \mathbf{7}, z \mapsto \mathbf{0}]$ fulfills the condition for $\mathbf{n} = \mathbf{5}$ and $\mathbf{m} = \mathbf{7}$.
The proof of partial corectness is represented in the following **proof tree:**

$$\dfrac{\dfrac{\{p_0\}z := x\{p_1\} \quad \{p_1\}x := y\{p_2\}}{\{p_0\}z := x; x := y\{p_2\}} \qquad \{p_2\}y := z\{p_3\}}{\{p_0\}z := x; x := y; y := z\{p_3\}}$$

where

$$\begin{array}{ll} p_0 \equiv x = \mathbf{n} \wedge y = \mathbf{m} & p_1 \equiv z = \mathbf{n} \wedge y = \mathbf{m} \\ p_2 \equiv z = \mathbf{n} \wedge x = \mathbf{m} & p_3 \equiv y = \mathbf{n} \wedge x = \mathbf{m} \end{array}$$

Action semantics

Authors of Action semantics are Peter David Mosses and David Watt (1992).

- is hybrid semantics – **modularization** of denotational semantics by dividing the formalization process into two levels,
- the first level is **macrosemantics** and the second one is **microsemantics**,
- action semantics works with three kinds of entities: **actions, data** and **yielders**.

Example in action semantics

We consider again the trivial program:

$$z := x; x := y; y := z$$

and an initial state $s_0 = [x \mapsto \mathbf{5}, y \mapsto \mathbf{7}, z \mapsto \mathbf{0}]$.
Program is being transcribed into sequence of actions:
$\mathscr{E}xecute[\![z := x;\ x := y;\ y := z]\!] =$
$(\mathscr{E}xecute[\![z := x]\!]$ and then $\mathscr{E}xecute[\![x := y]\!])$ and then $\mathscr{E}xecute[\![y := z]\!]$
The next step(s):
(($\mathscr{E}valuate\ x$ and then store the given number in the cell bound to z)
and then
($\mathscr{E}valuate\ y$ and then store the given number in the cell bound to x))
and then
($\mathscr{E}valuate\ z$ and then store the given number in the cell bound to y) =

((give the number stored in the cell bound to x then store the given number in the cell bound to z)
and then
(give the number stored in the cell bound to y then store the given number in the cell bound to x))
and then
(give the number stored in the cell bound to z then store the given number in the cell bound to y)

The resulting state is obtained as follows:
$(s_1\,[z \mapsto x]\,(x = \mathbf{5})$ and then $s_2\,[x \mapsto y]\,(y = \mathbf{7}))$ and then $s_3\,[y \mapsto z]\,(z = \mathbf{5})$

Finally, the resulting state is $s_3\,[x \mapsto \mathbf{7}, y \mapsto \mathbf{5}, z \mapsto \mathbf{5}]$

Using of formal methdos

Generally, formal semantics:

- provides unambiguous meaning of programs written in programming language,
- helps designers to prepare good and useful programming languages,
- serves for designers to write correct compilers,
- encourages users/programmers how to use language constructions properly.

Bibliography:

- H.R.Nielson, F.Nielson: Semantics with applications. An appetizer. Springer, 2007.
- D.A Schmidt: Denotational semantics. Methodology for language development. Allyn and Bacon, 1986.
- P.D.Mosses: A tutorial on action semantics. Formal Methods Oxford, 1996.
- C.A.R.Hoare, N.Wirth: An axiomatic definition of the programming language Pascal, Acta Informatica 2(4), pp.335-355, 1973 .