Structure of Programming Languages I: Denotational Semantics

Wolfgang Schreiner
Research Institute for Symbolic Computation
Johannes Kepler University
Wolfgang.Schreiner@risc.uni-linz.ac.at

Programming Language

Any notation for giving instructions,
e.g. Pascal, input commands to application program, ...

- Syntax

  Appearance and structure of sentences.

- Semantics

  Assignment of meaning to sentences
  e.g. numbers, functions, machine actions,
  ...

- Pragmatics

  Usability of the language
  e.g. application areas, performance, ...

Features of every computer program.

Implementation

Two main parts:

1. Input checker module ("Parser")

   Reads input, verifies that it has proper syntax, generates internal representation.

2. Evaluation module

   Evaluates input to corresponding output thus defining the semantics of the language.

Implementation of language is a pragmatic issue.

Interpretation

Execution of the program.

Interpreter defines meaning (by its actions).

Compilation

Transformation of the program into an equivalent version in the machine language.

Compiler preserves meaning (by notion of "equivalence").

Formal Specifications

- Syntax: Backus-Naur Form (BNF)
  - *Correspondence between BNF and parser.*
  - *Input to parser generator.*
- Semantics: ?
  - *Precise Standard for implementation.*
  - *Useful user documentation.*
  - *Tool for design and analysis of language*
  - *Input to compiler generator.*

Semantics is much more difficult to describe ("semantics is everything that cannot be described in BNF").

Operational Semantics

- Language defined by interpreter (abstract machine).
- Each construct defined by a transition rule.
- Meaning of a program is a sequence of interpreter states.

$$P : S_1 \to S_2 \to \ldots \to S_n$$

P ... *program*
S_i ... *interpreter state*

Notation for interpreter may be as complex as language itself.

Axiomatic semantics

- Language defined by system of logical axioms and inference rules.
- Each construct defined by an axiom.
- Not the meaning of a program but its properties are defined.

$$\{A\}\ P\ \{B\}$$

P ... *program*
A ... *precondition*
B ... *postcondition*

Provable properties need not characterize program uniquely.

Denotational semantics

- The meaning of a program is a (mathematical) object.
- Each construct is mapped by a valuation function into its meaning (denotation).

$$F(P) = D$$

P ... *program*
F ... *valuation function*
D ... *denotation*

More abstract than operational semantics (no computation steps), more concrete than axiomatic semantics (explicit meaning).

Application Areas

- Axiomatic: initial specification.

  Which properties shall language have?

- Denotational: meaning.

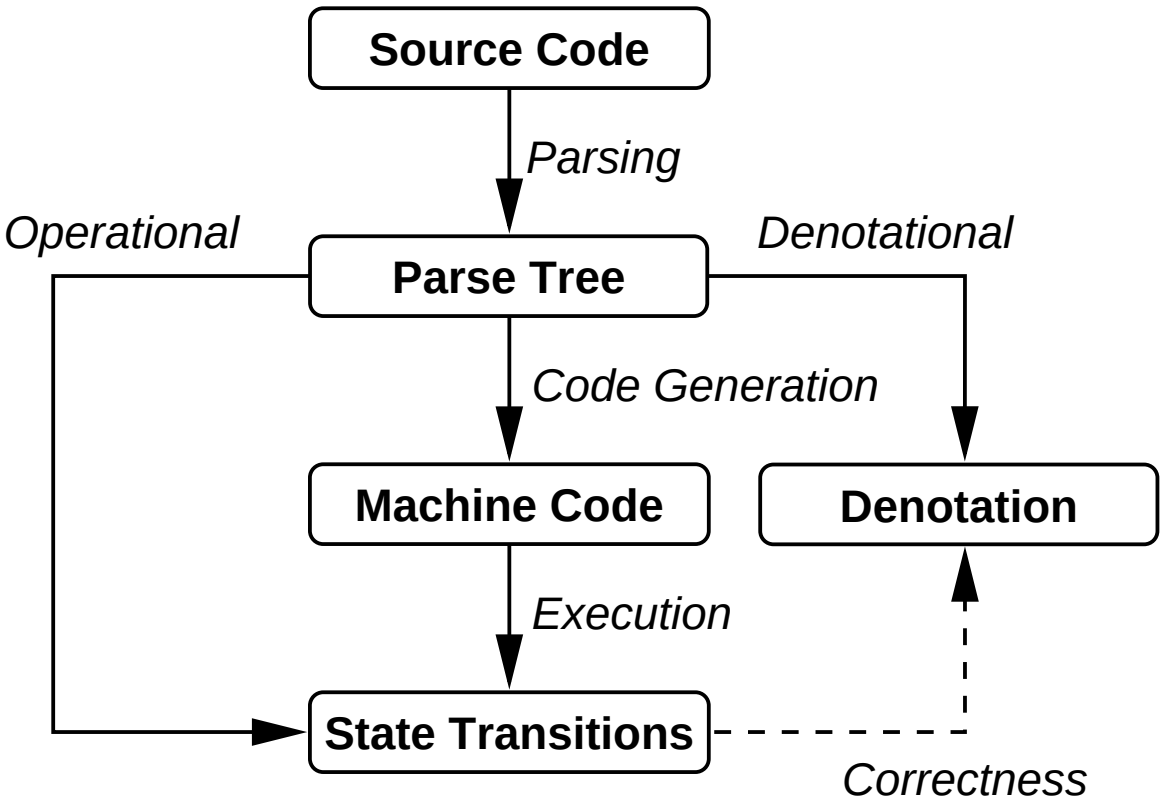
  Which semantics provides properties?

- Operational: implementation.

  How can semantics be implemented?

Complementary aspects.

Relationship



Correctness of implementation can be verified with respect to the denotation.

Valuation Function

- Domain: Abstract syntax structures ("parse trees") of the language.
- Target: Objects of semantic domains.
- Structural definition (meaning of a tree is defined by meanings of its subtrees).

A valuation function maps an abstract syntax into some semantic domain.

A Language of Binary Numerals

- Abstract Syntax:

  Syntactic domains:

  B ∈ Binary-numeral

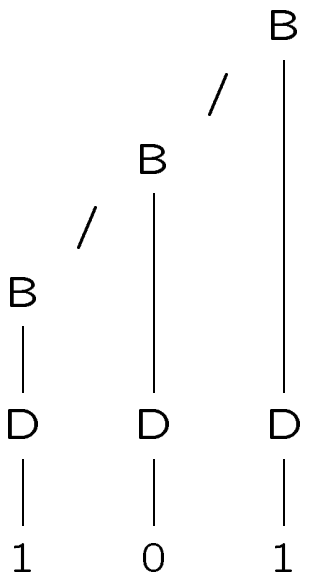
  D ∈ Binary-digit

  Syntax rules:

  B ::= BD | D

  D :: = 0 | 1

- Sentences:



Meaning of Terminal Sentences

- Subtree:

- Meaning: **D**(D) = *zero*

- Notation: **D** [[0]] = *zero*

Valuation function:

D [[0]] = *zero*

D [[1]] = *one*

Meaning of Non-Terminal Sentences

- Subtree:

- Meaning: $\mathbf{B}(\ B\ ) = \mathbf{D}(\ \ D\ )$

- Notation: $\mathbf{B}[\![\ D\ ]\!] = \mathbf{D}[\![\ D\ ]\!]$

Valuation function:

$\mathbf{B}[\![\ D\ ]\!] = \mathbf{D}[\![\ D\ ]\!]$

$\mathbf{B}[\![\ BD\ ]\!] =$

$(\mathbf{B}[\![\ B\ ]\!]$ *times two*$)$ *plus* $\mathbf{D}[\![\ D\ ]\!]$

Abstract Syntax

B ∈ Binary-numeral

D ∈ Binary-digit

B ::= BD | D

D :: = 0 | 1

Semantic Algebras

Natural Numbers:

Domain *Nat* = **N**

Operations *zero*, *one*, *two*, . . . : *Nat*

plus, *times*: *Nat* × *Nat* ⟶ *Nat*

Valuation Functions

B: Binary-numeral ⟶ *Nat*

B[[D]] = **D**[[D]]

B[[BD]] =

(**B**[[B]] *times two*) *plus* **D**[[D]]

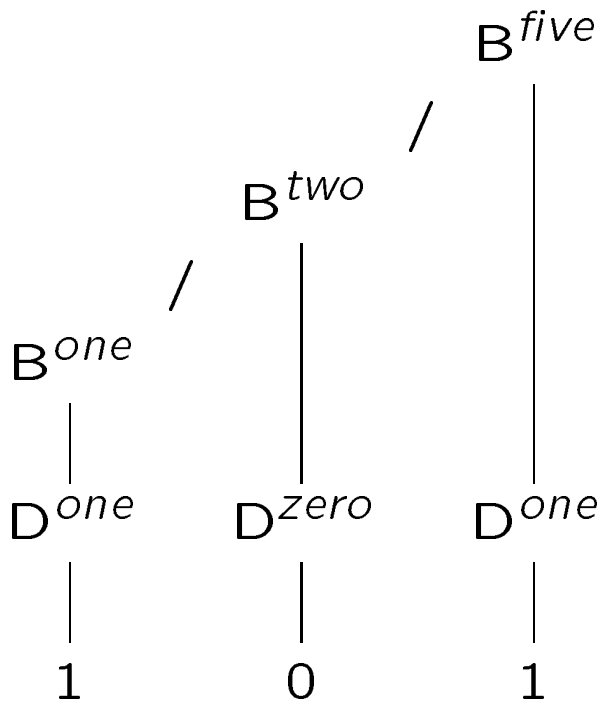
D: Binary-digit ⟶ *Nat*

D [[0]] = *zero*

D [[1]] = *one*

Meaning of Sentence

Annotation of abstract syntax tree



Computation can proceed bottom-up or top-down!

Course Schedule

David A. Schmidt. *Denotational Semantics — A Methodology for Language Development.* Allyn and Bacon, Boston, Massachusetts, 1986.

1. Introduction

   Methods for semantics specifications, preview on basic structure of denotational definitions.

2. Syntax and Semantics

   Abstract syntax definitions, mathematical and structural induction, sets, functions, semantic domains.

3. Domain Theory I

   Semantic algebras, primitive domains, compound domains, recursive function and domain definitions.

4. Basic Structure of Denotational Definitions

   Valuation function, format of a denotational definition, a calculator language.

5. Imperative Languages I

   A language with assignment, an interactive file editor, input and output.

6. Imperative Languages II

   Altering the properties of stores, delayed evaluation, multiple stores, noncommunicating commands.

7. Domain Theory II

   Recursively defined functions, partial orderings, continuous functions, least fixed points, domains are cpos, examples.

8. Languages with Contexts I

   A block-structured language, stack-managed storage, the meaning of identifiers.

9. Languages with Contexts II

   An applicative language, scoping rules, self-application, recursive definitions.

10. Languages with Contexts III

    Compound data structures.

11. Language Design Principles I

    Abstraction, recursive bindings, parameterization, polymorphism and typing.

12. Language Design Principles II

    Correspondence, qualification, orthagonality.