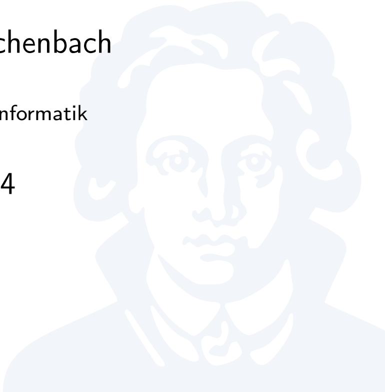
Foundations of Programming Languages

Introduction

Prof. Dr. Christoph Reichenbach

Fachbereich 12 / Institut für Informatik

17. Oktober 2014



Contents

- Languages: structure and semantics
- Language Implementation
- Foundations of program analysis
- Foundations of software tools

What we won't be covering

- Advanced topics in formal semantics
- Compiler backends (register selection, instruction selection)
- Lexing and Parsing

Literature

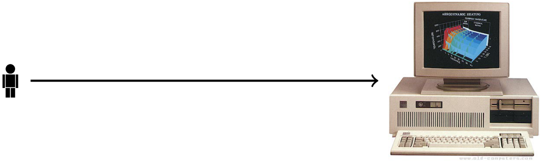
- **Programming Languages and Semantics**
  - "Types and Programming Languages" by Benjamin C. Pierce
  - "Concepts of Programming Languages (5th or later edition)" by Robert W. Sebesta
- **Compilers and Program Analysis**
  - "Compilers: Principles, Techniques, and Tools (2nd Edition)" by Alfred Aho, Monica Lam, Ravi Sethi and Jeffrey Ullman
  - "Principles of Program Analysis" by Flemming Nielson, Hanne R. Nielson and Chris Hankin
  - "Modern Compiler Implementation in C/Java/ML" by Andrew Appel
- **Assembly and Machine Language**
  - "Computer Organization and Design: The Hardware/Software Interface", by David Patterson and John Hennessy
- **C**
  - "The C Programming Language", by Brian Kernighan and Dennis Ritchie
  - C11 specification

First two lectures

1. Today we will look at:
    - *syntax*: Describe structure of programs
    - *semantics*: Derive meaning from syntax
2. For next week we will look at:
    - *assembly/machine language*: The CPU's own language
    - *language implementations*: Teaching the CPU higher-level languages

Programming Languages

Why Programming Languages? (1/3)



- Mouse clicks & drags
- Pushing & Swiping
- Voice commands
- *Text input*

Many ways to talk to the computer

Why Programming Languages? (2/3)

Utility of interaction method:

- Can I interact *quickly*?
- Can I record my instructions?
- Can I inspect/modify the recorded instructions?
- Are my records *precise*?
- Can I communicate with other humans *about* my records?
    - Do they match a known vocabulary?

Why Programming Languages? (3/3)

	Click&Drag	**Swipe**	**Voice**	**Text**
Speed	++	++	+	−
Record	?	?	+	++
Record Precision	?	?	+	++
Record: Inspect/-Mod	?	?	−	++
Communicate About	−	−	++	++

What do programs mean?

Let's run the following program in some language:

```
print(32767 + 1);
```

Which of the following outputs is correct?

- `32768`
- `32767 + 1`
- `-32768`
- `octopus`
- *no* visible output

Must know the language's syntax and semancis

Structure and Meaning

Pragmatics: Intent
"I need more space on my disk"

Semantics: Meaning
"Delete all temporary files"

Syntax: Word choice & arrangement

```
rm -rf /tmp/*
```

Semantics

Semantics: The study of *meaning* (logic, linguistics)

- “meaning should follow structure”
  - This is a *hypothesis* in linguistics (seems to hold)
  - And a *proposal* in logic (turns out to work reasonably well)

Example:

- If expression ‘`X`’ has meaning ‘v’
- And expression ‘`Y`’ has meaning ‘w’
- Then expression ‘`(X) / (Y)`’ has meaning ‘whatever number you get when you compute $\frac{v}{w}$’

What if ‘v’ is not a number, or ‘w’ is zero?

Backus-Naur Form: Specifying Syntax

Assume *nat* is a natural number:
Formalise the rules with *Backus-Naur-Form* (BNF):

- ‘Any number is an expression.’
  - $\mathit{expr} ::= \mathit{nat}$
- ‘Any two expressions with a + in between is also an expression.’
  - $\mathit{expr} ::= \langle \mathit{expr} \rangle \text{‘+’} \langle \mathit{expr} \rangle$
- ‘Any two expressions with a * in between is also an expression.’
  - $\mathit{expr} ::= \langle \mathit{expr} \rangle \text{‘*’} \langle \mathit{expr} \rangle$

Or in short:

$$\mathit{expr} ::= \mathit{nat} \mid \langle \mathit{expr} \rangle \text{‘+’} \langle \mathit{expr} \rangle \mid \langle \mathit{expr} \rangle \text{‘*’} \langle \mathit{expr} \rangle$$

Backus-Naur Form: Example

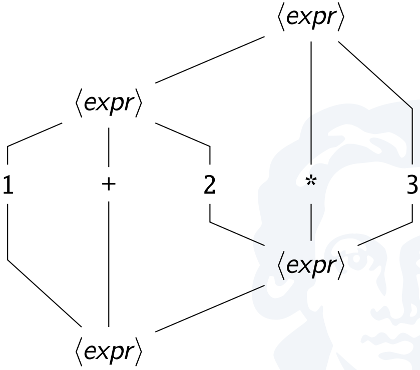
$$\mathit{expr} ::= \mathit{nat} \mid \langle \mathit{expr} \rangle \text{`+'} \langle \mathit{expr} \rangle \mid \langle \mathit{expr} \rangle \text{`*'} \langle \mathit{expr} \rangle$$

(1+2)*3

alternative parse:

a parse:

1+(2*3)



Ambiguity! Parsers must know which parse we mean!

Syntax of a simple toy language

Syntax of language STOL:

$$
\begin{array}{lcl}
\mathit{expr} & ::= & \mathit{nat} \\
 & \mid & \langle \mathit{expr} \rangle \text{`}+\text{'} \langle \mathit{expr} \rangle \\
 & \mid & \text{`}\texttt{ifnz}\text{'} \langle \mathit{expr} \rangle \text{`}\texttt{then}\text{'} \langle \mathit{expr} \rangle \text{`}\texttt{else}\text{'} \langle \mathit{expr} \rangle
\end{array}
$$

Examples:

- `5`
- `5 + 27`
- `ifnz 5 + 2 then 0 else 1`

Meaning of our toy language: examples

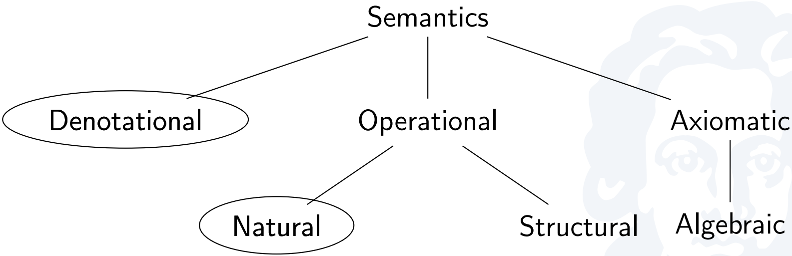
What we want the meaning to be:

`5`	5
`5 + 27`	32
`ifnz` `5 + 2` **`then`** `1` **`else`** `0`	1

Can we describe this formally?

Defining Meaning

The principal schools of semantics:



Semantics

Denotational

Operational

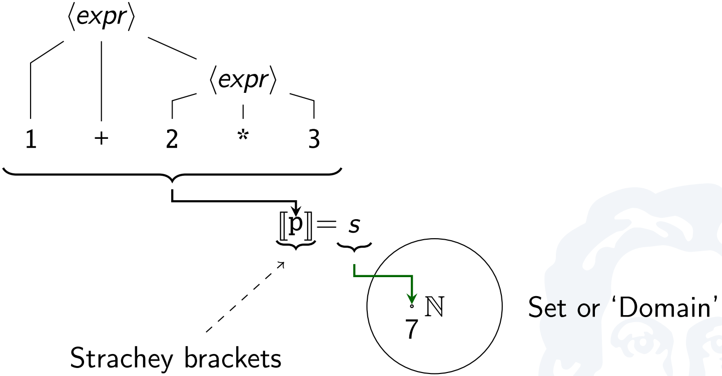
Axiomatic

Natural

Structural

Algebraic

Denotational Semantics



⟨*expr*⟩

⟨*expr*⟩

1 + 2 * 3

$\underbrace{\llbracket \mathrm{p} \rrbracket}= \underbrace{s}$

ℕ

7

Strachey brackets

Set or ‘Domain’

- Maps program to mathematical object
- Equational theory to reason about programs

Directly maps program to its mathematical ‘meaning’

Denotational semantics of STOL

Distinguish:

- `nat` is set of program numbers (`0`, `1`, `2`, …)
  (In compilers: *character strings*)
- $\mathbb{N}$ is set of natural numbers (0, 1, 2, …)
  (In compilers: *unsigned int* or *BigInt* types)

$$
\begin{array}{rcl}
n & \in & \mathtt{nat} \\
e, e_1, e_2, e_3 & \in & \mathtt{expr} \\
[\![n]\!] & = & \left\{ \begin{array}{lcl} 0 & \iff & n = \mathtt{0} \\ 1 & \iff & n = \mathtt{1} \\ 2 & \iff & n = \mathtt{2} \\ & & \ldots \end{array} \right. \\
[\![e_1 \mathtt{+} e_2]\!] & = & [\![e_1]\!] + [\![e_2]\!] \\
[\![\textbf{ifnz}\ e_1\ \textbf{then}\ e_2\ \textbf{else}\ e_3]\!] & = & \left\{ \begin{array}{lcl} [\![e_2]\!] & \iff & [\![e_1]\!] \neq 0 \\ [\![e_3]\!] & \iff & [\![e_1]\!] = 0 \end{array} \right.
\end{array}
$$

Operational Semantics: The two branches

- Natural Semantics (Big-Step Semantics)
  - $p \Downarrow v$: p evaluates to v
  - Describes *complete* evaluation
  - Compact, useful to describe interpreters
- Structural Operational Semantics (Small-Step Semantics)
  - $p_1 \rightarrow p_2$: p_1 evaluates one step to p_2
  - Captures individual *evaluation steps*
  - Verbose/detailed, useful for formal proofs

Natural (Operational) Semantics

$$\frac{\overbrace{P_1 \quad \cdots \quad P_n}^{\text{Preconditions}}}{\underbrace{\mathsf{e}}_{\text{Program/Expression}} \Downarrow \underbrace{\mathsf{v}}_{\text{Value}}}$$

If $P_1, \ldots, P_n$ all hold, then e evaluates to v.

- e: Arbitrary program (expression, in our example)
- v: Value that can't be evaluated any further (natural number, in our example)

Natural Semantics of our simple toy language

$$\begin{array}{rcl} n, n_1, n_2, n_3 & \in & \texttt{nat} \\ e, e_1, e_2, e_3 & \in & \texttt{expr} \end{array}$$

$$\frac{}{n \Downarrow n}\ (\mathit{val}) \qquad \frac{e_1 \Downarrow n_1 \quad e_2 \Downarrow n_2 \quad n = n_1 + n_2}{e_1 + e_2 \Downarrow n}\ (\mathit{add})$$

$$\frac{e_1 \Downarrow n \quad n \neq 0 \quad e_2 \Downarrow n_2}{\textbf{ifnz}\ e_1\ \textbf{then}\ e_2\ \textbf{else}\ e_3 \Downarrow n_2}\ (\mathit{ifnz})$$

$$\frac{e_1 \Downarrow 0 \quad e_3 \Downarrow n_3}{\textbf{ifnz}\ e_1\ \textbf{then}\ e_2\ \textbf{else}\ e_3 \Downarrow n_3}\ (\mathit{ifz})$$

Note:

- $(+)$ is arithmetic addition
- $+$ is a symbol in our language
- For simplicity, we set $\texttt{nat} = \mathbb{N}$

Natural Semantics: Example

$$\cfrac{\cfrac{\cfrac{}{3 \Downarrow 3}\ (\mathit{val}) \qquad \cfrac{}{2 \Downarrow 2}\ (\mathit{val}) \qquad 5 = 3+2}{3+2 \Downarrow 5}\ (\mathit{add}) \qquad \cfrac{}{1 \Downarrow 1}\ \mathit{val}}{\texttt{\textbf{ifnz}}\ 3+2\ \texttt{\textbf{then}}\ 1\ \texttt{\textbf{else}}\ 0 \Downarrow 1}\ (\mathit{ifnz})$$

What's the point?

- Denotational and natural semantics *look* very similar
- Structural differences:
  - Denotational semantics describe a *function* $\llbracket - \rrbracket$
  - Natural semantics define a *relation* $(\Downarrow)$
  - Denotational semantics relies on mathematical *domain* with underlying equational theory
- Practical differences:
  - Natural Semantics requires less formal apparatus to describe (no domains)
  - Natural Semantics can't describe partial progress in non-terminating programs

Extending our language with ‘let’

Name bindings $x \in \mathit{name}$:

$$
\begin{array}{lcl}
\mathit{expr} & ::= & \mathit{nat} \\
 & | & \langle \mathit{expr} \rangle \text{‘}+\text{’} \langle \mathit{expr} \rangle \\
 & | & \text{‘}\texttt{ifnz}\text{’} \langle \mathit{expr} \rangle \text{‘}\texttt{then}\text{’} \langle \mathit{expr} \rangle \text{‘}\texttt{else}\text{’} \langle \mathit{expr} \rangle \\
 & | & \mathit{name} \\
 & | & \text{‘}\texttt{let}\text{’} \mathit{name} \text{‘}=\text{’} \langle \mathit{expr} \rangle \text{‘}\texttt{in}\text{’} \langle \mathit{expr} \rangle
\end{array}
$$

Example:

$$[\![\mathbf{let}\ \texttt{x = 2 + 3}\ \mathbf{in}\ \texttt{x + x}]\!] = 10$$

But what is $[\![x]\!]$ by itself?

Environments

The meaning of a variable depends on what value we bind it to.

Environment: $E : \texttt{\textbf{name}} \rightarrow \texttt{\textbf{value}}$

- Environments are *partial functions* from names to ‘values’
- In our running example, `value` = `nat`

Notation:

let $E' = [x := v]E$

then:

$$E'(y) = \begin{cases} v & \iff \quad y = x \\ E(y) & \quad\quad\;\; \textit{otherwise} \end{cases}$$

Environments in Denotational Semantics

Introduce E as index to semantic function:

$$[\![-]\!]_E = \ldots$$

$$\begin{array}{rcl}
n & \in & \mathtt{nat} \\
e, e_1, e_2, e_3 & \in & \mathtt{expr} \\
x & \in & \mathtt{name} \\
[\![n]\!]_E & = & n \text{ interpreted in } \mathbb{N} \\
[\![e_1 \mathtt{+} e_2]\!]_E & = & [\![e_1]\!]_E + [\![e_2]\!]_E \\
[\![\mathbf{\mathtt{ifnz}}\ e_1\ \mathbf{\mathtt{then}}\ e_2\ \mathbf{\mathtt{else}}\ e_3]\!]_E & = & \begin{cases} [\![e_2]\!]_E & \iff & [\![e_1]\!]_E \neq 0 \\ [\![e_3]\!]_E & \iff & [\![e_1]\!]_E = 0 \end{cases} \\
[\![x]\!]_E & = & E(x) \\
[\![\mathbf{\mathtt{let}}\ x = e_1\ \mathbf{\mathtt{in}}\ e_2]\!]_E & = & [\![e_2]\!]_{[x := [\![e_1]\!]_E] E}
\end{array}$$

Environments in Natural Semantics

We borrow the turnstile ($\vdash$) from formal logic:

$$\frac{}{E \vdash n \Downarrow n}\ (\mathit{val}) \qquad \frac{E \vdash e_1 \Downarrow n_1 \quad E \vdash e_2 \Downarrow n_2 \quad n = n_1 + n_2}{E \vdash e_1 + e_2 \Downarrow n}\ (\mathit{add})$$

$$\frac{E \vdash e_1 \Downarrow n \quad n \neq 0 \quad E \vdash e_2 \Downarrow n_2}{E \vdash \mathtt{\textbf{ifnz}}\ e_1\ \mathtt{\textbf{then}}\ e_2\ \mathtt{\textbf{else}}\ e_3 \Downarrow n_2}\ (\mathit{ifnz})$$

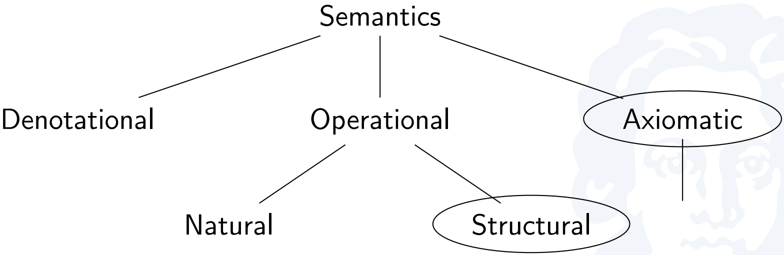
$$\frac{E \vdash e_1 \Downarrow 0 \quad E \vdash e_3 \Downarrow n_3}{E \vdash \mathtt{\textbf{ifnz}}\ e_1\ \mathtt{\textbf{then}}\ e_2\ \mathtt{\textbf{else}}\ e_3 \Downarrow n_3}\ (\mathit{ifz})$$

$$\frac{E(x) = v}{E \vdash x \Downarrow v}\ (\mathit{var})$$

$$\frac{E \vdash e_1 \Downarrow v \quad ([x := v]E) \vdash e_2 \Downarrow v'}{E \vdash \mathtt{\textbf{let}}\ x = e_1\ \mathtt{\textbf{in}}\ e_2 \Downarrow v'}\ (\mathit{let})$$

Defining Meaning

Let's consider the other schools of semantics now:



Semantics

Denotational

Operational

Axiomatic

Natural

Structural

Structural Operational Semantics (SOS)

(Definition on STOL)

$$\frac{e_1 \longrightarrow^\star 0}{\mathbf{\texttt{ifnz}}\ e_1\ \mathbf{\texttt{then}}\ e_2\ \mathbf{\texttt{else}}\ e_3 \longrightarrow e_3}\ (\mathit{ifz})$$

$$\frac{e_1 \longrightarrow^\star n \quad \nexists n'.n \longrightarrow n' \quad n \neq 0}{\mathbf{\texttt{ifnz}}\ e_1\ \mathbf{\texttt{then}}\ e_2\ \mathbf{\texttt{else}}\ e_3 \longrightarrow e_2}\ (\mathit{ifnz})$$

Comparison to Natural Semantics:

$\Downarrow \subseteq \texttt{expr} \times \texttt{nat}$	$\longrightarrow \subseteq \texttt{expr} \times \texttt{expr}$
rhs is alwyas *fully* evaluated	rhs can be intermediate result

SOS can capture intermediate computational results

STOL-S: State updates

- We remove **let** bindings and instead use:
- `p := 23` Updates variable `p` to 23 (cf. `p = 23` in Python).
- `(p := 23; p)` Sequence: assign, read&return (Sequencing operation, cf. `{ p = 23; return p; }`)

Example:

```
(
  r := 2;
  r := r + r;
  r + 1
) ⟶* 5
```

Stores

Store: $\sigma : \mathtt{\mathbf{name}} \rightarrow \mathtt{\mathbf{value}}$

- Analogous to environments
- Store maps names (‘`name`’) to ‘values’
- Again, `value` = `nat` (for now)

Stores in SOS (1)

- Recursive evaluation may update the store…
- …which the caller must be able to see.
- We adjust $\longrightarrow$ to evaluate tuples $\langle e|\sigma\rangle$:
  $\langle e|\sigma\rangle \longrightarrow \langle v|\sigma'\rangle$
  means:
  - Given a store σ:
  - e evaluates to v, and
  - σ is updated to σ' in the process

Example:

$$\frac{E \vdash \langle e_1|\sigma\rangle \longrightarrow \langle n_1|\sigma'\rangle \quad E \vdash \langle e_2|\sigma'\rangle \longrightarrow \langle n_2|\sigma''\rangle \quad n = n_1 + n_2}{E \vdash \langle e_1 + e_2|\sigma\rangle \longrightarrow \langle n|\sigma''\rangle}\ (add)$$

State is *threaded through* the rule: *evaluation order*

Stores in SOS (2)

Return value for assignment; choice of ‘0’ is arbitrary

$$\frac{\sigma(x) = v}{\langle x | \sigma \rangle \longrightarrow \langle v | \sigma' \rangle}\ (\mathit{var})$$

$$\frac{\langle e | \sigma \rangle \longrightarrow \langle v | \sigma' \rangle}{\langle x := e | \sigma \rangle \longrightarrow \langle 0, [x := v]\sigma'' \rangle}\ (\mathit{update})$$

We discard the return value left of the semicolon

$$\frac{\langle e_1 | \sigma \rangle \longrightarrow \langle v | \sigma' \rangle \quad \langle e_2 | \sigma' \rangle \longrightarrow \langle v' | \sigma'' \rangle}{\langle (e_1 ; e_2) | \sigma \rangle \longrightarrow \langle v' | \sigma'' \rangle}\ (\mathit{seq})$$

Analogously for the other rules.

Axiomatic Semantics

Describe *statements*– not good fit for our current language

$$\{P\}\mathit{statement}\{Q\}$$

- P: Precondition
- Q: Postcondition
- if P holds, then *statement* ensures that Q holds

Example:

$$\{x \geq 0\}\texttt{x := x + 1;}\{x > 0\}$$

Frequently used for "design-by-contract" software development

Comparison

- *Denotational Semantics*
  Equational theory, also describes nontermination
- *Natural Semantics*
  Compact, describes interpreter, doesn't give semantics to nonterminating programs
- *Structural Operational Semantics*
  Describes evaluation strategy, approximates semantics for nontermination
- *Axiomatic Semantics*
  Describes effect of *statements* (before/after), no nontermination
- *Algebraic Semantics*
  Describes effect of *operations* on opaque data structures, no nontermination