EDAP05: Concepts of Programming Languages

LECTURE 1: INTRODUCTION

Christoph Reichenbach

Contents

- Programming languages: structure and semantics
- Some language implementation considerations
  - See the **Compilers** course for more details!
- How to evaluate and compare languages

What we will not be covering

- Assembly language
- Concurrency
- Software tools
- How to build a compiler

Course Structure

Information

- Today's lecture
- Our Textbook
- Course Supplements

Interaction

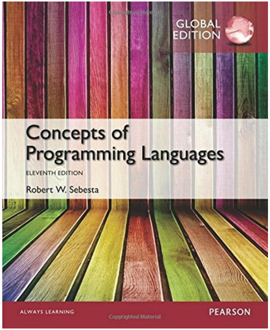
- $2\times$ per week: Class Sessions
- Exercises
- Online discussions via Piazza
- e-mail: `christoph.reichenbach@cs.lth.se`
- TAs:
  - Noric: `noric.couderc@cs.lth.se`
  - Alex: `alexandru.dura@cs.lth.se`

Skills

- Skill-based learning:
  - Enumerated list of skills that you need to pass the exam
  - Skill numbers connected to book, supplements, exercises

Conversational Classroom

- Future lectures are based on the textbook:



(+ Supplements)

- **Read the sections of the book listed on the weekly schedule, prepare your questions ahead of time!**
- Lecture slots *interactive* Q&A

Bring your questions!

Online Systems

All accessible via `http://cs.lth.se/EDAP05` :

- Schedule and Skillset overview
  - What skills are you supposed to know?
  - What lecture / reading material helps you with those skills?
- Discussions via *Piazza*
- Group and Homework management via the *Course Online system* (Online Friday)

Exercises

- Five weekly exercises
  - Starting next week
  - **Available**: Wednesday mornings
  - **Deadline**: Wednesday evening the week after
    One exception per group can be handed in late
  - **Submission**: Course online system
- Done in groups of two (group selection in online system)
- Get help from TAs during labs (sign-up: online system):

Thu	08:15–10:00	E:Alfa, E:Beta
Thu	13:15–15:00	E:Gamma
Fri	08:15–10:00	E:Hacke, E:Panter

- Need 50% on each assignment to be admitted to final exam
- Bonus on final exam if you get 80% or better right:
  - 1% for 80% to $<$ 90%
  - 2% for 90% or more
- *Late exceptions don't count towards bonus points*

Exam

17 January (Fri), 14:00–19:00, in MA:10 G-J

- All exam questions based on the skills from our skill list
- No more than 25% of points based on *synthesis*:
  - Interaction between two or more skills
- Alternative option (only for exchange students): Project + Report + Presentation

Week Overview

Mo	Tu	We	Th	Fr
Class Session	Class Session	New Exercise	Labs	Labs
Mo	**Tu**	**We**	**Th**	**Fr**
		Submit exercise solution		

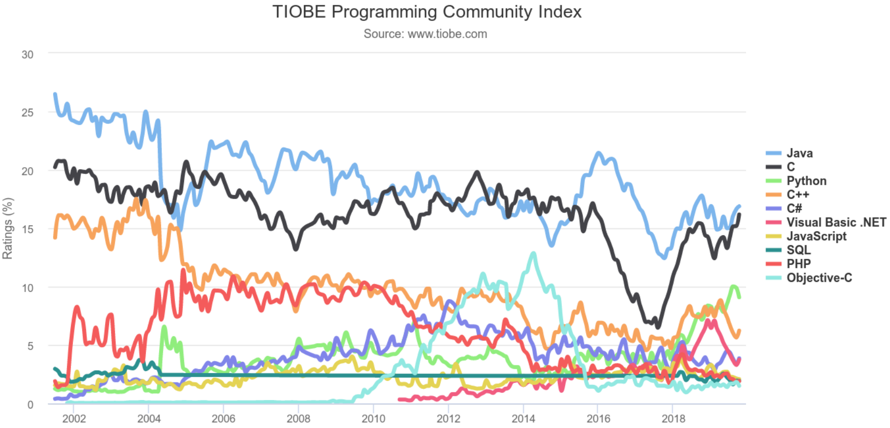
Why Study Programming Languages?

TIOBE Programming Language Index

Oct 2019	Oct 2018	Change	Programming Language	Ratings	Change
1	1		Java	16.884%	-0.92%
2	2		C	16.180%	+0.80%
3	4	↑	Python	9.089%	+1.93%
4	3	↓	C++	6.229%	-1.36%
5	6	↑	C#	3.860%	+0.37%
6	5	↓	Visual Basic .NET	3.745%	-2.14%
7	8	↑	JavaScript	2.076%	-0.20%
8	9	↑	SQL	1.935%	-0.10%
9	7	↓	PHP	1.909%	-0.89%
10	15	⇈	Objective-C	1.501%	+0.30%
11	28	⇈	Groovy	1.394%	+0.96%
12	10	↓	Swift	1.362%	-0.14%
13	18	⇈	Ruby	1.318%	+0.21%
14	13	↓	Assembly language	1.307%	+0.06%
15	14	↓	R	1.261%	+0.05%
16	20	⇈	Visual Basic	1.234%	+0.58%
17	12	⇊	Go	1.100%	-0.15%

Source: `tiobe.com`

TIOBE Programming Language Chart



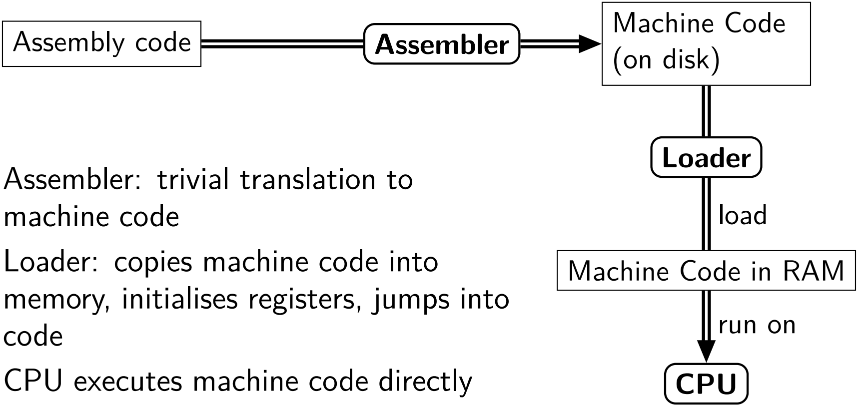
Some Languages

How We Will Proceed

- What are programming languages (not)?
- Describing languages
- Comparing language features
- Exploring language features:
  - Meaning
  - Impact on language implementation

Languages vs. Language Implementations

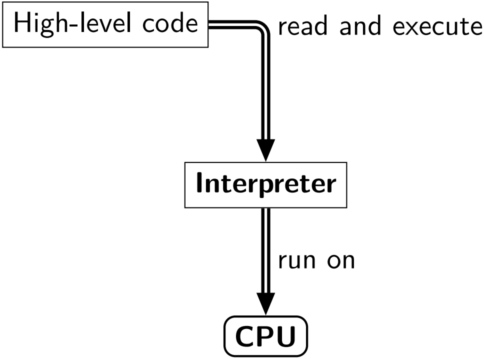
Program Execution



- Assembler: trivial translation to machine code
- Loader: copies machine code into memory, initialises registers, jumps into code
- CPU executes machine code directly

How about languages that the CPU can't execute directly?

Interpretation



- Interpreter reads high-level code, then alternates:
  - Figure out next command
  - Execute command
- May directly encode operational semantics

Examples: Python, Perl, Ruby, Bash, AWK, ...

Example: CPython ('normal' Python)

```
# Python source code
i = 0
while i <= 10:
    print i
    i += 1
```

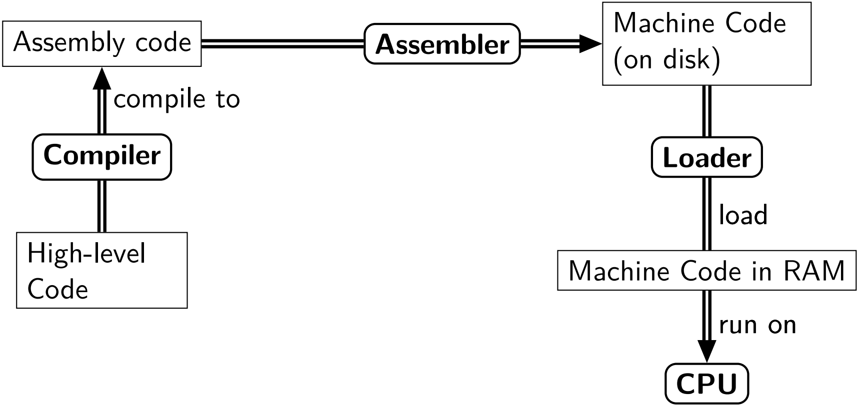
```
0     LOAD_CONST
3     STORE_FAST

6     SETUP_LOOP
9     LOAD_FAST
12    LOAD_CONST
15    COMPARE_OP
18    POP_JUMP_IF_FALSE

21    LOAD_FAST
24    PRINT_ITEM
25    PRINT_NEWLINE

26    LOAD_FAST
29    LOAD_CONST
32    INPLACE_ADD
33    STORE_FAST
36    JUMP_ABSOLUTE

39    POP_BLOCK
```

Python execution (simplified)

- Loop:
  - Load next Python operation
  - Which instruction is it? Jump to specialised code that knows how to execute the instruction:
    - Load parameters to operation
    - Perform operation
    - Continue to next operation

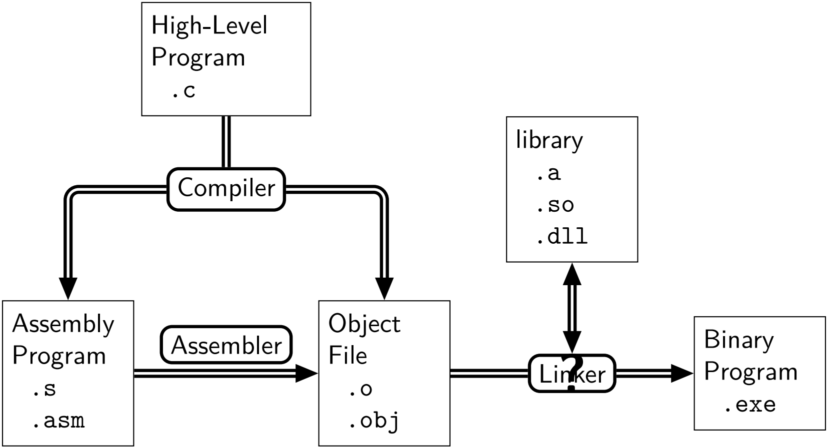
Executing e.g. an addition in CPython takes dozens of assembly instructions

Compilation



Examples: C, C++, SML, Haskell, FORTRAN, . . .

Compiling and Linking in C



Binary program is machine code, can be run by CPU

Comparison: Compilation vs Interpretation

Property	Interpretation	Compilation
Execution performance	slow	**fast**
Turnaround	**fast**	slow (compile & link)
Language flexibility	**high**	limited*

$\star$) Compiler Optimisation $\lightning$ Flexibility

Dynamic Compilation

- Idea: compile code *while executing*
- Theory: best of both worlds
- Practice:
  - Difficult to build
  - Memory usage can increase
  - Performance can be higher than pre-compiled code

Examples: Java, Scala, C#, JavaScript, ...

Summary

- Languages implemented via:
  - stand-alone **Compiler**
  - **Interpreter**
  - **Hybrid Implementation**
    - Part compiler, part interpreter
    - May include: **Dynamic Compiler**
- Trade-off between:
  - Language flexibility
  - CPU time / RAM usage
- Languages may have multiple implementations
  - Example: CPython vs. Jython
  - `gcc` vs. `llvm/clang` vs. MSVC

Language Critique

- What is the best programming language?
  - Best for *what task*?
  - Measured by *what criteria*?
  - Measurements obtained *how*?
    (For most criteria, we don't have good measurement tools!)
- Qualities of:
  - the language
  - the implementation(s)
  - the available tooling
  - the available libraries
  - other infrastructure (user groups, books, ...)

Criterion: Readability

- How easy is it to read software in the language?

A Program

```
++++++++[>++++[
>++>+++>+++>+<<
<<-]>+>+>->>+[<
]<-]>>.>---.+++
++++..+++.>>.<-
.<.+++.------.-
-------.>>+.>++.
```

- Program 1:

$$\sqrt{\sum_{v \in S} v^2}$$

- Program 2:
  Multiply each number in S with itself, add up all the results to compute a *sum*, and then give me the nonnegative number that, when multiplied with itself, is equal to that *sum*.

- Readability depends on:
  - Problem domain (typical notation?)
  - Reader's background
- Multiple general *characteristics* help us understand readability

Simplicity

- Small number of features
- Minimal redundancy

Example

- Modula-3 language: Design deliberately limited to 50 pages

Counter-Example

Python

```
def d(x):
    r = x[::-1]
    return x == r
```

Orthogonality

- Features can be combined freely
- Minimal overlap between features

Example

- loops / conditionals may contain other loops / conditionals
- Many functional languages: 'Everything is a value'

Counter-Example

C

```c
// global variable section

float f1 = 2.0f * 2.0f;
float f2 = sqrt(2.0f); // error
```

Syntax Design

Example

C

```
if (cond)
  print(a);
  print(b);
```

⇓

Go

```
if cond {
  print(a);
  print(b);
}
```

Counter-Example

Fortran 95

```
program hello
  implicit none
  integer end, do
  do = 0
  end = 10
  do do=do,end
    print *,do
  end do
end program hello
```

Data Types

- Datatypes can communicate intent
- Possibly enforce checking

Java

```java
enum Color {
  Red, Green, Blue
};
...
Color c = readColorFromUser();
```

Summary: Readability Characteristics

- **Readability** helps us understand code
- Core *characteristics*:
  - Simplicity
  - Orthogonality
  - Syntax Design
  - Datatypes

Criterion: Writability

- How easy is it to write software in the language?
- Characteristics that contribute to **Readability** contribute to **Writability**
- Further criteria for **Writability**:
  - **Support for Abstraction**
    - over values (via variables)
    - over expressions (via functions)
    - over statements (via subprograms)
    - over types...
  - **Expressivity**

Criterion: Reliability

- How easy is it to write *reliable* software in the language?
- Criteria that contribute to **Readability** or **Writability** also contribute to **Reliability**
- Further criteria:
  - **Type Checking**
    - The language prevents *type errors* ($\rightarrow$ in two weeks)
  - **Exception Handling**
    - The language allows errors during execution to be systematically escalated ($\rightarrow$ in four weeks)
  - **Restricted Aliasing**

Restricted Aliasing

Java

```
public static <T> void
concat(List<T> lhs, List<T> rhs) {
    for (int i = 0; i < rhs.size(); i++) {
        lhs.add(rhs.get(i));
    }
}

concat(a, a);
```

- Attach rhs to the end of lhs
- This code misbehaves (infinite loop) when passed the same list for both parameters
- **Aliasing**: two different names mean the same thing

Criterion: Cost

- **Cost** explains the investment needed to use a language:
  - Training time
  - Programming time
  - Compilation time
  - Run time
  - Financial cost of special software
  - Cost of limited reliability
    - Maintenance time
    - Insurance cost

Language Evaluation Summary

	Readability	**Writability**	**Reliability**
Simplicity	+	+	+
Orthogonality	+	+	+
Types	+	+	+
Syntax Design	+	+	+
Abstraction Support		+	+
Expressivity		+	+
Type Checking			+
Exception Handling			+
Restricted Aliasing			+

(this is Robert W. Sebesta, "Concepts of Programming Languages", Table 1.1)

- Separate dimension: **Cost**
- Alternative (more detailed) model: Green and Petre, "Cognitive Dimensions of Notation"

Describing Languages

- Program structure
- Program meaning
  - Well-formedness
  - Runtime behaviour

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.'
  - $\langle expr \rangle \longrightarrow nat$
- 'Two expressions with a + between them form an expression.'
  - $\langle expr \rangle \longrightarrow \langle expr \rangle + \langle expr \rangle$
- 'Two expressions with a $*$ between them form an expression.'
  - $\langle expr \rangle \longrightarrow \langle expr \rangle * \langle expr \rangle$

Or in short:

$$\langle expr \rangle \longrightarrow nat \mid \langle expr \rangle + \langle expr \rangle \mid \langle expr \rangle * \langle expr \rangle$$

- We call *nat*, +, $*$ *terminals*
- We call $\langle expr \rangle$ a *nonterminal*
  Nonterminals can appear on left-hand side of ($\longrightarrow$)

Backus-Naur Form: Example

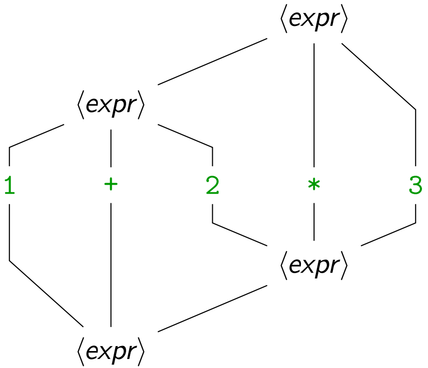
$$\langle \mathit{expr} \rangle \longrightarrow \mathit{nat} \mid \langle \mathit{expr} \rangle \texttt{+} \langle \mathit{expr} \rangle \mid \langle \mathit{expr} \rangle \texttt{*} \langle \mathit{expr} \rangle$$

(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:

$$\begin{array}{lcl} \langle \mathit{expr} \rangle & \longrightarrow & \mathit{nat} \\ & | & \langle \mathit{expr} \rangle \texttt{+} \langle \mathit{expr} \rangle \\ & | & \texttt{ifnz} \langle \mathit{expr} \rangle \texttt{then} \langle \mathit{expr} \rangle \texttt{else} \langle \mathit{expr} \rangle \end{array}$$

Examples:

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

Meaning of our Toy Language: Examples

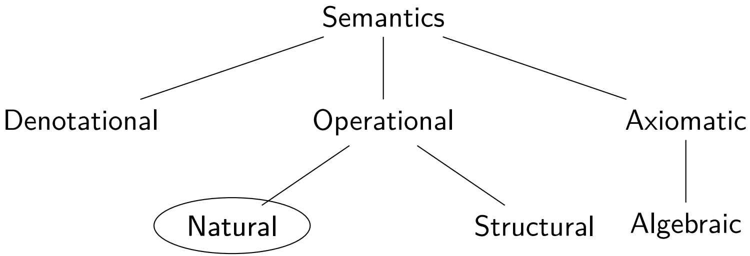
What we want the meaning to be:

`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:



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 Semantics of our simple toy language

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

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

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

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

Note:

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

Natural Semantics: Example

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

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)

Extending our language with 'let'

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

$$
\begin{array}{rcl}
\langle \textit{expr} \rangle & \longrightarrow & \textit{nat} \\
& | & \langle \textit{expr} \rangle \texttt{+} \langle \textit{expr} \rangle \\
& | & \texttt{ifnz} \langle \textit{expr} \rangle \texttt{then} \langle \textit{expr} \rangle \texttt{else} \langle \textit{expr} \rangle \\
& | & \textit{name} \\
& | & \texttt{let}\ \ \textit{name}\ \texttt{=} \langle \textit{expr} \rangle \texttt{in} \langle \textit{expr} \rangle
\end{array}
$$

Example:

$$\texttt{let x = 2 + 3 in x + x} \Downarrow 10$$

But how can we describe $x \Downarrow \ldots$ by itself?

Environments

- With variables, the meaning of program depends on their *environment*

$$\boxed{\textbf{Environment: } E : \mathtt{name} \rightarrow \mathtt{value}}$$

- Environments are *partial functions* from names to 'values'
- In our running example, $\mathtt{value} = \mathtt{nat}$

Notation:

$E(x)$	look up value for x
$E[x \mapsto v]$	update environment E, x maps to v

$$E[x \mapsto v](y) = \begin{cases} v & \iff \quad y = x \\ E(y) & \qquad \textit{otherwise} \end{cases}$$

Environments in Natural Semantics

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

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

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

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

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

$$\frac{E \vdash e_1 \Downarrow v \quad (E[x \mapsto v]) \vdash e_2 \Downarrow v'}{E \vdash \texttt{let}\ x\texttt{=}e_1\ \texttt{in}\ e_2 \Downarrow v'}\ (let)$$

Summary

- Natural Semantics describe program behaviour through reduction rules
- Analogous to interpreters
- Notation: $p \Downarrow v$
  - p: program
  - v: value (cannot be reduced further)
- Uses inference rules:

$$\frac{\textit{preconditions}}{p \Downarrow v}$$

- Can pass extra parameters (e.g., environment for variable bindings):

$$A, B, C \vdash p \Downarrow v$$

- Requires well-formed program

Program Well-Formedness

- Consider the program `a + b`
  - E(`a`) and E(`b`) will be undefined
  - Compiler would issue error message
- Other examples:
  - References to modules that don't exist
  - Type errors
  - Function definition without **return** statement
- *Static semantics*: analysis and error-checking before execution

Describing Languages revisited

- Program structure: **Syntax**
- Program meaning: **Semantics**
  - Well-formedness: **Static Semantics**
  - Runtime behaviour: **Dynamic Semantics**

Daily Summary

- Languages vs. Language Implementations
- Implementation tyoes
  - **Interpreter**, **Compiler**, **Hybrid Implementation**
- Language evaluation criteria:
  - **Readability**, **Writability**, **Reliability**, **Cost**
  - Various characteristics contribute to the criteria
- **Syntax**: Backus-Naur Form (BNF)
- **Semantics**: Program behaviour
  - **Static**: Well-formedness
  - **Dynamic**: Run-time behaviour (only for well-formed code)

Next Week

- Syntax
- Variables, Binding, Scope
- Semantics
- Basic Expressions
- Primitive Types

Read the listed parts of the book, bring your questions!