

Programming Languages Lecture 1

Syntax, Types of Semantics, Impcore

Allyn Dimock

U. Mass. Lowell

Material in these notes is from lecture notes, handouts, and texts by Norman Ramsey, Robert Harper, John Reynolds, and Hanna Nielson and Fleming Nielson.

91.531 Design of Programming Languages

`http://cs.uml.edu/~dimock/courses/languages/
Spring2005/index.html`

Lecture: Thursday 5:30 – 8:30, Olsen 414

Instructor: Allyn Dimock, Olsen 215

Office Hours: T 3–4, F 11 – 1

Email: `dimock@cs.uml.edu`

TA: ??

What are programming languages for?

Express computations

- precisely,
- at a high level,
- in a way we can reason about them.

Why study programming languages?

- Learn new ways of thinking about programming
 - “language shapes thought” — Whorf
- Writing programs is fundamental
- Learn how language can help or hinder
- Learn to write what you mean
- Become a sophisticated, skeptical consumer

91.531 Agenda

Intellectual tools to understand & evaluate languages

- Language features

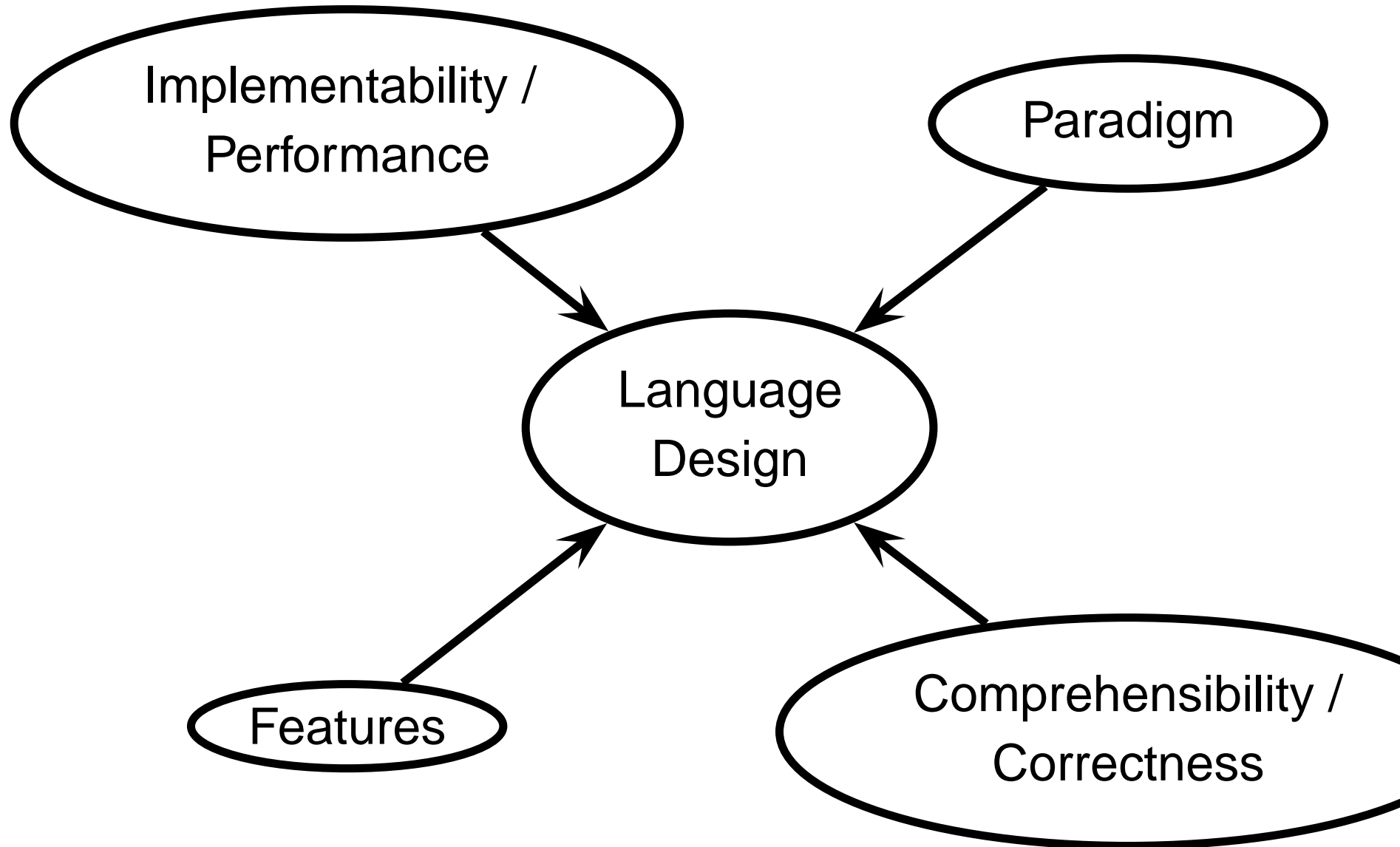
Learn the notations of the trade

- Precise ways to model languages
- Foundation for further study

Learn by doing

- Write lots of (mostly short) programs
- Some math
- Write / modify features of interpreters

Language Design



Language Design

- Do we design a language in this course?

No!

Think of the size of the term project...

- Cover basic paradigms, Imperative, Functional, OO, Logic in textbook and handouts.
- Cover lots of features.
- Efficient implementation is topic of 91.534 (Compilers I), but we modify interpreters.
- Comprehensibility and Correctness.
 - Comprehensibility: Can you mentally single-step through the process generated from a program and know what it does?
 - Correctness: Can you prove you were right?

Study of Language as Study of Features

Language features influence code (Whorf)

Choose abstractions (languages) to fit needs

Build your vocabulary (add to your toolbox)

- Higher-order functions
- Polymorphism (reuse)
- Pattern matching for symbolic computing
- Data for symbolic computing: lists, tables, sets
- Abstract datatypes, encapsulation
- Objects and subtyping
- Modules, parameterization
- Searching and backtracking

How to use Features

From the definition of Scheme:

Programming languages should be designed not by piling feature on top of feature, but by removing the weaknesses and restrictions that make additional features appear necessary.

(Larry Wall and Bjarne Stroustrup might not agree.)

Why study weird features?

Some languages more powerful than others

Mistake to use any but the most powerful

Except for: compatibility, libraries

Problem: habit blinds us to power

Happy user of *Blub*: beats Cobol, machine code

Use Haskell, Lisp, or Icon? No! Equivalent to

Blub, plus weird stuff nobody uses

Blub looks good enough because I think in *Blub*

Ramsey and Kamin textbook is a tour of power in languages

The Search for Expressive Power

- Functions
- Types
- Pattern Matching
 - Unification
 - Regular Expressions
- Control constructs

Abstraction, parameterization, correspondence principles:
try to minimize confusion in using power of language.

Influence of Methodology

Programming methodologies, software engineering

- Structured programming
- Abstract data types
- Modules
- Objects and Inheritance
- Components
- Aspects
- ...

Separate compilation/smart recompilation

Influence of Implementation

Techniques

- Parser generators
- Attribute-grammar systems
- Memory allocation
- Garbage collection
- Runtime typing/tagging
- Efficiency concerns
 - fast execution
 - fast compilation
 - fast program construction

(Primary topic of 91.534)

Correctness

Semantics

- Reference (definitional) Interpreter / Compiler
- Operational Semantics: simple models of execution – several varieties
- Denotational Semantics: what does my program mean
- Axiomatic Semantics: how can I prove that my program does what I claim it does

How do these fit together

Administration – Grading

- Homework, quizzes, final.
 - Circa 10 homework assignments.
 - Up to 10 late days allowed, but max of 2 late days per assignment without medical excuse.
 - Some written answers, some programming, some opinion questions.
 - Conciseness and clarity count.
- Don't use proprietary formats: instructor can read (most) handwriting in English, can read .ps, .pdf. Instructor can *not* read most other formats including .doc.

Weights of assignments may be adjusted at instructor's discretion.

Administration – Working together

Limited collaboration is encouraged.

- Use the class discussion board for discussion of problems, techniques, ideas.

`http://teaching.cs.uml.edu/cgi-bin/ikonboard/ikonboard.cgi`

“Register” button on top left of screen.

- If discussing in person, post results on class discussion board.
- Discuss only problems, techniques, ideas.
- All in person discussions must be acknowledged.

Must not collaborate on answers or code.

- Must not even see other’s answers or code.

Some answers can be found in library / on WWW.

- Don’t overuse.
- *Must* be acknowledged with bibliographical citation.

Administration – Computing

Course software runs on `mercury.cs.uml.edu` (x86 Linux).

Any software not generally installed can be found in
`~dimock/public_html/courses/languages/
software/`

Programming in: C, SML, languages from Ramsey & Kamin, Prolog.

Prerequisites

Good programming skills:

- 91.301 or equivalent (*Structure and Interpretation of Computer Programs* or *Schematics of Computation*).
- Will be modifying other's code.
- Will learn SML as needed.

Unix.

Good basic math skills:

- Sets, relations, and functions.
- Proof by induction.
- Basic logic.

Readings

Ramsey & Kamin: A sequence of languages illustrating concepts. Interpreters in C or SML. Evaluation semantics as a formalism. Small section on program correctness.

“Foundations of Functional Programming” by Lawrence C. Paulson. Good intro to λ -calculus in circa 25 pages (out of circa 50).

Introduction to Denotational Semantics: Excerpts from “Lecture Notes on Denotational Semantics” by Andrew M. Pitts.

Learning SML: Ullman: Elements of ML Programming

Notes on Hoare Logic: Andrew J.C. Gordon.

Syllabus

1. Notation, concrete and abstract syntax.
2. A core imperative language (Ch 1&2)
3. Scheme, recursive programming (Ch 3)
4. The Lambda Calculus (Paulson)
5. Intro to SML (Ullman, Ch 5)
6. Type systems (Ch 6, Cardelli)
7. ML type inference (Ch 7)
8. Intro to denotational semantics (Pitts)
9. Prolog and logic programming (Ch 10, handout, Periera & Shieber)
10. Data abstraction and CLU (Ch 8, handout)
11. Program correctness (Sec. 8.6, Gordon)
12. Smalltalk and OO programming (Ch 9)
13. Module systems (handout)
14. Relating semantics (handout)

Another breakdown of material

- Writing about programs...
 - Longhand: Judgments, a way of constructively proving
 - what a program computes (operation semantics)
 - the type of an expression (type checking / inference)
 - Turning judgments into code:
Operational semantics becomes interpreters.
Type checking becomes type checkers.
- Paradigms: Imperative, Functional, Logic Programming, Object Oriented.
Little programs illustrating coding techniques in these paradigms.

Another breakdown of material

- Pure paradigm (= a logic) → language (= a logic + features + compromises for implementability)
 - λ-calculus → Scheme
 - First-order logic → Prolog
- Grab bag of techniques and features:
 - What is a variable?
 - Function calling standards?
 - Exceptions and control
 - Type systems
 - Denotational semantics: another way of determining what a program computes.
 - Support for abstract data types.
 - Showing correctness of ADTs, code.
 - Module systems.

Syntax, Rules, and all that

Syntax, Rules, and all that

A **Judgment** is a formal statement:

zero nat

A judgment can contain variables (“meta-variables”)

When using judgments, meta-variables pattern match against phrases.

Syntax, Rules, and all that

Rules / Rule schemas:

$$\frac{\text{premises (judgments, conditions)}}{\text{conclusion (judgment)}} \quad (\text{NAME OF RULE})$$
$$\frac{}{\text{judgment}} \quad (\text{NAME OF AXIOM})$$

In case of an axiom, often omit overbar.

Proof trees: A judgment that is the the premise for one use of a rule can be the conclusion of another rule, or an axiom.

Syntax, Rules, and all that

What is a natural number?

one axiom, one rule. x is a “meta-variable”

$$\frac{}{\text{zero nat}} \quad (\text{Zero}) \qquad \frac{x \text{ nat}}{\text{succ}(x) \text{ nat}} \quad (\text{Successor})$$

A proof tree: 2 is a natural number!

$$\frac{\frac{\frac{}{\text{zero nat}}}{\text{succ}(\text{zero}) \text{ nat}}}{\text{succ}(\text{succ}(\text{zero})) \text{ nat}}$$

Concrete syntax, Grammars (BNF)

$N ::= \text{zero} \mid \text{succ} (N)$

Deriving 2 from our 1-rule grammar

$N \Rightarrow \text{succ}(N) \Rightarrow \text{succ}(\text{succ}(N)) \Rightarrow \text{succ}(\text{succ}(\text{zero}))$

⟨ Terminals, Nonterminals, Productions, Start nonterminal ⟩

Terminals = {zero, succ, (,)}

Nonterminals = { N }

Productions = {

$N ::= \text{zero}$,

$N ::= \text{succ} (N)$

}

Start nonterminal = N

Formally, Production is pair of Nonterminal, string of Terminals and Nonterminals. Write ε for the empty string.

Concrete syntax (BNF)

Backus-Naur form has abbreviations from formal definition:

- Just write productions: The start nonterminal is the nonterminal on the left of the first production.
- Combine productions with same left-hand side with “|”.
- Nonterminals are the symbols on left-hand sides, all symbols that are not on left-hand sides are terminals.

Concrete syntax (EBNF)

Extended Backus-Naur form has more abbreviations:

- “one of” $A ::= \alpha (\beta_1 \cdots \beta_n) \gamma$ expands to:

$$A ::= \alpha A' \gamma$$

$$A' ::= \beta_1 \mid \cdots \mid \beta_n$$

- “zero or one” $A ::= \alpha [\beta] \gamma$ expands to:

$$A ::= \alpha A' \gamma$$

$$A' ::= \beta \mid \varepsilon$$

Equivalently $A ::= \alpha \beta \gamma \mid \alpha \gamma$

- “zero or more” $A ::= \alpha \{\beta\} \gamma$ expands to:

$$A ::= \alpha A' \gamma$$

$$A' ::= \beta A' \mid \varepsilon$$

(Book's notation, not standardized)

Syntax is rules!

$$\begin{aligned} N & ::= \text{zero} \\ N & ::= \text{succ} (N) \end{aligned}$$

$$\frac{}{\text{zero isaN}}$$

$$\frac{x \text{ isaN}}{\text{succ} (x) \text{ isaN}}$$

Idea: replace nonterminal N on left-hand side of a production with assertion / conclusion that the phrase “is an N ”.

Replace each occurrence of nonterminal on the right-hand side of a production with a variable. For each variable have a premise that the variable “is a M ” for whatever nonterminal M is replacing.

Abstract syntax

$E ::= \text{true} \mid \text{false} \mid \dots$

$S ::= \{\} \mid \text{if } E \text{ then } S \text{ else } S \mid \text{if } E \text{ then } S$

$\text{if true then if true then } \{\} \text{ else } \{\}$

which if does the else go with?

Abstract syntax is any way of writing out syntax so there is no ambiguity.

Functional notation: (abstract syntax in book)

$S ::= \text{emptyblock}() \mid \text{ifthenelse}(E, S, S) \mid$
 $\text{ifthen}(E, S)$

Parenthesized prefix notation (concrete syntax in book):

$S ::= \text{emptyblock} \mid (\text{if } E \ S \ S) \mid (\text{if } E \ S)$

Tree notation (parse trees) ...

Semantics

Denotation Semantics: meaning

$$D ::= 0 \mid 1 \mid 2 \mid \dots \mid 9$$

$$N ::= D \mid N D$$

$$E ::= N \mid (E + E)$$

Let d be a variable ranging over digits D . Let i, j, k be variables ranging over numerals N . Let a, b be variables ranging over expressions E .

Functions from strings to meanings, defined compositionally.

$$\mathcal{D}[[0]] = 0 \dots \mathcal{D}[[9]] = 9 \quad (\text{maps digits to numbers})$$

$$\mathcal{N}[[d]] = \mathcal{D}[[d]] \quad (\text{maps numerals to numbers})$$

$$\mathcal{N}[[i d]] = \mathcal{N}[[i]] \times 10 + \mathcal{D}[[d]]$$

$$\mathcal{E}[[i]] = \mathcal{N}[[i]] \quad (\text{maps expressions to numbers})$$

$$\mathcal{E}[[a + b]] = \mathcal{E}[[a]] + \mathcal{E}[[b]]$$

$$\mathcal{E}[[40 + 2]] = \mathcal{E}[[40]] + \mathcal{E}[[2]] = \mathcal{N}[[40]] + \mathcal{N}[[2]]$$

$$= \mathcal{N}[[4]] \times 10 + \mathcal{D}[[0]] + \mathcal{D}[[2]] = \mathcal{D}[[4]] \times 10 + \mathcal{D}[[0]] + \mathcal{D}[[2]] =$$

$$4 \times 10 + 0 + 2 = 42$$

Operational Semantics: Transition system

A *Transition system* $\langle S, F, \rightsquigarrow \rangle$ is a relation where:

- a set of *states*: S
- a subset of *final states*: $F \subseteq S$
- a relation symbol \rightsquigarrow
that maps from states in S to states in S .

$s_0 \rightsquigarrow s_1 \rightsquigarrow s_2 \rightsquigarrow \dots \rightsquigarrow f_n \not\rightsquigarrow$

If $s \not\rightsquigarrow$ and $s \notin F$ we call s a *stuck state*.

Often, but not always, a transition system will not allow transitions out of final states.

Transition semantics

(Structural operational semantics, small-step semantics).

The **states** are expressions

The **final states** are values

The **relation** is defined by rules.

Axioms and rules for calculating one step at a time:

$$\frac{}{(i + j) \Rightarrow k} \quad \text{where } \mathcal{N}[[i]] + \mathcal{N}[[j]] = \mathcal{N}[[k]]$$

$$\frac{a \Rightarrow a'}{(a + b) \Rightarrow (a' + b)}$$

$$\frac{b \Rightarrow b'}{(i + b) \Rightarrow (i + b')}$$

Transition Semantics for $((1+2)+3)+4$

$$\frac{\frac{\overline{(1 + 2) \Rightarrow 3}}{\overline{((1 + 2) + 3) \Rightarrow (3 + 3)}}}{\overline{(((1 + 2) + 3) + 4) \Rightarrow ((3 + 3) + 4)}}$$

then

$$\frac{\overline{(3 + 3) \Rightarrow 6}}{\overline{((3 + 3) + 4) \Rightarrow (6 + 4)}}$$

then

$$\overline{(6 + 4) \Rightarrow 10}$$

Evaluation Semantics

(Natural semantics, big-step semantics).

Axioms and rules for calculating based on sub-calculations:

$$\frac{}{i \Downarrow i} \quad \frac{a \Downarrow i \quad b \Downarrow j}{(a + b) \Downarrow k} \quad \text{where } \mathcal{N}[[i]] + \mathcal{N}[[j]] = \mathcal{N}[[k]]$$

$$\frac{\frac{\frac{1 \Downarrow 1 \quad 2 \Downarrow 2}{(1 + 2) \Downarrow 3} \quad 3 \Downarrow 3}{((1 + 2) + 3) \Downarrow 6} \quad 4 \Downarrow 4}{(((1 + 2) + 3) + 4) \Downarrow 10}$$

Impcore: a core imperative language