编程语言的设计原理
Design Principles of Programming Languages

Haiyan Zhao, Di Wang

趙海燕，王迪

Peking University, Spring Term 2025

Teaching Team

Instructors

Teaching Assistant

Instructors

- Haiyan Zhao （赵海燕）
  - 1988, BS, Peking Univ.
  - 1991, MS, Peking Univ
  - 2003, PhD, Univ. of Tokyo
  - 2003-, Assoc. Professor, Peking Univ.
- Research Interest
  - Software engineering
  - Requirements Engineering, Domain Engineering
  - Programming Languages
- Contact
  - Office: Rm. 1809, Science Blg #1, Yanyuan / Rm 432, CS Blg, Changping
  - Phone： 62757670
  - Email： zhhy.sei@pku.edu.cn

Instructors

- Di Wang （王迪)
  - 2017, BS, Peking Univ.
  - 2022, PhD, Carnegie Mellon Univ.
  - 2022-, Assistant Professor, Peking Univ.
- Research Interests
  - Programming Languages
  - Quantitative Program Analysis and Verification
  - Probabilistic Programming
- Contact
  - Office: Rm. 520, Yanyuan Mansion
  - Tel: 62757242
  - Email: wangdi95@pku.edu.cn
  - Webpage: https://stonebuddha.github.io

Teaching Assistant

- Qihao Lian (练琪灏)
  - PhD student from Programming Languages Lab, PKU

- Contact
  - mepy@stu.pku.edu.cn

- Homework submit to
  - pku-dppl@outlook.com
  - Each assignment should be submitted before 0:00 AM (midnight) on the following Monday, and please try to submit all assignments for each week in a single email, and indicate your student ID, name, and the week number of the assignment in the email subject (in the format of “2100012345-John-1”)

Information

- Course website: http://pku-dppl.github.io/2025
  - Syllabus
  - News/Announcements
  - Lecture Notes (slides)
  - Other useful resources
  - Projects

- Time：Monday 7-9（15:10-18:00）
- Place：昌平教学楼 109

群聊：2025春季DPPL课程

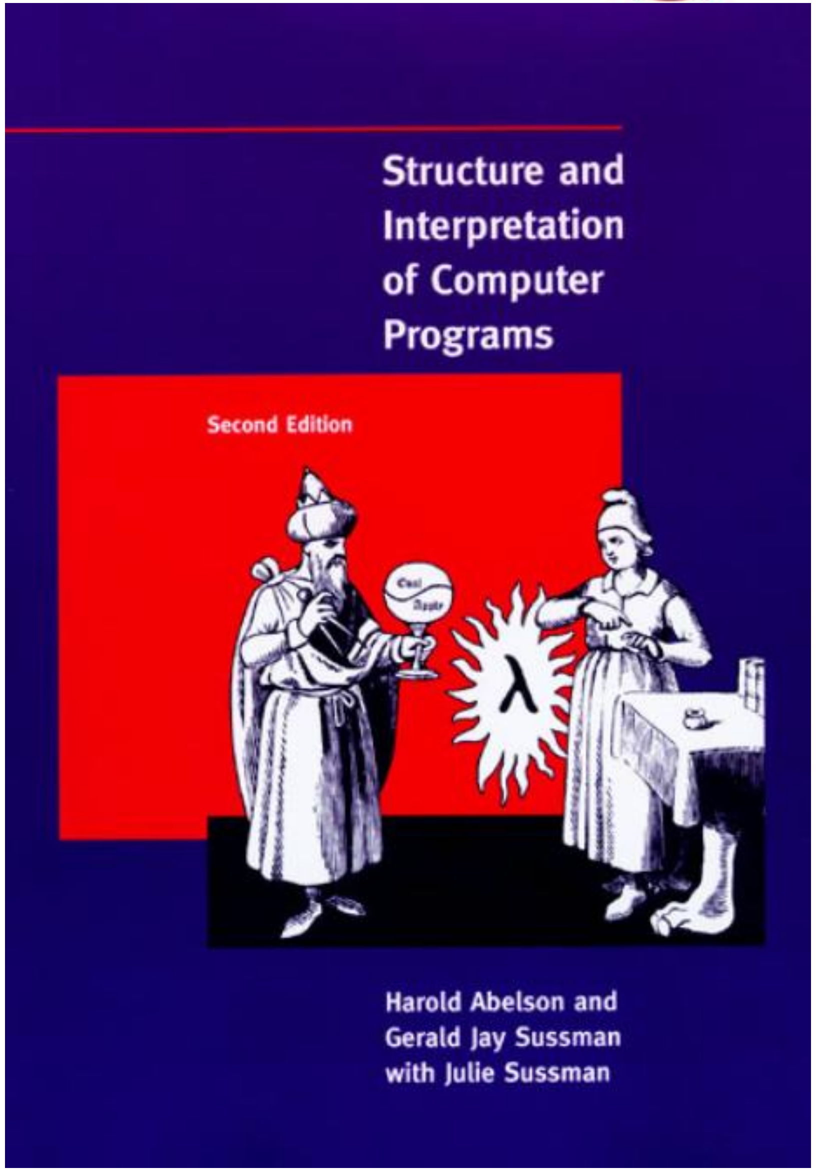
该二维码7天内(2月24日前)有效，重新进入将更新

Course Overview

Computer Science vs PL Construction

System = Specification + Program

" . . . the technology for coping with *large-scale computer systems* merges with the technology for *building new computer languages*, and *computer science itself* becomes no more (and no less) than the discipline of *constructing appropriate descriptive languages* "



Isn’t PL a solved problem?

- A fundamental area within CS
  - ……
  - 1930’s:
  - 1940’s:
  - 1950’s
  - 1960’s:
  - 1970’s：
  - 1980’s：
  - 1990’s：
  - 2000’s：
  - ……

Isn’t PL a solved problem?

- A fundamental area within CS
  - 1930’s: lambda-calculus
  - 1940’s:
  - 1950’s: Fortran, LISP, COBOL, …
  - 1960’s: ALGOL60, PL/1, ALGOL68, …
  - 1970’s：C, Pascal, Smalltalk, MODULA, Scheme, ML, …
  - 1980’s： Ada, C++, …
  - 1990’s： Java, …
  - 2000’s： Rust, …
  - ……

Programming Languages

- Touches most other areas of CS
  - Theory:
  - Systems:
  - Arch:
  - Numeric
  - DB:
  - Networking:
  - Graphics:
  - Security:
  - Software Engineering:
  - ….
- Both *theory*(math) and *practice* (engineering)

Programming Languages

- Touches most other areas of CS
  - Theory: DFAs, TMs, ….
  - Systems: system calls, memory management , …
  - Arch: compiler targets. Optimizations, stack frames , …
  - Numeric: FORTRAN, matlab , …
  - DB: SQL , …
  - Networking: packet filter. protocols , …
  - Graphics: OpenGL, LaTeX, PostScript , …
  - Security: buffer overruns, .net, bytecode , …
  - Software Engineering: bug finding, refactoring, types, …
  - ….
- Both *theory* (math) and *practice* (engineering)

This course is not about …

- An introduction to programming
- A course on compiler
- A course on functional programming
- A course on language paradigms/styles

All the above are certainly helpful for your deep understanding of this course.

What is this course about?

- Study fundamental (formal) approaches to describing *program behaviors* that are both *precise* and *abstract*.
  - precise so that we can use mathematical tools to *formalize and check* interesting *properties*
  - abstract so that properties of interest can be *discussed clearly*, *without getting bogged down* in low-level details

What you can get out of this course?

- A more *sophisticated perspective* on programs, programming languages, and the activity of programming
  - How to *view programs and whole languages* as *formal, mathematical objects*
  - How to *make and prove rigorous claims* about them
  - Detailed *study* of a range of *basic language features*

- Powerful tools/techniques for language design, description, and analysis

What background is required?

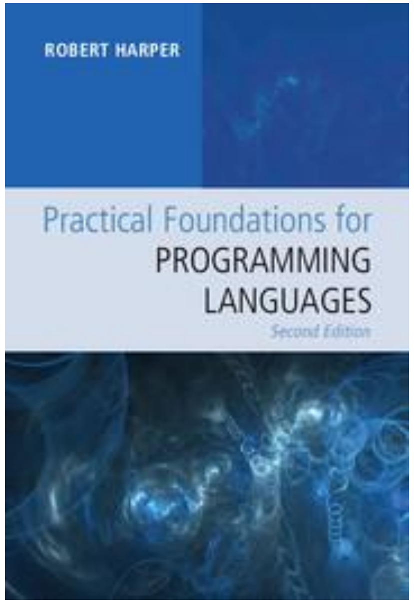
- Basic knowledge on
  - Discrete mathematics: sets, functions, relations, orders
  - Algorithms: list, tree, graph, stack, queue, heap
  - Elementary logics: propositional logic, first-order logic

- Familiar with a *programming language* and basic knowledge of *compiler construction*

Textbook & Reference

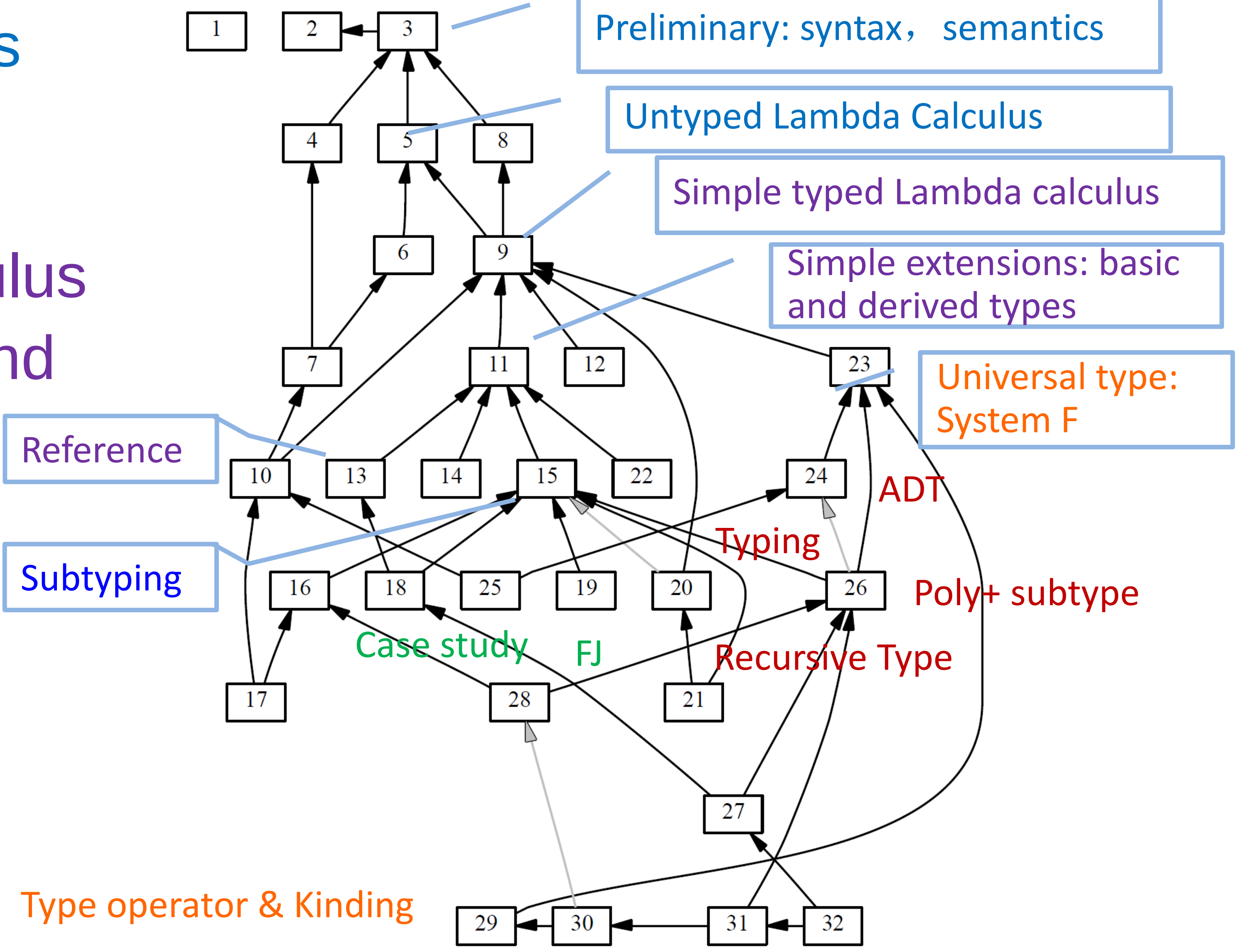
- Types and Programming Languages
  - Benjamin Pierce
  - The MIT Press, 2002

- Practical Foundations for Programming Languages (Second Edition)
  - Robert Harper
  - Cambridge University Press, 2016
  - https://www.cs.cmu.edu/~rwh/pfpl/



Outline

- Basic operational semantics and proof techniques
- Untyped Lambda calculus
- Simply typed Lambda calculus
- Simple extensions (basic and derived types)
- References
- Exceptions
- Subtyping
- Recursive types
- Polymorphism
- [Higher-order systems]



Outline

- Basic operational semantics and proof technique
- Untyped Lambda calculus
- Simple typed Lambda calculus
- Simple extensions (basic and derived types)
- References
- Exceptions
- Subtyping
- Recursive types
- Polymorphism
- [Higher-order systems]

Class	Date	Topic and Readings	Lecturer
1	17-Feb	Introduction Untyped Arithmetic Operations	Zhao
2	24-Feb	OCaml/MoonBit	Zhao/Wang
3	3-Mar	Lambda Calculus Nameless Representation	Zhao
4	10-Mar	Type Basics Simply Typed Lambda Calculus Simple Extensions	Zhao
5	17-Mar	Reference	Zhao
6	24-Mar	Exception	Zhao
7	31-Mar	Subtyping Metatheory of Subtyping	Zhao
8	7-Apr	Project Proposal (Midterm Test)	Zhao/Wang
9	14-Apr	Recursive Types	Wang
10	21-Apr	Variable Types	Wang
11	28-Apr	Type-Level Computation	Wang
12	5-May	May Festival (No Class)	/
13	12-May	Type Inference	Wang
14	19-May	Substructural Types	Wang
15	26-May	Effect Types	Wang
16	2-Jun	Project Final Presentation	Zhao/Wang

Grading

- Homework (+ Activity in class + take home quiz) : 40 %
- Course project : 60%

You will design and implement a *typed programming language* with certain features.

You are encouraged to draw inspiration from popular or emerging languages, such as Rust, Go, TypeScript, Elm, Scala, Haskell, OCaml, MoonBit, Koka, Crystal, and Zig.

As a course project, you need to choose *a key feature* that interests you, formalize a core calculus for it (ideally based on lambda calculus), prove its soundness (at least roughly), and implement a prototype (ideally based on the checkers here from our course material) with *an interpreter* and *a type checker*.

Grading

- Homework (+ Activity in class + take home quiz) : 40 %
- Course project : 60%
  - At the end of the semester, you will give a presentation about your work and submit an artifact containing the following:
    - A document that includes your motivation, illustrative examples, a formalization of the core calculus, and a soundness proof.
    - A prototype implementation of your language along with a suite of benchmark programs.

You will design and implement a *typed programming language* with certain features. You are encouraged to draw inspiration from popular or emerging languages, such as Rust, Go, TypeScript, Elm, Scala, Haskell, OCaml, MoonBit, Koka, Crystal, and Zig.

As a course project, you need to choose *a key feature* that interests you, formalize a core calculus for it (ideally based on lambda calculus), prove its soundness (at least roughly), and implement a prototype (ideally based on the checkers here from our course material) with *an interpreter* and *a type checker*.

Grading

- Homework (+ Activity in class + take home quiz) : 40 %
- Course projects : 60%
  - Here are some example projects from previous cohorts of students:

设计一个带类型系统的程序语言，解决实践中的问题，给出基本实现

- 设计一个语言，保证永远不会发生内存/资源泄露。
- 设计一个汇编语言的类型系统
- 设计一个没有停机问题的编程语言
- 设计一个嵌入复杂度表示的类型系统，
  保证编写的程序的复杂度不会高于类型标示的复杂度。
- 设计一个类型系统，使得敏感信息永远不会泄露。
- 设计一个类型系统，使得写出的并行程序没有竞争问题
- 设计一个类型系统，保证所有的浮点计算都满足一定精度要求
- 解决自己研究领域的具体问题

Grading

- Homework (+ Activity in class + take home quiz) : 40 %
- Course projects : 60%
  - several potential projects you might consider
  - https://pku-dppl.github.io/2025/projects.html
    - Project 1: Extensible Records
    - Project 2: Gradual Typing
    - Project 3: Typeclass or Trait
    - Project 4: Functional In-place Update
    - Project 5: Refinement Types
    - Project 6: Asynchronous Programming

How to study this course?

- **Before class**: scanning through the chapters to learn and gain feeling about what will be studied
- **In class**: trying your best to understand the contents and *raising hands when you have questions at any time*
  - Discussion / lecture
- **After class**: doing exercises seriously

★	Quick check	30 seconds to 5 minutes
★★	Easy	≤ 1 hour
★★★	Moderate	≤ 3 hours
★★★★	Challenging	> 3 hours

Chapter 1
Introduction

What is a type system

What type systems are good for

Type systems and programming languages

Type system in PL (CS)

1870s	*origins of formal logic*	Frege (1879)
1900s	*formalization of mathematics*	Whitehead and Russell (1910)
1930s	*untyped lambda-calculus*	Church (1941)
1940s	*simply typed lambda-calculus*	Church (1940), Curry and Feys (1958)
1950s	Fortran	Backus (1981)
	Algol-60	Naur et al. (1963)
1960s	*Automath project*	de Bruijn (1980)
	Simula	Birtwistle et al. (1979)
	Curry-Howard correspondence	Howard (1980)
	Algol-68	(van Wijngaarden et al., 1975)
1970s	Pascal	Wirth (1971)
	Martin-Löf type theory	Martin-Löf (1973, 1982)
	System F, F^ω	Girard (1972)
	polymorphic lambda-calculus	Reynolds (1974)
	CLU	Liskov et al. (1981)
	polymorphic type inference	Milner (1978), Damas and Milner (1982)
	ML	Gordon, Milner, and Wadsworth (1979)
	intersection types	Coppo and Dezani (1978)
		Coppo, Dezani, and Sallé (1979), Pottinger (1980)
1980s	NuPRL project	Constable et al. (1986)
	subtyping	Reynolds (1980), Cardelli (1984), Mitchell (1984a)
	ADTs as existential types	Mitchell and Plotkin (1988)
	calculus of constructions	Coquand (1985), Coquand and Huet (1988)
	linear logic	Girard (1987) , Girard et al. (1989)
	bounded quantification	Cardelli and Wegner (1985)
		Curien and Ghelli (1992), Cardelli et al. (1994)
	Edinburgh Logical Framework	Harper, Honsell, and Plotkin (1992)
	Forsythe	Reynolds (1988)
	pure type systems	Terlouw (1989), Berardi (1988), Barendregt (1991)
	dependent types and modularity	Burstall and Lampson (1984), MacQueen (1986)
	Quest	Cardelli (1991)
	effect systems	Gifford et al. (1987), Talpin and Jouvelot (1992)
	row variables; extensible records	Wand (1987), Rémy (1989)
		Cardelli and Mitchell (1991)
1990s	higher-order subtyping	Cardelli (1990), Cardelli and Longo (1991)
	typed intermediate languages	Tarditi, Morrisett, et al. (1996)
	object calculus	Abadi and Cardelli (1996)
	translucent types and modularity	Harper and Lillibridge (1994), Leroy (1994)
	typed assembly language	Morrisett et al. (1998)

What is a type system (type theory)?

- A *type system* is a tractable syntactic method for proving the *absence of certain program behaviors* by classifying phrases according to the kinds of values they compute.
  - Tools for program reasoning
  - Classification of terms
    - according to the properties of the values that the terms (syntactic phrases) will compute when executed.
  - Static approximation
    - calculating a kind of static approximation to the run-time behaviors of the terms
  - Proving the absence rather than presence of bad program behaviors
    - Being static, type systems are necessarily conservative, and the tension between conservativity and expressiveness is a fundamental fact of life in the design of type systems
    - only guarantee that well-typed programs are free from certain kinds of misbehavior
  - Fully automatic (and efficient)
    - Typecheckers are typically built into compilers or linkers

What are type systems good for?

- Detecting Errors
  - Many programming errors can be detected early, fixed intermediately and easily.
  - Errors can often be pinpointed more accurately during typechecking than at run time
  - Expressive type systems offer numerous “tricks” for encoding information about structure in terms of types.
- Abstraction
  - Type systems form the backbone of the module languages and tie together the components of large systems in the context of large-scale software composition
  - An interface itself can be viewed as “the type of a module” , providing a summary of the facilities provided by the module.
- Documentation
  - Type declarations in *procedure headers* and *module interfaces* constitute a form of (checkable) documentation, which cannot become outdated as it is checked during every run of the compiler.
  - This role of types is particularly important in module signatures.

What are type systems good for?

- Language Safety
  - A safe language is one that protects its own abstractions.
  - Safety refers to the language's ability to guarantee *the integrity* of these abstractions and of higher-level abstractions introduced by the programmer using the definitional facilities of the language.
  - Language safety *is not the same thing* as static type safety, and can be achieved by static checking, but also by run-time checks.
- Efficiency
  - Removal of dynamic checking; smart code-generation.
  - Most high-performance compilers today rely heavily on information gathered by the typechecker during optimization and code-generation phases.

Type Systems and Languages Design

- Language design should go hand-in-hand with type system design.
  - Languages without type systems tend to offer features that make *type-checking difficult or infeasible*.
  - Concrete syntax of typed languages tends to be *more complicated* than that of untyped languages, since type annotations must be taken into account.

In typed languages the type system itself is often taken as the foundation of the design and the organizing principle in light of which every other aspect of the design is considered.

Design Programming Languages

- Simplicity
  - syntax
  - semantics
- Readability
- Safety
- Support for programming large systems
- Efficiency (of execution and compilation)

-- Hints on programming language design by C.A.R. Hoare

Design Programming Languages

- Choose a specific application area
- Make the design committee as small as possible
- Choose some precise design goals
- Release version one of the language to a small set of interested people
- Revise the language definition
- Attempt to build a prototype compiler / to provide a formal definition of the language semantics
- Revise the language definition again
- Produce a clear, concise language manual and release it
- Provide a production quality compiler and distribute it widely
- Write marvelously clear primers explaining how to use the language

-- "*Fundamentals of Programming Languages*" by Ellis Horowitz

Chapter 3
Untyped Arithmetic Expressions

A small language of Numbers and Booleans

Basic aspects of programming languages

Introduction

Grammar

Programs

Evaluation

Grammar (Syntax)

t ::=	terms:
true	*constant true*
false	*constant false*
if t then t else t	*conditional*
0	*constant zero*
succ t	*successor*
pred t	*predecessor*
iszero t	*zero test*

t: *metavaraible* in the right-hand side (non-terminal symbol)

For the moment, the words *term* and *expression* are used interchangeably

Programs and Evaluations

- A *program* in the language is just *a term* built from *the forms* given by the grammar

if false then 0 else 1 (1 = succ 0)

→ 1

iszero (pred (succ 0))

→ true

succ (succ (succ (0)))

→?

iszero pred succ 0

succ succ succ 0

Syntax

Many ways of defining syntax (besides grammar)

Terms, Inductively

The set of terms is the **smallest set** T such that

1. {**true**, **false**, **0**} $\subseteq$ T;
2. if $t_1 \in T$,

   then {**succ** t_1, **pred** t_1, **iszero** t_1} $\subseteq$ T;
3. if $t_1 \in T$, $t_2 \in T$, and $t_3 \in T$,

   then **if** t_1 **then** t_2 **else** $t_3 \in T$.

Terms, by Inference Rules

The set of terms is defined by the following *rules*:

$$\texttt{true} \in \mathcal{T} \qquad \texttt{false} \in \mathcal{T} \qquad 0 \in \mathcal{T}$$

$$\frac{t_1 \in \mathcal{T}}{\texttt{succ } t_1 \in \mathcal{T}} \qquad \frac{t_1 \in \mathcal{T}}{\texttt{pred } t_1 \in \mathcal{T}} \qquad \frac{t_1 \in \mathcal{T}}{\texttt{iszero } t_1 \in \mathcal{T}}$$

$$\frac{t_1 \in \mathcal{T} \qquad t_2 \in \mathcal{T} \qquad t_3 \in \mathcal{T}}{\texttt{if } t_1 \texttt{ then } t_2 \texttt{ else } t_3 \in \mathcal{T}}$$

each rule: *If we have established the statements* in ***the premise(s)*** *listed above the line*, then we may derive ***the conclusion*** below the line

Inference rules = Axioms + Proper rules

Terms, Concretely

For each natural number i, define a set S_i as follows:

$$\begin{array}{rcl} S_0 & = & \emptyset \\ S_{i+1} & = & \{\mathtt{true}, \mathtt{false}, 0\} \\ & & \cup\ \{\mathtt{succ}\ t_1, \mathtt{pred}\ t_1, \mathtt{iszero}\ t_1 \mid t_1 \in S_i\} \\ & & \cup\ \{\mathtt{if}\ t_1\ \mathtt{then}\ t_2\ \mathtt{else}\ t_3 \mid t_1, t_2, t_3 \in S_i\}. \end{array}$$

Finally, let

$$S = \bigcup_i S_i.$$

Exercise [**]: How many elements does S_3 have?

Proposition: T = S

Induction on Terms

Inductive definitions

Inductive proofs

Inductive Definitions

The set of *constants* appearing in a term *t*, written *Consts(t)*, is defined as:

$$
\begin{array}{lcl}
Consts(\mathtt{true}) & = & \{\mathtt{true}\} \\
Consts(\mathtt{false}) & = & \{\mathtt{false}\} \\
Consts(\mathtt{0}) & = & \{\mathtt{0}\} \\
Consts(\mathtt{succ}\ \mathtt{t}_1) & = & Consts(\mathtt{t}_1) \\
Consts(\mathtt{pred}\ \mathtt{t}_1) & = & Consts(\mathtt{t}_1) \\
Consts(\mathtt{iszero}\ \mathtt{t}_1) & = & Consts(\mathtt{t}_1) \\
Consts(\mathtt{if}\ \mathtt{t}_1\ \mathtt{then}\ \mathtt{t}_2\ \mathtt{else}\ \mathtt{t}_3) & = & Consts(\mathtt{t}_1) \cup Consts(\mathtt{t}_2) \cup Consts(\mathtt{t}_3)
\end{array}
$$

Inductive Definitions

The size of a term t, written $size(t)$, is defined as follows:

$$
\begin{aligned}
size(\mathtt{true}) &= 1 \\
size(\mathtt{false}) &= 1 \\
size(\mathtt{0}) &= 1 \\
size(\mathtt{succ}\ \mathtt{t}_1) &= size(\mathtt{t}_1) + 1 \\
size(\mathtt{pred}\ \mathtt{t}_1) &= size(\mathtt{t}_1) + 1 \\
size(\mathtt{iszero}\ \mathtt{t}_1) &= size(\mathtt{t}_1) + 1 \\
size(\mathtt{if}\ \mathtt{t}_1\ \mathtt{then}\ \mathtt{t}_2\ \mathtt{else}\ \mathtt{t}_3) &= size(\mathtt{t}_1) + size(\mathtt{t}_2) + size(\mathtt{t}_3) + 1
\end{aligned}
$$

Inductive Definitions

The *depth* of a term *t*, written *depth(t)*, is defined as follows:

$$
\begin{aligned}
depth(\texttt{true}) &= 1 \\
depth(\texttt{false}) &= 1 \\
depth(\texttt{0}) &= 1 \\
depth(\texttt{succ } t_1) &= depth(t_1) + 1 \\
depth(\texttt{pred } t_1) &= depth(t_1) + 1 \\
depth(\texttt{iszero } t_1) &= depth(t_1) + 1 \\
depth(\texttt{if } t_1 \texttt{ then } t_2 \texttt{ else } t_3) &= \max(depth(t_1), depth(t_2), depth(t_3)) + 1
\end{aligned}
$$

Inductive Proof

Lemma. The number of ***distinct*** *constants* in a term t is no greater than the *size* of t:

$$|\,\mathrm{Consts}(t)\,| \leq \mathrm{size}(t)$$

Proof. By induction over the *depth* of *t.*

- Case t is a constant : $|\mathrm{Consts}(t)| = |\{t\}| = 1 = \mathrm{size}(t)$.
- Case t is pred t_1, succ t_1, or iszero t_1

  By the induction hypothesis, $|\mathrm{Consts}(t_1)| \leq \mathrm{size}(t_1)$, and we have:
  $|\mathrm{Consts}(t)| = |\mathrm{Consts}(t_1)| \leq \mathrm{size}(t_1) < \mathrm{size}(t)$.
- Case t is if t_1 then t_2 else t_3

  ?

Inductive Proof

- Induction on depth/size of terms is analogous to complete induction on natural numbers
- Ordinary *structural induction,* which is used to prove properties pf recursively/inductively defined structures, corresponds to the ordinary natural number induction principle where the induction step requires that P(n+1) be established from just the assumption P(n)
  - it is common practice to use structural induction wherever possible, since it works on terms directly, avoiding the detour via numbers

Inductive Proof

- Ordinary *structural induction,* which is used to prove properties pf recursively/inductively defined structures, corresponds to the ordinary natural number induction principle where the induction step requires that P(n+1) be established from just the assumption P(n)
  - structural induction, wherever possible, works on terms directly

Theorem [Structural Induction]

If, for each term s,

given P(r) for all immediate subterms r of s, we can show P(s),

then P(s) holds for all s.

Suppose P is a predicate on terms, and separately considering *each of the possible forms* that term s could have

Semantic Styles

Three basic approaches

Operational Semantics

- Operational semantics specifies the *behavior* of a programming language by defining a simple abstract machine for it.

- An example (often used in this course):
  - terms as *states,* rather than some low-level microprocessor instruction set
  - behavior : *transition from one state to another* as *simplification*
  - **meaning** of ***t*** is ***the final state*** starting from the state corresponding to ***t***

Denotational Semantics

- The *meaning* of a *term* is taken to be some *mathematical object*, such as a number or a function
  - basically it's related to **mathematical functions,** which take **something as an input,** do some computation that you don't care about and produce **a result,** which you **care about**
- Giving denotational semantics for a language consists of
  - finding a *collection of semantic domains*, and then
  - defining an *interpretation function* mapping *terms* into *elements of these domains*.

- Main advantage: It abstracts from the gritty details of evaluation and highlights *the essential concepts* of the language.

Axiomatic Semantics

- Axiomatic methods take the *laws* (properties) themselves *as the definition of the language.*
  - Instead of first defining the behaviors of programs (by giving some operational or denotational semantics) and then deriving laws from this definition
  - axiomatic semantics is more concerned with specifying the conditions under which programs are correct.

- The meaning of a *term* is just *what* can be proved about it
  - They focus attention on *the process of reasoning* about programs
  - Hoare logic: define the meaning of imperative languages

Axiomatic Semantics

- Key Concepts:
  - Logical Axioms: used to describe the properties of basic language constructs. These axioms serve as the foundation for reasoning about the correctness of programs.
    - e.g., in a language with assignment statements, an axiom might state that if x:=e is executed, the value of x will be the value of e after execution.
  - Inference Rules: used to derive properties of more complex constructs from simpler ones. These rules allow us to build up a proof of correctness for a program by combining the properties of its components.
    - e.g., the rule of composition allows us to combine the properties of two statements executed sequentially.
  - Preconditions and Postconditions: to specify the correctness of programs.
    - A precondition is a logical condition that must be true before a program segment is executed; A postcondition is a logical condition that must be true after the program segment has executed.
    - e.g. , for a statement S with precondition P and postcondition Q, Hoare triplet {P}S{Q} means that if P holds before S is executed, then Q will hold after S is executed.
  - Loop Invariants: a condition that remains true throughout the execution of a loop.
    - e.g. , for a while loop: {P} while b do S {Q}
    - The invariant P must be true before the loop starts, must be preserved by the loop body S, and must imply the postcondition Q when the loop terminates.

Evaluation

Evaluation relation (small-step/big-step)

Normal form

Confluence and termination

Evaluation on Booleans

Syntax

t	::=		*terms:*
		true	*constant true*
		false	*constant false*
		if t then t else t	*conditional*

v	::=		*values:*
		true	*true value*
		false	*false value*

Evaluation $\boxed{t \longrightarrow t'}$

$$\text{if true then } t_2 \text{ else } t_3 \longrightarrow t_2 \qquad \text{(E-IfTrue)}$$

$$\text{if false then } t_2 \text{ else } t_3 \longrightarrow t_3 \qquad \text{(E-IfFalse)}$$

$$\frac{t_1 \longrightarrow t_1'}{\text{if } t_1 \text{ then } t_2 \text{ else } t_3 \longrightarrow \text{if } t_1' \text{ then } t_2 \text{ else } t_3} \qquad \text{(E-If)}$$

t evaluates to t’ in one step

One-step Evaluation Relation

How to evaluate the term? :

if true then (if false then false else false) else true

- The *one-step evaluation relation* → is the *smallest binary relation* on terms satisfying the three rules

$$\texttt{if true then } t_2 \texttt{ else } t_3 \longrightarrow t_2 \quad \text{(E-IfTrue)}$$

$$\texttt{if false then } t_2 \texttt{ else } t_3 \longrightarrow t_3 \quad \text{(E-IfFalse)}$$

Computation rules

$$\frac{t_1 \longrightarrow t_1'}{\texttt{if } t_1 \texttt{ then } t_2 \texttt{ else } t_3 \longrightarrow \texttt{if } t_1' \texttt{ then } t_2 \texttt{ else } t_3} \quad \text{(E-If)}$$

congruence rule

- When the *pair (t, t')* is in the evaluation relation, we say that

"t → t' is *derivable*."

Derivation Tree

$$s \overset{\text{def}}{=} \texttt{if true then false else false}$$

$$t \overset{\text{def}}{=} \texttt{if s then true else true}$$

$$u \overset{\text{def}}{=} \texttt{if false then true else true}$$

"if t then false else false → if u then false else false" is witnessed by the following derivation tree:

$$\dfrac{\dfrac{\dfrac{}{\texttt{s} \longrightarrow \texttt{false}}\ \text{E-IfTrue}}{\texttt{t} \longrightarrow \texttt{u}}\ \text{E-If}}{\texttt{if t then false else false} \longrightarrow \texttt{if u then false else false}}\ \text{E-If}$$

an evaluation statement t → t' is derivable iff there is a derivation tree with t → t' as the label at its root

Induction on Derivation

a powerful proof technique based on the structure of **derivation trees** and leverages induction to prove properties that hold for all possible derivations in a formal system.

Theorem [**Determinacy of one-step evaluation**]:

$$\text{If } t \rightarrow t' \text{ and } t \rightarrow t'', \text{ then } t' = t''.$$

Proof. By induction on derivation of $t \rightarrow t'$.

If *the last rule* used in the derivation of $t \rightarrow t'$ is E-IfTrue, then t has the form

if true then t2 else t3.

It can be shown that there is only one way to reduce such t.

……

At each step of the induction, we assume the desired result for all smaller derivations, and proceed by a case analysis of the evaluation rule used at the root of the derivation.

Normal Form

One-step evaluation relation shows how an abstract machine *moves from one state to the next* while evaluating a given term.

However, from the perspective of programmers, we are interested in the **final results of evaluation,** i.e., in states from which the machine cannot take a step.

- **Definition**: A term *t* is in normal form if *no evaluation rule* applies to it.

- **Theorem**: Every *value* is in normal form.
  - At present, the converse of this Theorem is also true: every normal form is a value.
- **Theorem**: If *t* is in normal form, then *t* is a *value*.
  - Prove by contradiction (then by structural induction).

Multi-step Evaluation Relation

Relates a term to all of the terms that can be derived from it by zero or more single steps of evaluation.

- It is sometimes convenient to be able to view many steps of evaluation as one big state transition.

- **Definition**: The multi-step evaluation relation $\longrightarrow*$ is the *reflexive*, *transitive closure* of one-step evaluation.

- **Theorem** [Uniqueness of normal forms]:

  If t $\longrightarrow*$ u and t $\longrightarrow*$ u′, where u and u′ are both normal forms, then

  u = u′.

- **Theorem** [Termination of Evaluation]:

  For every term t there is some normal form t′ such that t $\longrightarrow*$ t′.

Extending Evaluation to Numbers

New syntactic forms

t	::=	...	*terms:*
		0	*constant zero*
		succ t	*successor*
		pred t	*predecessor*
		iszero t	*zero test*
v	::=	...	*values:*
		nv	*numeric value*
nv	::=		*numeric values:*
		0	*zero value*
		succ nv	*successor value*

New evaluation rules $\boxed{\mathtt{t} \longrightarrow \mathtt{t}'}$

$$\frac{\mathtt{t}_1 \longrightarrow \mathtt{t}_1'}{\mathtt{succ}\ \mathtt{t}_1 \longrightarrow \mathtt{succ}\ \mathtt{t}_1'} \quad \text{(E-Succ)}$$

$$\mathtt{pred}\ 0 \longrightarrow 0 \quad \text{(E-PredZero)}$$

$$\mathtt{pred}\ (\mathtt{succ}\ \mathtt{nv}_1) \longrightarrow \mathtt{nv}_1 \quad \text{(E-PredSucc)}$$

$$\frac{\mathtt{t}_1 \longrightarrow \mathtt{t}_1'}{\mathtt{pred}\ \mathtt{t}_1 \longrightarrow \mathtt{pred}\ \mathtt{t}_1'} \quad \text{(E-Pred)}$$

$$\mathtt{iszero}\ 0 \longrightarrow \mathtt{true} \quad \text{(E-IszeroZero)}$$

$$\mathtt{iszero}\ (\mathtt{succ}\ \mathtt{nv}_1) \longrightarrow \mathtt{false} \quad \text{(E-IszeroSucc)}$$

$$\frac{\mathtt{t}_1 \longrightarrow \mathtt{t}_1'}{\mathtt{iszero}\ \mathtt{t}_1 \longrightarrow \mathtt{iszero}\ \mathtt{t}_1'} \quad \text{(E-IsZero)}$$

Stuckness

- Definition: A closed term is stuck if it is in *normal form* but *not a value*.

- Examples:
  - succ true
  - succ false
  - if zero then true else false

Big-step Evaluation

$$v \Downarrow v \qquad \text{(B-VALUE)}$$

$$\frac{t_1 \Downarrow \texttt{true} \qquad t_2 \Downarrow v_2}{\texttt{if } t_1 \texttt{ then } t_2 \texttt{ else } t_3 \Downarrow v_2} \qquad \text{(B-IFTRUE)}$$

$$\frac{t_1 \Downarrow \texttt{false} \qquad t_3 \Downarrow v_3}{\texttt{if } t_1 \texttt{ then } t_2 \texttt{ else } t_3 \Downarrow v_3} \qquad \text{(B-IFFALSE)}$$

$$\frac{t_1 \Downarrow nv_1}{\texttt{succ } t_1 \Downarrow \texttt{succ } nv_1} \qquad \text{(B-SUCC)}$$

$$\frac{t_1 \Downarrow 0}{\texttt{pred } t_1 \Downarrow 0} \qquad \text{(B-PREDZERO)}$$

$$\frac{t_1 \Downarrow \texttt{succ } nv_1}{\texttt{pred } t_1 \Downarrow nv_1} \qquad \text{(B-PREDSUCC)}$$

$$\frac{t_1 \Downarrow 0}{\texttt{iszero } t_1 \Downarrow \texttt{true}} \qquad \text{(B-ISZEROZERO)}$$

$$\frac{t_1 \Downarrow \texttt{succ } nv_1}{\texttt{iszero } t_1 \Downarrow \texttt{false}} \qquad \text{(B-ISZEROSUCC)}$$

Summary

- How to define syntax?
  - Grammar, Inductively, Inference Rules, Generative

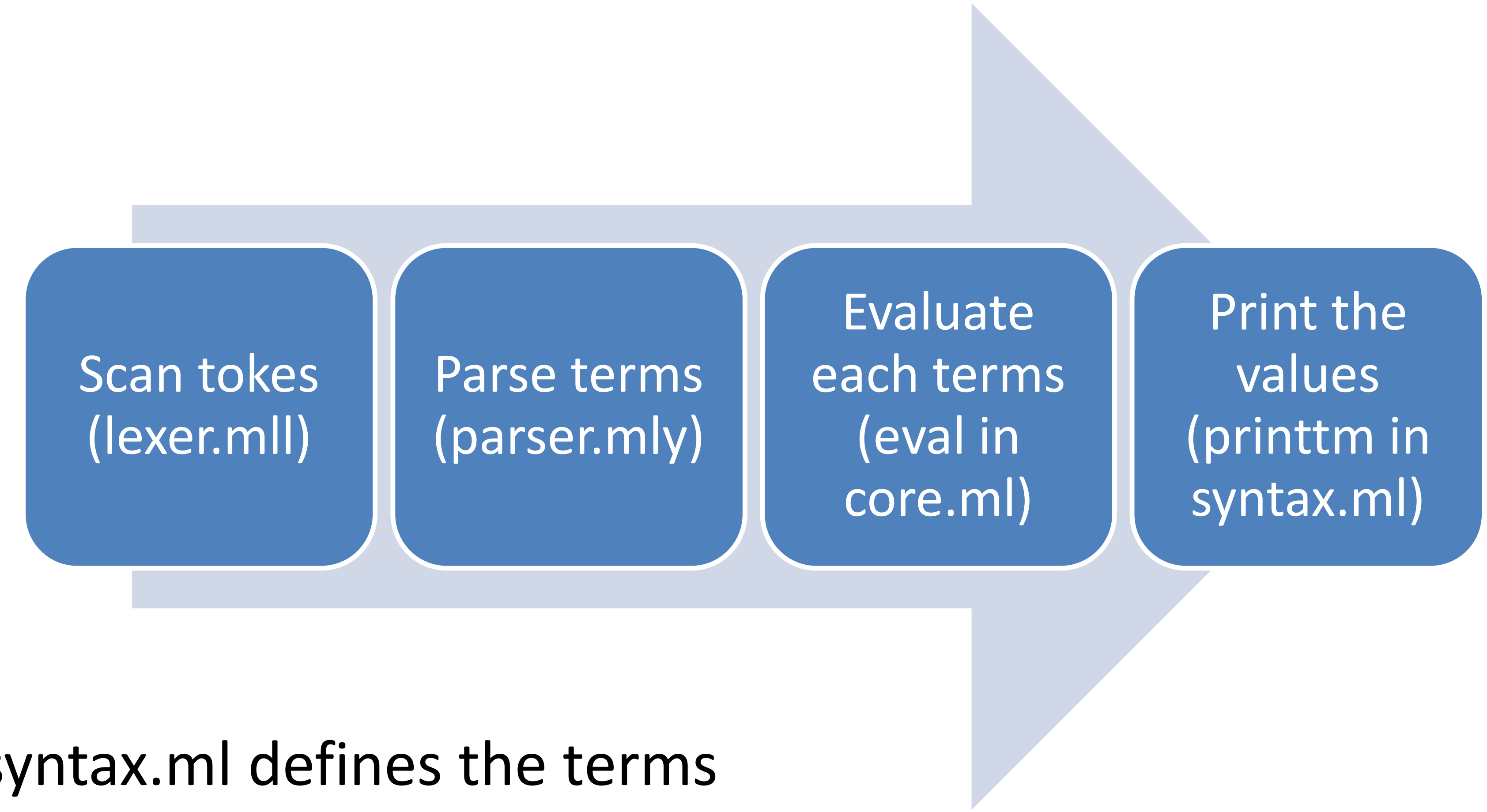
- How to define semantics?
  - Operational, Denotational, Axomatic

- How to define evaluation relation (operational semantics)?
  - Small-step/Big-step evaluation relation
  - Normal form
  - Confluence/termination

An Implementation for Arithmetic Expression

Structure of arith

main.ml drives the whole process



syntax.ml defines the terms

- Makefile
- core.ml
- core.mli
- lexer.mll
- main.ml
- parser.mly
- support.ml
- support.mli
- syntax.ml
- syntax.mli
- test.f

Makefile

```
#
# Rules for compiling and linking the typechecker/evaluator
#
# Type
#   make         to rebuild the executable file f
#   make windows to rebuild the executable file f.exe
#   make test    to rebuild the executable and run it on input file test.f
#   make clean   to remove all intermediate and temporary files
#   make depend  to rebuild the intermodule dependency graph that is used
#                by make to determine which order to schedule compilations.  You should not need to do this unless
#                you add new modules or new dependencies between existing modules.  (The graph is stored in the file
#                .depend)

# These are the object files needed to rebuild the main executable file
#
OBJS = support.cmo  syntax.cmo  core.cmo  parser.cmo  lexer.cmo  main.cmo

# Files that need to be generated from other files
DEPEND += lexer.ml parser.ml
```

Syntax.ml

```
type term =
      TmTrue of info
    | TmFalse of info
    | TmIf of info * term * term * term
    | TmZero of info
    | TmSucc of info * term
    | TmPred of info * term
    | TmIsZero of info * term
```

info: a data type recording the position of the term in the source file

eval in core.ml

```
let rec eval t =
  try let t' = eval1 t
      in eval t'
  with NoRuleApplies → t
```

eval1: perform a single step reduction

Commands

- Each line of the source file is parsed *as a command*
  - type command = | Eval of info * term
  - New commands will be added later

- Main routine for each file

```
 let process_file f  =
            alreadyImported := f :: !alreadyImported;
    let cmds = parseFile f in
    let g  c =
        open_hvbox 0;
        let results = process_command  c in
   print_flush();
   results
 in
   List.iter g  cmds
```

Homework

- Read Chapters 1 and 2.
- Read Chapter 3 and do Exercise 3.5.13 & 3.5.16.

- Please preview and install MoonBit/ OCaml and its utilities
  - MoonBit Language
    - https://www.moonbitlang.com/
  - Install OCaml and read “Basics” [optional]
    - Overview
      - https://ocaml.org/docs/
    - Installation
      - https://ocaml.org/docs/up-and-running

Homework

- Read Chapter to see how to implement a language, and download the implementation package of the TAPL (either in Ocaml or MoonBit), and digest the source codes in archives of *arith*
  - https://github.com/pku-dppl/TAPL-in-MoonBit/
  - https://www.cis.upenn.edu/~bcpierce/tapl/checkers/

- [optional] Please give your implementation for Chap. 4, and try to use *arith* to write the following equation
  - Return five if two is not zero, otherwise return nine
  - Hint: read the code in parser.mly

Thanks for listening