

Software foundations (a.k.a. "theory of programming languages") is the study of the **meaning** of programs.

The goal is finding ways to describe program behaviors that are both **precise** and **abstract**.

- ◆ Precise because we would like to prove things about how programs behave.
- ◆ Abstract because we would like the techniques that we use to apply to lots of different programs, and lots of different programming languages.

What is "software foundations"?

CIS 500
 Software Foundations
 Fall 2005
 7 September

Why study software foundations?

Course Overview

Why study software foundations?

- ◆ To be able to prove specific facts about particular programs (i.e., program verification)
- ◆ Important in some domains (safety-critical systems, hardware design, security protocols, inner loops of key algorithms, ...), but still quite difficult and expensive
- ◆ To develop intuitions for informal reasoning about programs

CIS 500, 7 September

4-b

Why study software foundations?

- ◆ To be able to prove specific facts about particular programs (i.e., program verification)
- ◆ Important in some domains (safety-critical systems, hardware design, security protocols, inner loops of key algorithms, ...), but still quite difficult and expensive
- ◆ To develop intuitions for informal reasoning about programs
- ◆ To prove general facts about all the programs in a given programming language (e.g., safety or isolation properties)
- ◆ To understand language features (and their interactions) deeply and develop principles for better language design

CIS 500, 7 September

4-d

Why study software foundations?

- ◆ To be able to prove specific facts about particular programs (i.e., program verification)
- ◆ Important in some domains (safety-critical systems, hardware design, security protocols, inner loops of key algorithms, ...), but still quite difficult and expensive

CIS 500, 7 September

4-a

Why study software foundations?

- ◆ To be able to prove specific facts about particular programs (i.e., program verification)
- ◆ Important in some domains (safety-critical systems, hardware design, security protocols, inner loops of key algorithms, ...), but still quite difficult and expensive
- ◆ To develop intuitions for informal reasoning about programs
- ◆ To prove general facts about all the programs in a given programming language (e.g., safety or isolation properties)

CIS 500, 7 September

4-c

- ◆ An introduction to programming (if this is what you want, you should be in CIT 591)
- ◆ A course on functional programming (though we'll be doing some functional programming along the way)
- ◆ A course on compilers (you should already have basic concepts such as lexical analysis, parsing, abstract syntax, and scope under your belt)
- ◆ A comparative survey of many different programming languages and styles (boring!)
- ◆ A seminar on programming language research (see CIS 670, MW 1:30-3:00, Moore 212!)

What this course is not

- ◆ To be able to prove specific facts about particular programs (i.e., program verification)
 - ◆ Important in some domains (safety-critical systems, hardware design, security protocols, inner loops of key algorithms, ...), but still quite difficult and expensive
 - ◆ To develop intuitions for informal reasoning about programs
 - ◆ To prove general facts about all the programs in a given programming language (e.g., safety or isolation properties)
 - ◆ To understand language features (and their interactions) deeply and develop principles for better language design
- PL is the “materials science” of computer science...

Why study software foundations?

“Program meaning” can be approached in many different ways.

Approaches

- ◆ 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
 - ◆ Deep intuitions about key language properties such as type safety
 - ◆ Powerful tools for language design, description, and analysis
- N.b.: most good software designers are language designers!

What you can expect to get out of the course

- “Program meaning” can be approached in many different ways.
- ◆ **Denotational semantics** and **domain theory** view programs as simple mathematical objects, abstracting away their flow of control and concentrating on their input-output behavior.
 - ◆ **Program logics** such as **Hoare logic** and **dependent type theories** focus on systems of logical rules for reasoning about programs.
 - ◆ **Operational semantics** describes program behaviors by means of **abstract machines**. This approach is somewhat lower-level than the others, but is extremely flexible.

Approaches

- “Program meaning” can be approached in many different ways.
- ◆ **Denotational semantics** and **domain theory** view programs as simple mathematical objects, abstracting away their flow of control and concentrating on their input-output behavior.
 - ◆ **Program logics** such as **Hoare logic** and **dependent type theories** focus on systems of logical rules for reasoning about programs.
 - ◆ **Operational semantics** describes program behaviors by means of **abstract machines**. This approach is somewhat lower-level than the others, but is extremely flexible.
 - ◆ **Process calculi** focus on the communication and synchronization behaviors of complex concurrent systems.

Approaches

- “Program meaning” can be approached in many different ways.
- ◆ **Denotational semantics** and **domain theory** view programs as simple mathematical objects, abstracting away their flow of control and concentrating on their input-output behavior.

Approaches

- “Program meaning” can be approached in many different ways.
- ◆ **Denotational semantics** and **domain theory** view programs as simple mathematical objects, abstracting away their flow of control and concentrating on their input-output behavior.
 - ◆ **Program logics** such as **Hoare logic** and **dependent type theories** focus on systems of logical rules for reasoning about programs.

Approaches

Overview

In this course, we will concentrate on operational techniques and type systems.

◆ Part O: Functional Programming

- ◆ A taste of OCaml
- ◆ Functional programming style
- ◆ Implementing programming languages
- ◆ Part I: Modelling programming languages
- ◆ Syntax and operational semantics
- ◆ Inductive proof techniques
- ◆ The lambda-calculus
- ◆ Syntactic sugar; fully abstract translations

8

CIS 500, 7 September

Administrative Stuff

10

CIS 500, 7 September

Approaches

“Program meaning” can be approached in many different ways.

- ◆ **Denotational semantics** and **domain theory** view programs as simple mathematical objects, abstracting away their flow of control and concentrating on their input-output behavior.
- ◆ **Program logics** such as **Hoare logic** and **dependent type theories** focus on systems of logical rules for reasoning about programs.
- ◆ **Operational semantics** describes program behaviors by means of **abstract machines**. This approach is somewhat lower-level than the others, but is extremely flexible.
- ◆ **Process calculi** focus on the communication and synchronization behaviors of complex concurrent systems.
- ◆ **Type systems** describe **approximations** of program behaviors, concentrating on the shapes of the values passed between different parts of the program.

7-e

CIS 500, 7 September

- ◆ Part II: Type systems
 - ◆ Simple types
 - ◆ Type safety
 - ◆ References
 - ◆ Subtyping
- ◆ Part III: Object-oriented features (case study)
 - ◆ A simple imperative object model
 - ◆ An analysis of core Java

9

CIS 500, 7 September

Personnel

Email for all staff: cis500@cis.upenn.edu
 Instructor: Stephanie Weirich
 Levine 510
sweirich@cis.upenn.edu
 Office hours today:
 4:30-5:30
 Throughout the semester: Mondays 4:30-5:30
 Teaching Assistants: Brian Aydemir
 Office hours: TBA
 Aaron Bohannon
 Office hours: TBA

Textbook: Types and Programming Languages,
 Benjamin C. Pierce, MIT Press, 2002
 Webpage: <http://www.seas.upenn.edu/~cis500>
 Mailing list: cis500-001-05@lists.seas.upenn.edu

Information

Administrative Assistant

Cheryl Hickey, Levine 502
 If you are unable to reach me please contact Cheryl Hickey, 215-898-3538 or cherylh@central.cis.upenn.edu. You may find your class folder in the filing cabinet outside of Room 502 Levine for all graded homeworks and extra handouts. Please see Cheryl for your graded exams.

Exams

1. **First mid-term:** Wed, October 12, in class.
2. **Second mid-term:** Wed, November 16, in class.
3. **Final:** Wed, December 14, 12:00-2:00PM.

Additional administrative information will be posted as necessary during the semester. Keep an eye on the course web page and (especially) the mailing list.

— Anon.

“You never really misunderstand something until you try to teach it..”

- ◆ Collaboration on homework is **strongly encouraged**
- ◆ Studying with other people is the best way to internalize the material
- ◆ Form study groups! 2 or 3 people is a nice size. 4 is too many for all to have equal input.
- ◆ We will help form groups for those that have not already done so
- ◆ Even if you are fairly confident about the course, you must be in a group.
- ◆ Groups should work individually, however.

Collaboration

◆ Readings from TAPL

- ◆ Should be completed **before** lecture (see course web page).
- ◆ Do all one star questions while reading (do not need to turn in).
- ◆ Write down questions to ask in class or recitation.
- ◆ Written homework
- ◆ Small part of your grade, yet a large part of your understanding.
- ◆ Submit one assignment per study group. Submit all assignments this semester with the same group. You must form your study group before the first assignment is due. Even if you find this assignment easy, you will want a group for later assignments!
- ◆ Grading is random but fair. We may not grade every problem.
- ◆ Some solutions are in the back of the book. Write your answer down **before** looking.
- ◆ Late (non-)policy: Homework will **not be accepted** after the announced deadline.

Homework

Final course grades will be computed as follows:

- ◆ Homework: 20%
- ◆ 2 midterms: 20% each
- ◆ Final: 40%

Grading

1. Grade improvements can only be obtained by sitting in on the course next year and turning in all homeworks and exams. If you are doing this to improve your grade from last year, let me know.
2. Do not ask me for extra credit projects, either during the semester or after the course ends: concentrate your efforts on this course, now.

Extra Credit

- ◆ Everyone in the class should attend one of the **recitation sections**
- ◆ Meetings of recitation sections will start **next week**.
- ◆ All sections will cover the same material. You should pick one and keep going with it.

Wed 3:30-5:00 PM	Levine 315	Bohannon
Thurs 10:30-12 PM	Levine 612	Aydemir
Thurs 10:30-3 PM	Levine 512	Bohannon
Fri 9:30-11 AM	Levine 512	Aydemir

Recitations

- ◆ Your study group
- ◆ Recitation sections
- ◆ The course mailing list cis500-001-05c@lists.seas.upenn.edu
- ◆ Office hours

Where to go for help

- ◆ PhD students in CIS must pass a Written Preliminary Exam (WPE-I). Software Foundations is one of the WPE-I areas
- ◆ The final for this course is also the software foundations WPE-I exam
- ◆ Near the end of the semester, you will be given an opportunity to declare your intention to take the final exam for WPE credit

The WPE-I

- ◆ The first homework assignment is due Monday, September 19, by noon. (You have 1.5 weeks for this assignment.)
- ◆ The assignment is posted on the course web page.
- ◆ All assignments must be typeset and submitted electronically. The use of LaTeX is strongly encouraged.

First Homework Assignment

- ◆ The department offers a **Research Seminar** most weeks during the Fall semester
- ◆ Thursday afternoons, 3 – 4, in Levine Auditorium
- ◆ Speakers and topics are announced on the CIS newsgroups and mailing lists
- ◆ First-year CIS PhD students are required to attend. Others are welcome.

Announcement

- ◆ You do not need to be enrolled in the course to take the exam for WPE credit
- ◆ If you are enrolled in the course and also take the exam for WPE credit, you will receive two grades: a letter grade for the course final and a Pass/Fail for the WPE
- ◆ You may take the exam for WPE credit even if you are not currently enrolled in the PhD program.

The WPE-I (continued)

What is a programming language?

- ◆ Reading knowledge of core OCaml
- ◆ Chapters 1-11 and 13-19 of TAPL

The WPE-I syllabus

- The set B of **boolean terms** is the smallest set such that
1. $\{\text{true}, \text{false}\} \subseteq B$;
 2. if $t_1 \in B$, then $\{\text{not } t_1\} \subseteq B$;
 3. if $t_1 \in B$, $t_2 \in B$, and $t_3 \in B$, then if t_1 then t_2 else $t_3 \in B$.

Another form of the definition

Syntax

Q1: Does this grammar define a set of **character strings**, a set of **token lists**, or a set of **abstract syntax trees**?

Abstract vs. concrete syntax

We can define the **terms** of a programming language in a number of different ways.
 Here is a BNF grammar for a very simple language of boolean expressions:

$t ::=$
 true
 false
 $\text{not } t$
 $\text{if } t \text{ then } t \text{ else } t$
conditional
 constant true
 constant false
negation
conditional
terms

Terminology:

◆ t here is a **metavariable**

Defining a programming language

Semantics

Abstract Syntax, not semantics

We've only defined the abstract syntax of our language. That means our language is just a set of terms.
We haven't assigned any meanings to those terms yet. So there is no reason why we should equate **true** or **not false**, they're just uninterpreted terms. Soon we will start talking about how we can decide what these terms mean.

Q: So, are
not false
not (false)
(((not (((false))))))
"the same term"?
What about
true
not false
?

Abstract vs. concrete syntax

Q1: Does this grammar define a set of **character strings**, a set of **token lists**, or a set of **abstract syntax trees**?
A: In a sense, all three. But we are interested in abstract syntax trees. For this reason, grammars like the one on the previous slide are sometimes called **abstract grammars**. An abstract grammar **defines** a set of abstract syntax trees and **suggests** a mapping from character strings to trees. We then **write** terms as linear character strings rather than trees simply for convenience. If there is any potential confusion about what tree is intended, we use parentheses to disambiguate.

Defining what a language “means”

As well as defining the syntax of a programming language, we also need to define its semantics or the “meaning” of expressions written in that language.

Styles of semantics

1. **Operational Semantics** specifies the behavior of programs, much like an interpreter.
 2. **Denotational Semantics** translates programs to a domain that we already know the meaning of: mathematics. The meaning of a term is a mathematical object like a function.
 3. **Axiomatic Semantics** describes the meaning of a program through laws that describe its behavior.
- In this course we will concentrate on **operational semantics**.

Operational Semantics

Eval is a relation between terms in B . It is the smallest set such that:

Operational Semantics

- ◆ Describes the evaluation of programs on an abstract machine.
- ◆ Defined by a relation between each program and its result of evaluation.
- ◆ Several ways to define operational semantics—we’ll look at a few in this course.
- ◆ We want the programs **not false** and **true** to mean the same thing.

Operational Semantics

Eval is a relation between terms in B . It is the smallest set such that:

- ◆ $(\text{true}, \text{true}) \in \text{Eval}$

- Operational Semantics**
-
- Eval* is a relation between terms in \mathcal{B} . It is the smallest set such that:
- ◆ $(\text{true}, \text{true}) \in \text{Eval}$
 - ◆ $(\text{false}, \text{false}) \in \text{Eval}$
 - ◆ $(\text{not } t, \text{true}) \in \text{Eval}$ when $(t, \text{false}) \in \text{Eval}$
 - ◆ $(\text{not } t, \text{false}) \in \text{Eval}$ when $(t, \text{true}) \in \text{Eval}$

- Operational Semantics**
-
- Eval* is a relation between terms in \mathcal{B} . It is the smallest set such that:
- ◆ $(\text{true}, \text{true}) \in \text{Eval}$
 - ◆ $(\text{false}, \text{false}) \in \text{Eval}$
 - ◆ $(\text{not } t, \text{true}) \in \text{Eval}$ when $(t, \text{false}) \in \text{Eval}$
 - ◆ $(\text{not } t, \text{false}) \in \text{Eval}$ when $(t, \text{true}) \in \text{Eval}$
 - ◆ If t_1 then t_2 else t_3 , $t) \in \text{Eval}$ when either:
 - ◆ $(t_1, \text{true}) \in \text{Eval}$ and $(t_2, t) \in \text{Eval}$
 - ◆ $(t_1, \text{false}) \in \text{Eval}$ and $(t_3, t) \in \text{Eval}$

- Operational Semantics**
-
- Eval* is a relation between terms in \mathcal{B} . It is the smallest set such that:
- ◆ $(\text{true}, \text{true}) \in \text{Eval}$
 - ◆ $(\text{false}, \text{false}) \in \text{Eval}$

- Operational Semantics**
-
- Eval* is a relation between terms in \mathcal{B} . It is the smallest set such that:
- ◆ $(\text{true}, \text{true}) \in \text{Eval}$
 - ◆ $(\text{false}, \text{false}) \in \text{Eval}$
 - ◆ $(\text{not } t, \text{true}) \in \text{Eval}$ when $(t, \text{false}) \in \text{Eval}$

Proving properties about programming languages

We want to show that a property is true for all $t \in \mathcal{B}$?
Can't do it by case analysis: \mathcal{B} is an infinite set.

Example: Natural number induction

Principle of **ordinary induction** on natural numbers
Suppose that P is a predicate on the natural numbers. Then:
If $P(0)$
and, for all $i \in \mathcal{N}$, $P(i)$ implies $P(i+1)$,
then $P(n)$ holds for all $n \in \mathcal{N}$.

Operational Semantics

$Eval$ is a relation between terms in \mathcal{B} . It is the smallest set such that:

- ◆ $(true, true) \in Eval$
 - ◆ $(false, false) \in Eval$
 - ◆ $(not\ t, true) \in Eval$ when $(t, false) \in Eval$
 - ◆ $(not\ t, false) \in Eval$ when $(t, true) \in Eval$
 - ◆ $(if\ t_1\ then\ t_2\ else\ t_3, t) \in Eval$ when either:
 - ◆ $(t_1, true) \in Eval$ and $(t_2, t) \in Eval$
 - ◆ $(t_1, false) \in Eval$ and $(t_3, t) \in Eval$
- If $(t_1, t_2) \in Eval$ we say that t_2 is the **meaning** of t_1 .

Properties of boolean language

Now that we have defined the **syntax** and **semantics** of the boolean language, what properties are true?

- ◆ $(true, false) \notin Eval$.
- ◆ **not false** and **true** have the same meaning.
- ◆ All boolean terms have meanings ($Eval$ is total).
- ◆ There is only one meaning for each term ($Eval$ is deterministic).

How do we show that these properties are true?

Theorem: $2^0 + 2^1 + \dots + 2^n = 2^{n+1} - 1$, for every n .
 Proof:
 ♦ Let $P(i)$ be “ $2^0 + 2^1 + \dots + 2^i = 2^{i+1} - 1$.”
 ♦ Show $P(0)$:
 $2^0 = 1 = 2^1 - 1$

Example

Theorem: $2^0 + 2^1 + \dots + 2^n = 2^{n+1} - 1$, for every n .
 Proof:
 ♦ Let $P(i)$ be “ $2^0 + 2^1 + \dots + 2^i = 2^{i+1} - 1$.”
 ♦ Show $P(0)$:
 $2^0 = 1 = 2^1 - 1$
 ♦ Show that $P(i)$ implies $P(i+1)$:
 $2^0 + 2^1 + \dots + 2^{i+1} = (2^0 + 2^1 + \dots + 2^i) + 2^{i+1}$
 $= (2^{i+1} - 1) + 2^{i+1}$
 $= 2 \cdot (2^{i+1}) - 1$
 $= 2^{i+2} - 1$
 by IH

Example

The reason that we have an induction principle for natural numbers, is because they are defined in a certain way:
 The set \mathcal{N} is the smallest set such that
 1. $0 \in \mathcal{N}$.
 2. If $n \in \mathcal{N}$ then $n+1 \in \mathcal{N}$.
 For shorthand, we sometimes abbreviate $0 + 1$ as 1 , and $0 + 1 + 1$ as 2 , and $0 + 1 + 1 + 1$ as 3 , etc.

Natural numbers

Theorem: $2^0 + 2^1 + \dots + 2^n = 2^{n+1} - 1$, for every n .
 Proof:
 ♦ Let $P(i)$ be “ $2^0 + 2^1 + \dots + 2^i = 2^{i+1} - 1$.”

Example

- This is the same way we defined what boolean terms were.
- The set \mathcal{B} of **boolean terms** is the smallest set such that
1. $\{\text{true}, \text{false}\} \subseteq \mathcal{B}$;
 2. if $t_1 \in \mathcal{B}$, then $\{\text{not } t_1\} \subseteq \mathcal{B}$;
 3. if $t_1 \in \mathcal{B}$, $t_2 \in \mathcal{B}$, and $t_3 \in \mathcal{B}$, then if t_1 then t_2 else $t_3 \in \mathcal{B}$.

Inductive definitions

- We can also use **induction** for boolean terms. The way we have defined terms gives us an induction principle:
- For all $t \in \mathcal{B}$, $P(t)$ is true if and only if
- ◆ $P(\text{true})$ and $P(\text{false})$ hold
 - ◆ for all $t_1 \in \mathcal{B}$, if $P(t_1)$ holds, then $P(\text{not } t_1)$ hold.
 - ◆ for all $t_1, t_2, t_3 \in \mathcal{B}$, if $P(t_1)$, $P(t_2)$ and $P(t_3)$ holds, then $P(\text{if } t_1 \text{ then } t_2 \text{ else } t_3)$ holds.

Structural Induction

- ◆ The result ($P(n)$) for all n) follows by the principle of induction.
- $$\begin{aligned}
 2^0 + 2^1 + \dots + 2^{i+1} &= (2^0 + 2^1 + \dots + 2^i) + 2^{i+1} \\
 &= (2^{i+1} - 1) + 2^{i+1} \\
 &= 2 \cdot (2^{i+1}) - 1 \\
 &= 2^{i+2} - 1
 \end{aligned}$$
- by IH
- ◆ Show that $P(i)$ implies $P(i + 1)$:
- $$2^0 = 1 = 2^1 - 1$$
- ◆ Show $P(0)$:
- ◆ Let $P(i)$ be " $2^0 + 2^1 + \dots + 2^i = 2^{i+1} - 1$."

Example

Theorem: $2^0 + 2^1 + \dots + 2^n = 2^{n+1} - 1$, for every n .
 Proof: By induction on n .

- ◆ Base case ($n = 0$): $2^0 = 1 = 2^1 - 1$
- ◆ Inductive case ($n = i + 1$):

$$\begin{aligned}
 2^0 + 2^1 + \dots + 2^{i+1} &= (2^0 + 2^1 + \dots + 2^i) + 2^{i+1} \\
 &= (2^{i+1} - 1) + 2^{i+1} \\
 &= 2 \cdot (2^{i+1}) - 1 \\
 &= 2^{i+2} - 1
 \end{aligned}$$

IH

Shorthand form

Proofs by induction

We'll prove that evaluation is deterministic. In other words: For all t there exists **at most** one t' such that $(t, t') \in Eval$.

This gives us the property:

$P(t) =$ exists at most one t' such that $(t, t') \in Eval$.

So we want to show:

◆ $P(\text{true})$ (i.e. exists at most one t' such that $(\text{true}, t') \in Eval$)

◆ $P(\text{false})$

◆ $P(\text{not } t_1)$ given that $P(t_1)$ holds.

◆ $P(\text{if } t_1 \text{ then } t_2 \text{ else } t_3)$ given that $P(t_1)$, $P(t_2)$ and $P(t_3)$ all hold.

Definition of *Eval*

Definition: *Eval* is the smallest set such that:

◆ $(\text{true}, \text{true}) \in Eval$

◆ $(\text{false}, \text{false}) \in Eval$

◆ $(\text{not } t, \text{true}) \in Eval$ when $(t, \text{false}) \in Eval$

◆ $(\text{not } t, \text{false}) \in Eval$ when $(t, \text{true}) \in Eval$

◆ $(\text{if } t_1 \text{ then } t_2 \text{ else } t_3, t) \in Eval$ when either:

◆ $(t_1, \text{true}) \in Eval$ and $(t_2, t) \in Eval$

◆ $(t_1, \text{false}) \in Eval$ and $(t_3, t) \in Eval$