What is "Software Foundations"?

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 to prove things about how programs behave in different programs and lots of different programming languages.

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.

P.S. The "material science" of computer science...
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, verification)

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

PLece the material in science of computer science...
Why study software foundations?

To be able to prove specific facts about particular programs (e.g., program verification)

Important in some domains (e.g., cyber-physical systems, hardware design, safety-critical systems, ...), but still quite difficult and expensive

To prove general facts about all the programs in a given programming paradigm (e.g., functional programming)

To develop intuitions for informal reasoning about programs

To understand intuitions formed by their interactions (e.g., dependency graphs)

To develop intuitions for informal reasoning about programs

Secure protocols, integer loops of key algorithms, ... but still quite difficult and expensive

To solve general problems and all the programs in a given programming paradigm

To prove general facts about all the programs in a given programming paradigm

To prove theorems about particular programs (e.g., program verification)

What can you expect to get out of the course

A more sophisticated perspective on programs' programming language

How to prove and prove theorems about them

A more sophisticated perspective on programs' programming language

Deep intuitions about key language properties such as type safety

Detailed study of a range of basic language features (e.g., syntax)

How to view programs and whole languages as formal, mathematical objects

N.B.: Most good software designers are language designers

What this course is not

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)

A course on compilers (you should already have basic concepts such as lexical analysis, parsing, abstract syntax, and scope under your belt)

A course on concurrency (you should already have basic concepts such as lock guards, mutual exclusion, and scope under your belt)

V. Operational semantics describes program behaviors by means of abstract machines. This approach is somewhat less flexible.

V. Type systems describe approximations of program behaviors, concentrating on the shapes of the values passed between different parts of the program.

Process calculi focus on the communication and synchronization behaviors of complex concurrent systems.

Program logics such as Hoare logic and dependent type theories focus on systems of logical rules for reasoning about programs.

V. Denotational semantics and domain theory view programs as simple mathematical objects, abstracting away their control and concentrating on their input-output behavior.

V. Computational survey of many different programming languages and styles

V. Course on concurrency (you should already have basic concepts such as lock guards, mutual exclusion, and scope under your belt)

V. Course on compactness (though we'll be doing some)

V. Course in functional programming

V. Course in programming languages

An introduction to programming (it this is what you want, you should be in CIT 591)

Approaches

- Programs meaning can be approached in many different ways.

What you can expect to get out of the course

A more sophisticated perspective on programs' programming language

How to view programs and whole languages as formal, mathematical objects

Deep intuitions about key language properties such as type safety

Detailed study of a range of basic language features (e.g., syntax)

How to prove and prove theorems about them

A more sophisticated perspective on programs' programming language

Why this course is not

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)

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
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 that simulate the run-time behavior of programs.

Process calculi focus on the communication and synchronization behaviors of complex concurrent systems.

Typesystems describe approximations of program behaviors, concentrating on the shapes of the values passed between different parts of the program.

Extremely flexible.

Intermediate means. This approach is somewhat lower-level than the others, but is extremely flexible.

Operations semantics describes program behaviors by means of abstract machines that simulate the run-time behavior of programs.

Program logics such as Hoare logic and dependent type theories focus on the input-output behavior of programs, abstracting away their flow of control and concentrating on their shapes.

Denotational semantics and domain theory view programs as simple.

Program meaning can be approached in many different ways.
Approaches

Program meaning can be approached in many different ways.

- Denotational and domain theory view programs as simple mathematical objects, abstracting away their 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.
- Operationals describe programs behaviors by means of abstract machines. This approach is somewhat low-level than the others but is extremely flexible.
- Process calculi focus on the communication and synchronization behaviors of complex concurrent systems.
- Types describe approximations of program behaviors, concentrating on the shapes of the values passed between different parts of the program.

Overview

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

1. First midterm: Wed, October 12, in class.
3. Final Exam: Wed, December 14, 12:00-2:00PM

Information

Textbook: Types and Programming Languages, Benjamin C. Pierce, MIT Press, 2002

Webpage: http://www.seas.upenn.edu/cis500

Mailing list: cis500-001-05c@lists.seas.upenn.edu

Office hours: Thursdays: 4:30-5:30
Office hours location: Faraday B02

Office hours: Thursdays: 4:30-5:30
Office hours location: Faraday B02

Instructor: Stephanie Weirich
Email: sweirich@cis.upenn.edu

Personal Assistant: Cheryl Hickey, Levine 502
Email: cherylh@central.cis.upenn.edu

Administrative Assistant: Cheryl Hickey, Levine 502
Office hours: Thursdays: 4:30-5:30
Office hours location: Faraday B02
Grading
Final course grades will be computed as follows:

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

Extra Credit
1. Grade improvements can only be obtained by studying 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.

Collaboration
Collaboration on homework is strongly encouraged. However, even if you are fairly confident about the course, you must be in a group. Form study groups of 2 or 3 people. 4 is too many for all to have equal input. Written homework should be completed before lecture (see course webpage). We will help form groups for those that have not already done so.

Homework
Readings from TAPL should be completed before lecture (see course webpage). Some solutions are in the back of the book. Write your answers down before looking.

Deadline:
Homework will not be accepted after the announced deadline.

Writing homework before looking.

Some solutions are in the back of the book. Write your answer down before looking.

Grading is random but fair. We may not grade every problem.

Sitting in 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.

You never really misunderstand something until you try to teach it... — Anon.

Groups should work individually. However, even if you are fairly confident about the course, you must be in a group. Form study groups of 2 or 3 people. 4 is too many for all to have equal input. Writing homework should be completed before lecture (see course webpage).
Where to go for help

Your study group
Recitation sections
The course mailing list

cis500-001-05c@lists.seas.upenn.edu

Office hours
The course mailing list
Recitation sections
Your study group

The WPE-I

PhD students in CIS must pass a Written Preliminary Exam (WPE-I).

First Homework Assignment

The first homework assignment is due Monday, September 19, by noon.

You have 1.5 weeks for this assignment.

All assignments must be typeset and submitted electronically. The use of LaTeX is strongly encouraged.

The final for this course is also the software foundations WPE-I exam.
You do not need to be enrolled in the course to take the exam for WPE credit.

**The WPE-I Syllabus**

- Chapters 1-11 and 12-19 of TAPL
- Reading knowledge of core OCaml

**Announcement**

- The department offers a Research Seminar most weeks during the Fall

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

**CIS500, 7 September 24**

**CIS500, 7 September 25**

**CIS500, 7 September 26**

---

**What is a Programming Language?**
Deﬁning a programming language

We can deﬁne the terms of a programming language in a number of different ways.

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?

Another form of the deﬁnition

The set \( B \) of boolean terms is the smallest set such that

1. \( \text{true}, \text{false} \in B \);
2. if \( t_1 \in B \) and \( t_2 \in B \), then \( \text{not} \ t_1, \text{if} \ t_1 \text{then} t_2 \in B \);
3. if \( t_1 \in B \) and \( t_2 \in B \), then \( \text{if} \ t_1 \text{then} t_2 \in B \).

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.
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 to distinguish is just a set of terms.

We've only defined the abstract syntax of our language. That means our

Abstract Syntax, not semantics

Soon we will start talking about how we can decide what those terms mean.

We haven't assigned any meanings to those terms yet. So there is no reason

Abstract Syntax, not semantics

Soon we will start talking about how we can decide what those terms mean.

We haven't assigned any meanings to those terms yet. So there is no reason

Abstract Syntax, not semantics

Soon we will start talking about how we can decide what those terms mean.

We haven't assigned any meanings to those terms yet. So there is no reason

Abstract Syntax, not semantics

Soon we will start talking about how we can decide what those terms mean.

We haven't assigned any meanings to those terms yet. So there is no reason

Abstract Syntax, not semantics

Soon we will start talking about how we can decide what those terms mean.

We haven't assigned any meanings to those terms yet. So there is no reason
Defining what a language means

As well as defining the syntax of a programming language, we also need to define what a language means.

In this course we will concentrate on operational semantics.

They describe the behavior of programs. Operational semantics describes the meaning of a program through laws.

Denotational Semantics translates programs to a domain that we already know the meaning of, e.g. mathematics. The meaning of a term is a mathematical object like a function.

Axiomatic Semantics specifies the behavior of programs, much like an interpreter.

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:

\[
\begin{align*}
(true, true) & \in \text{Eval} \\
(false, false) & \in \text{Eval} \\
\text{not} \ (true) & \in \text{Eval} \\
\text{not} \ (false) & \in \text{Eval} \\
\text{if} \ t_1 \ \text{then} \ t_2 \ \text{else} \ t_3 & \in \text{Eval} \\
\text{if} \ (true) \ t_1 \ (false) \ t_2 & \in \text{Eval} \\
\text{if} \ (false) \ t_1 \ (true) \ t_2 & \in \text{Eval}
\end{align*}
\]

If $(t_1, t_2) \in \text{Eval}$, we say that $t_2$ is the meaning of $t_1$. 
Operational Semantics

The meaning of an expression is the set of values that the expression can evaluate to. Mathematically, we can express this as:

$$\text{Eval} \ x \equiv \{ v \mid \exists \text{valuation } \Phi \text{ s.t. } v = \text{val}(x, \Phi) \}$$

where $$\text{val}(x, \Phi)$$ is the value of $$x$$ under the valuation $$\Phi$$.

Operational Semantics

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

- $$B$$ contains all the pairs $$(x, x)$$ for $$x$$ in $$\mathcal{D}$$.
- If $$B$$ contains $$(x, y)$$ and $$(y, z)$$, then $$(x, z)$$ is in $$B$$.

Operational Semantics

$$\text{Eval} \ x \equiv \text{Least } B \text{ containing all } (x, x) \in \mathcal{D} \times \mathcal{D}$$

where $$\text{Eval}(x, \Phi)$$ is the value of $$x$$ under the valuation $$\Phi$$.
Operational Semantics

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

- $(\text{true}; \text{true}) \in \text{Eval}$
- $(\text{false}; \text{false}) \in \text{Eval}$
- $(\text{not} \text{true}; \text{true}) \in \text{Eval}$
- $(\text{not} \text{false}; \text{false}) \in \text{Eval}$
- $(\text{if} t \text{then} t_1 \text{else} t_3; t) \in \text{Eval}$ where:
  - $(t_1; \text{true}) \in \text{Eval}$
  - $(t_1; \text{false}) \in \text{Eval}$
  - $(t_3; \text{true}) \in \text{Eval}$
  - $(t_3; \text{false}) \in \text{Eval}$

Properties of boolean language

Now that we have defined the syntax and semantics of the boolean language, we want to show that a property is true for all $t \in B$.

We can't do this by case analysis: $B$ is an infinite set.

Example: Natural number induction

Principle of ordinary induction on natural numbers

Suppose that $d$ is a relation on the natural numbers. Then:

- $(n \in \mathbb{N}) \text{ holds for all } n \in \mathbb{N}$.
- and, for all $n \in \mathbb{N}$, $d(1) \implies d(n+1)$.

Then $d$ is a relation on the natural numbers.

Properties of boolean language

Operational Semantics

Proving properties about programming languages
Theorem: $\sum_{i=0}^{n} z_i = 1 - z_{n+1} = 1 - 1 = 0$.

Proof: by IH.

**Example.**

Show that $\sum_{i=0}^{1} z_i = 1 - 1 = 0$.

Proof: by IH.

**Example.**

Show that $\sum_{i=0}^{2} z_i = 1 - 1 = 0$.

Proof: by IH.

**Example.**

Show that $\sum_{i=0}^{3} z_i = 1 - 1 = 0$.

Proof: by IH.

The reason that we have an induction principle for natural numbers is because...
We can also use induction for boolean terms. The way we have defined terms gives us an induction principle:

**Inductive Definitions**

**Theorem:** The result of applying all (u) follows by the principle of induction.

\[
\begin{align*}
1 - z + 1 \cdot z &= \\
1 - (1 + z) \cdot z &= \\
\text{IH} &
\end{align*}
\]

**Example:**

\[
\begin{align*}
1 - 1 - 1 - z &= \\
\text{IH} &= \\
1 - 0 - z &= \\
\text{Base case: } (0) \cdot u &= \\
\text{Proof: } &
\end{align*}
\]

\[
\begin{align*}
\text{Theorem: } z + \text{IH} + \varnothing &= \\
\text{IH} &= \\
\text{Base case: } (0) &= \\
\text{Proof: } &
\end{align*}
\]
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') ∈ Eval.

This gives us the property:

P(t) = exists at most one t' such that (t, t') ∈ Eval.

P(true) (i.e. exists at most one t' such that (true, t') ∈ Eval)

P(false)

P(not t) given that P(t) holds.

P(if t, then t' else t'') given that P(t), P(t'), P(t'') all hold.

Definition of Eval

Definition: Eval is the smallest set such that:

- (true, true) ∈ Eval
- (false, false) ∈ Eval
- (not t, true) ∈ Eval when (t, false) ∈ Eval
- (not t, false) ∈ Eval when (t, true) ∈ Eval
- (if t, then t', else t'') ∈ Eval when (t, t') ∈ Eval and (t, t'') ∈ Eval

- (t', true) ∈ Eval when (t, true) ∈ Eval
- (t', false) ∈ Eval when (t, false) ∈ Eval

- (not t', true) ∈ Eval when (t, false) ∈ Eval
- (not t', false) ∈ Eval when (t, true) ∈ Eval

- (if t', then t', else t'') ∈ Eval when (t, t') ∈ Eval and (t, t'') ∈ Eval.