

Computer Science 203
Programming Languages
Fall 2005

Cormac Flanagan

University of California, Santa Cruz

1

Administrivia

- Who am I
 - Cormac Flanagan
 - cormac@cs.ucsc.edu
 - <http://www.so.e.ucsc.edu/~cormac/>
- Office hours
 - after class
 - Thursday, 11-noon, Engineering 2 room 367
- Teaching assistant
 - Jessica Gronski (jgronski@soe.ucsc.edu)
 - Mon & Wed, 10-11am, E2-392
- Web page
 - <http://www.so.e.ucsc.edu/classes/cmcs203/Fall05/>
 - lecture slides, notes, homeworks, project ideas

2

Prerequisites

- Programming-language background
 - some programming experience
 - exposure to several languages (e.g., C, Java, ML, Lisp, Prolog)
 - Computer Science 112 or equivalent
 - comparative programming languages (declarative, functional, OO)
- Some mathematical sophistication
 - ability to follow and do proofs
 - eg, proofs by induction
 - some acquaintance with mathematical logic
 - ease with formal notation and manipulation

3

Prerequisites (cont.)

- Please ask me if:
 - you are not sure that you meet the prerequisites, or
 - you are an undergraduate, or
 - a graduate student from outside CS or CE, or
 - if you did not enroll in the course but want to attend

4

Course Goals

- Learn about techniques for understanding and describing programming languages
 - operational semantics
 - axiomatic semantics
 - type systems
- Learn about some languages and language features
 - lambda calculus, objects, concurrency
- Learn a little about research topics and applications
 - security of mobile code, information flow

5

Reading

- Required reading
 - "The Formal Semantics of Programming Languages" by Glynn Winskel
 - Several papers, indicated during the course
 - Language documentation
- Some optional, additional books
 - "Foundations for Programming Languages" by John Mitchell
 - "Types and Programming Languages" by Benjamin Pierce
- All books on reserve in the library

6

Course work

- Reading
- Class participation
- Homework
 - concentrated in first half of course
 - announced and explained in class
 - hand in on paper at start of class on due date
- A small final project
 - concentrated in second half of course
 - 20-40+ hours of work per person
 - short report and short class presentation

7

Grades

- Grades are determined as follows
 - homework: 60 - 70 %
 - project (including report and presentation): 20 - 30 %
 - class participation: the rest
- Regular class attendance is required.

8

Who are you?

9

The Landscape

- Programming languages is one of the oldest fields in computer science.
- It remains an active, vibrant field.
- Some trends:
 - The web renews interest in language design (see Java).
 - Type safety gains acceptance as playing a role in security.
 - Program analysis becomes a significant component of what is called software engineering.

10

Language Design in Practice

- Languages are adopted to fill a void.
 - To enable a previously difficult/impossible application.
 - To run on a new platform.
- Language quality is sometimes secondary.
- Programmer training is a huge cost.
 - Languages with many users are replaced rarely.
 - Popular languages become ossified.
 - Conversely, new applications and new platforms present opportunities for innovation.

11

Why So Many Languages?

- Many languages were created for specific applications
- Application domains have distinctive (and conflicting) needs.
- Examples:
 - Artificial intelligence: symbolic computation (Lisp, Prolog)
 - Scientific computing: high performance (Fortran)
 - Business: report generation (COBOL)
 - Systems programming: low-level access (C)
 - Customization: scripting (Perl, Javascript)
 - Distributed systems: mobile computation (Java, C#)
 - And more (LaTeX, awk, Hancock, ...)

12

Language Paradigms

- Imperative, procedural
 - Fortran, Algol, Cobol, C, Pascal
- Functional (mostly assignment-free)
 - Lisp, Scheme, ML, Haskell
- Object oriented
 - Simula, Smalltalk, Eiffel, Self, C++, Java
- Logic
 - Prolog, λProlog
- Concurrent
 - Fortran90

13

What Makes a Good Language?

- There are no universally accepted metrics for design.
- "A good language is one people use" ?
 - Is COBOL the best language?

14

What is Science?

- "When you can measure what you are speaking about, and express it in numbers, you know something about it; but when you cannot measure it, when you cannot express it in numbers, your knowledge of it is of a meager and unsatisfactory kind; it may be the beginning of knowledge, but you have scarcely, in your thoughts, advanced it to the stage of science."
 - Lord Kelvin (1824-1907)

15

Good Language Features

- Simplicity (of syntax and semantics)
- Readability and writability
- Familiarity
- Safety
- Machine independence
- Support for programming in the large
- Efficiency (of execution and compilation)

16

Good Language Features (cont.)

- These goals almost always conflict.
 - Safety checks cost something in either compilation or execution time.
 - Safety and machine independence may exclude efficient low-level operations.
 - Type systems restrict programming style in exchange for strong guarantees.
 - ...
- Compromises are hard.
- Simplicity helps with other goals, but is rare.
- Good language design is hard!

17

Languages vs. Features

- Popular languages are isolated points in a large space.
- The language designer should (according to Hoare)
 - be familiar with many features,
 - have good judgment in choosing and reconciling features,
 - know the application domain of the new language,
 - know how hairy the language should get,
 - have resources for implementations, documentation, and environment.
- The language designer should not include new, untried ideas
 - "Goal is consolidation, not innovation."
 - Java a good example of such consolidation

18

Studying Features

- A language concept or construct can be studied in the context of a large (real) language
 - good for understanding pragmatics in practice
- But is often easier to study a language feature by considering a *small language* that embodies it
 - easier experimentation
 - deeper, more precise reasoning and understanding

19

Considering Features in Small Languages

- Easier to:
 - experiment with the use of the language
 - define the language rigorously
 - syntax
 - compile-time checks
 - run-time behavior
 - study the properties of the language
 - is it deterministic?
 - what does typechecking guarantee?
 - applications: no information leakage? no viruses?

20

Considering Features in Small Languages

- Easier to:
 - experiment with the use of the language
 - define the language rigorously
 - syntax - grammars
 - compile-time checks
 - run-time behavior
 - study the properties of the language
 - is it deterministic?
 - what does typechecking guarantee?
 - applications: no information leakage? no viruses?

21

Considering Features in Small Languages

- Easier to:
 - experiment with the use of the language
 - define the language rigorously
 - syntax - grammars
 - compile-time checks - **type systems**
 - run-time behavior - **semantics**
 - study the properties of the language (**and prove them**)
 - is it deterministic?
 - what does typechecking guarantee?
 - no seg. faults?
 - no information leakage?
 - no viruses?

22

Homework 1

- Due at start of class, Sept 27 (strictly)
- Download the Objective Caml system from <http://caml.inria.fr/ocaml/>.
- Write a function f that takes an integer argument n and outputs the string "Hello world" followed by $|n|$ "!"s, where $|n|$ is the absolute value of n .
 - For example, $f(0)$ should output "Hello world", and $f(2)$ and $f(-2)$ should both output "Hello world!!".
 - Include some test runs in your homework

23

Break Time

24

Considering Features in Small Languages

- Easier to:
 - experiment with the use of the language
 - define the language *rigorously*
 - syntax - grammars
 - compile-time checks - **type systems**
 - run-time behavior - **semantics**
 - study the properties of the language (and prove them)
 - is it deterministic?
 - what does typechecking guarantee?
 - applications: no information leakage? no viruses?

25

Caveats on Semantics

- Semantics is very precise but heavyweight
 - not always necessary or cost-effective
- Most programmers are content with less precise knowledge and to learn by trial and error.

26

Who Needs Semantics

- Those who write programs that manipulate other programs:
 - compilers and interpreters
 - program transformation and instrumentation tools
 - program analyzers
 - software engineering tools
- Those who write critical software
- Those who want to describe unambiguously a language feature or a program transformation:
 - Semantics will help us in having precise discussions.
 - Semantics is the basis for most formal arguments.
 - Semantics is a standard tool.

27

A Story: The Clash of Two Features

- A real story about bad programming language design.
- The cast includes famous scientists.
- ML ('82) is a functional language with *polymorphism* and *monomorphic references*.
- Standard ML ('85) innovates by adding *polymorphic references*.
- It took 10 years to fix the "innovation".

28

Polymorphic References

Consider the following program fragment:

Code	Type inference
<code>fun id x = x</code>	<code>id : $\alpha \rightarrow \alpha$</code>
<code>val p = ref id</code>	<code>p : ($\alpha \rightarrow \alpha$) ref</code>
<code>fun inc x = x + 1</code>	<code>inc : int \rightarrow int</code>
<code>p := inc</code>	<code>Ok, since p : (int \rightarrow int) ref</code>
<code>(!p) true</code>	<code>Ok, since p : (bool \rightarrow bool) ref</code>

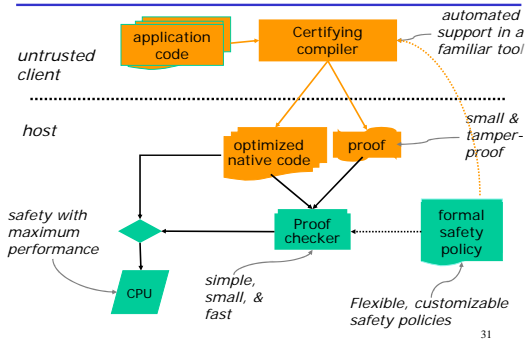
The type system fails to prevent a type error!

29

Java Bytecode Verification

- Java bytecodes constitute a typed programming language with some tricky features.
 - Local variables are reused with different types
 - Allocation and initialization are separate.
 - Subroutines (jsr/ret) provide a little polymorphism.
- A bytecode verifier can be described and implemented as a typechecker.
 - It helps to write formal rules and arguments.
 - Initial versions of Java had security holes in the verifier!
- Subroutines complicate the semantics of bytecodes and were the source of bugs. They are not worth it:
 - In 650K lines of Java code, 230 subroutines, saving 2427 bytes, or 0.02%.
 - More space could be saved by renaming "Java" to "Oak".³⁰

Proof-Carrying Code (PCC)



31

Information-Flow Analysis

- Untrusted code should respect the integrity and confidentiality of data.
- Data can be classified into types with security information, like Secret and Public.
 - Secret data cannot be stored in Public global variables.
 - A typechecker can enforce proper information flow.
- Type Soundness Theorem
 - If the type checker says the program is ok, then when the program is run it does not leak Secret information into Public variables.

32

Three Approaches to Language Semantics

- Operational semantics
 - how to evaluate or execute a program
 - useful for implementing a compiler or interpreter
- Axiomatic semantics
 - given a program and a set of starting states and inputs, what is the set of final states after execution
 - does it include "assertion failed"?
 - useful for proving program correctness (eg. no failed assertions)
- Denotational semantics
 - characterize meaning of each procedure as a mathematical function
 - mathematically heavyweight, de-emphasized in this course

33

A Simple Language of Arithmetic Expressions

syntax and operational semantics

34

Syntax of ARITH

- Concrete syntax: the rules by which programs can be expressed as strings of characters
 - relevant to many of our desiderata
 - (readability, familiarity, speed of compilation, ...)
- The same concepts can be embodied in many concrete, syntaxes, not all of them equally good.
 - Should semicolon separate or terminate statements?
 - How should comments be indicated?
- There are solid principles for concrete syntax.
 - Finite automata and context-free grammars.
 - Automatic parser generators.

35

Abstract Syntax

- We ignore parsing issues and study programs given as abstract syntax trees.
- An abstract syntax tree is a parse tree.
 - More convenient for formal and algorithmic manipulation.
 - Fairly independent of the concrete syntax.
- An abstract syntax for ARITH

$e ::= n$	integer literals
$ e_1 + e_2$	sum
$ e_1 * e_2$	product

36

Analysis of ARITH

- Questions to answer:
 - What is the "meaning" of a given ARITH expression?
 - How would we go about evaluating an ARITH expression?
 - How are the evaluator and the meaning related?

37

An Operational Semantics

- Specifies how expressions should be evaluated.
- Defined by cases on the form of expressions:
 - n evaluates to n
 - n is a normal form, no need to evaluate further
 - $e_1 + e_2$ evaluates to n if
 - e_1 evaluates to n_1
 - e_2 evaluates to n_2
 - and n is the sum of n_1 and n_2
 - $e_1 * e_2$ evaluates to n if
 - e_1 evaluates to n_1
 - e_2 evaluates to n_2
 - and n is the product of n_1 and n_2

38

Alternative Formulation

- Notation: $e \Downarrow n$ means that e evaluates to n
 - This is a judgment a statement about a relation between e and n
- Allows us to write evaluation rules more concisely
 - $n \Downarrow n$
 - $e_1 + e_2 \Downarrow n$ if
 - $e_1 \Downarrow n_1$
 - $e_2 \Downarrow n_2$
 - and n is the sum of n_1 and n_2
 - $e_1 * e_2 \Downarrow n$ if
 - $e_1 \Downarrow n_1$
 - $e_2 \Downarrow n_2$
 - and n is the product of n_1 and n_2

39

Operational Semantics as Inference Rules

$$\frac{n \Downarrow n}{e_1 \Downarrow n_1 \quad e_2 \Downarrow n_2 \quad n \text{ is the sum of } n_1 \text{ and } n_2} e_1 + e_2 \Downarrow n$$
$$\frac{e_1 \Downarrow n_1 \quad e_2 \Downarrow n_2 \quad n \text{ is the product of } n_1 \text{ and } n_2}{e_1 * e_2 \Downarrow n}$$

- Meaning: "above the line" implies "below the line"
- These rules are
 - *evaluation rules* for the big-step operational semantics
 - *derivation rules* for the judgement $e \Downarrow n$

40

Challenge!

- $(3 + 5) * (4 + 2)$ evaluates to what?
- $(3 + 5) * (4 + 2) \Downarrow ?$
- I start, you continue ...

41