CSci 4223
Principles of Programming Languages

Lecture 26
Prof. Clarkson
Spring 2013

Final Exam
- Monday, May 6, 3-5 pm, Funger 209
- Comprehensive: covers entire course
- Same format and kinds of questions as midterms
- May bring laptop to use for SML and Ruby
- May bring three pages of notes
- Exam grades will be uploaded to Blackboard as soon as we finish grading, which might take a week or so
- Final course grades will be made available through the university's usual mechanism
- Please drop by my office to your graded exam
  - Even if you stop by in August!

Course Evaluations
- Your written comments are the #1 way that the course improves!
  - I can't fix something unless I know it's broken
  - I might get rid of something unless I know it's working
- Here's what I wish the university would ask:
  - What are your three least favorite things about the course?
  - What are your three most favorite things about the course?

Victory Lap
Extra trip around the track by the exhausted victors (us)
Reflect on some big themes and perspectives
...the kind of stuff I hope you'll remember five years from now
...not a review for the final

Thank you!
- Huge thank-you to Andrew
  - Graded all the programming problems (getting other people's code to run is hard)
  - Volunteered to lead recitation section
  - Gave two lectures himself

Thank you!
- And a huge thank you to all of you
  - You could have been watching kitteh videos the whole time...instead, you were learning
  - Great attitude about a very different view of software
  - Good in-class questions, good attendance from core group
  - Occasionally laughed at stuff
RETROSPECTIVE
[from lecture 2]

1. Syntax: How do you write language constructs?
2. Semantics: What do programs mean? (Type checking, evaluation rules)
3. Idioms: What are typical patterns for using language features to express your computation?
4. Libraries: What facilities does the language (or a well-known project) provide “standard”? (E.g., file access, data structures)
5. Tools: What do language implementations provide to make your job easier? (E.g., REPL, debugger, GUI editor, …)

These are 5 separate issues
   - In practice, all are essential for good programmers
   - Many people confuse them, but shouldn’t

This course focuses on semantics and idioms

- Libraries and tools are crucial, but throughout your career you’ll learn new ones on the job every year
- Semantics is like a meta-tool: it will help you learn languages
- Idioms will make you a better programmer in those languages
- Syntax is almost always boring
  - A fact to learn, like “GW was founded in 1821”
  - People obsess over subjective preferences (yawn)

[from lecture 4]

- In ML, we create aliases all the time without thinking about it because it is impossible to tell where there is aliasing
  - Example: tl is constant time; does not copy rest of the list
  - So don’t worry and focus on your algorithm

- In Java, programmers are obsessed with aliasing and object identity
  - They have to be (!) so that subsequent assignments affect the right parts of the program
  - Often crucial to make copies in just the right places…

Tuples are just syntactic sugar for records with fields named 1, 2, … n

- Syntactic: Can describe the semantics entirely by the corresponding record syntax
- Sugar: They make the language sweeter
  - There are fewer semantics to worry about
    - Simply understanding the language
    - Simply implementing the language
  - We get the syntactic convenience of tuples

We’ll see many more examples of syntactic sugar
  (already saw Boolean ops and list expressions)

[from lecture 10]

Programming languages are like cars and shoes.
Programming languages are like cultures.
Programming languages are like works of art.

Cars are used for rather different things:
  - Winning the INDY 500
  - Taking kids to soccer practice
  - Off-roading
  - Hauling a mattress
  - Getting the wind in your hair
  - Staying dry in the rain

Shoes:
  - Playing basketball
  - Going to a formal
  - Going to the beach
[from lecture 10]

- To learn how cars work, it might make sense to start with a classic design rather than the latest model
  - A popular car may not be a good car for learning how cars work
  - Even better analogy: beginning pilots don’t fly 747’s

- A good mechanic might have a specialty (2004 Honda Civics), but also understands how all cars work.
  - They don’t get hung up on the syntax (e.g., upholstery color)

Programming languages are tools.

[from lecture 10]: Are all tools the same?

Yes:
- Any input-output behavior implementable in language X is implementable in language Y (see Turing completeness, CSci 3313)
- Java, ML, and a language with one loop and three infinitely-large integers are “the same”

Yes:
- Same fundamentals reappear: variables, abstraction, one-of types, recursive definitions,…

No:
- Some computations easy to express in one language, but awkward to express in another
- Some language features built-in, others have to be “coded up”

...choose the right tool for the job!

[from lecture 10]: Are all languages the same?

- Correct reasoning about programs, interfaces, and compilers requires a precise knowledge of semantics
  - Not “I feel that conditional expressions might work like this”
  - Not “I like curly braces more than parentheses”
  - Much of software development is designing precise interfaces; what a PL means is a really good example

- Idioms make you a better programmer
  - Best to see in multiple settings, including where they shine
  - You’ll understand (insert your favorite language here) better even if I never show you equivalent idioms in that language

Orthogonal language features

<table>
<thead>
<tr>
<th></th>
<th>Dynamically typed</th>
<th>Statically typed</th>
</tr>
</thead>
<tbody>
<tr>
<td>Functional</td>
<td>—</td>
<td>SML</td>
</tr>
<tr>
<td>Object oriented</td>
<td>Ruby</td>
<td>Java</td>
</tr>
</tbody>
</table>

Could fill in missing cell with Racket or Scheme
[from lecture 10]

- No such thing as a "best" PL
- There are good general design principles for PLs
- A good language is a relevant, crisp interface for writing software
- Software leaders should know PL semantics and idioms
- Learning PLs is not about syntactic tricks for small programs
- Functional languages have been on the leading edge for decades
  - Ideas get absorbed by the mainstream, but very slowly
  - Meanwhile, use the ideas to be a better C/java/PHP hacker

[from lecture 12]: Does dynamic scope exist?

- Lexical scope for variables is definitely the right default
  - Very common across languages
- Dynamic scope is occasionally convenient in some situations
  - So some languages (e.g., Perl, Racket) have special ways to do it
  - In some languages, it’s the norm (e.g., Emacs Lisp, LaTeX)
  - But most languages just don’t have it
- If you squint some, exception handling is like dynamic scope:
  - raise e transfers control to the current innermost handler
  - Does not have to be syntactically inside a handle expression (and usually isn’t)

[lecture 13]

<table>
<thead>
<tr>
<th>Weak</th>
<th>Strong</th>
</tr>
</thead>
<tbody>
<tr>
<td>Static</td>
<td>Dynamic</td>
</tr>
<tr>
<td></td>
<td>C, C++, SML, Java, Haskell, Ruby, Python, Scheme</td>
</tr>
</tbody>
</table>

- Manifest vs. implicit:
  - Manifest: type information supplied in source code
    - e.g., C, C++, Java
  - Implicit: type information not supplied in source code
    - Implementation 1: Dynamic typing
      - e.g., Lisp, Python, Ruby, PHP
    - Implementation 2: Type inference
      - e.g., Haskell, SML
- Tradeoff: ease of implementation vs. run-time performance
  - Again, in practice this becomes a spectrum
    - e.g., SML sometimes needs type annotations for records
    - e.g., no reasonable language requires you to write to provide the
      type of $5$ in $x:int = 5$

[lecture 14]: Lambda calculus

- Helps you to understand programming
  - The core of every functional language
- Helps you to understand computation
  - Every "algorithmically computable function" can be expressed in
    lambda calculus (Church’s Thesis)
- Helps you to understand reasoning
  - Modern mathematical logic and automated reasoning based on ideas
    from lambda calculus
  - There is no guaranteed-to-terminate algorithm that can correctly
    determine whether a first-order logic formula is provable (Church’s
    Theorem)

[lecture 17]: The OOP trade-off

Any method that makes calls to overridable methods can have its behavior changed in sub-classes even if it is not overridden
- Maybe on purpose, maybe by mistake
- Observable behavior includes calls-to-overridable methods
- So harder to reason about "the code you’re looking at"
  - Can avoid by disallowing overridding
    - "private" or "final" methods
- So easier for subclasses to affect behavior without copying code
  - Provided method in superclass is not modified later
from lecture 18

- FP and OOP doing the same thing in exact opposite way
  - Organize the program "by rows" or "by columns"
- Which is "most natural" may depend on what you are doing (e.g., an interpreter vs. a GUI) or personal taste
- Code layout is important, but there's no perfect way since software has many dimensions of structure
  - Tools, IDEs can help with multiple "views" (e.g., rows / columns)

from lecture 21

- Misconception: If we are making a new language, we can have whatever typing and subtyping rules we want
- Not if you want to prevent "bad things"
  - Here: No attempt to access non-existent fields
- Our typing rules were sound before we added subtyping
  - We should keep it that way
- Principle of substitutability:
  - If \( t_1 <: t_2 \), then any value of type \( t_1 \) must be usable in every way a \( t_2 \) is
  - Here: Any value of subtype needs all fields any value of supertype has
  - aka behavioral subtyping and Liskov substitution principle

from lecture 22) Classes vs. types

- A class defines an object's behavior
- Subclassing modifies run-time behavior by extension and overriding
- A type defines what fields an object has and what messages it can respond to
- Subtyping determines compile-time behavior by defining when one value is soundly substitutable for another
- These are separate concepts: try to use the terms correctly
  - Java/C# confuse them by conflating classes and types
  - This confusion is convenient in practice

from lecture 24

- Almost anything you might like to check statically is undecidable:
  - Any static checker that always terminates cannot be sound and complete
- Examples:
  - Will this function terminate on some input? On any input?
  - Will this function ever use a variable not in the environment?
  - Will this function treat a string as a function?
  - Will this function divide by zero?
- Undecidability is discussed in CSE 3313
  - The inherent approximation of static checking is probably its most important consequence

BIG IDEAS
1. Syntax and Semantics

- Every language feature can be defined in isolation from other features, with rules for:
  - syntax
  - static semantics (typing rules)
  - dynamic semantics (evaluation rules)

- Divide-and-conquer
  - Entire language can be defined mathematically and precisely
    - SML is. Read The Definition of Standard ML (Revised) (Tofte, Harper, MacQueen, 1997).

2. Syntactic sugar is sweet

- Core language can be small
  - Many ML features we initially learned (e.g., tuples, if expressions) actually don’t have to be part of the language at all
  - And there’s even more I never showed you...

- Extended language, with sugar, is convenient to program in and (curiously enough) has fewer features that programmers have to understand

- Good design to define complicated features in terms of simpler features

3. Benefits of Immutability

- Programmer can alias or copy without worry [lecture 4]
- Parallel programming easier with immutable data [lecture 10]
- Type inference is easier [lecture 13]
- Depth subtyping is sound [lecture 21]

(But mutability is appropriate when you need to model inherently state-based phenomena)

4. OOP and FP are closely related

- So exactly opposite, they’re the same [lecture 18]

- In an OOP language, can code in a functional way
  - using immutable objects [lecture 20]
  - using higher-order functions [lecture 12]

- In a functional language can “code up” objects [lecture 22]

5. Programming languages aren’t magic

- Pattern matching is something you can implement yourself [homework 3]
- Type inference is something you can implement yourself [homework 4]
- Can implement interpreters as straightforward (?) recursive functions over datatypes [homework 5]
- From a small set of elegant primitives with precise semantics, we have built a world that runs on software!
6. CS has an intellectual history created by people

WHAT DID WE ACCOMPLISH?

From the syllabus

- distinguish programming language features based on their static and dynamic semantics,
- describe differences and similarities between two major programming paradigms (objected oriented and functional),
- write programs that utilize features not yet found in main-stream languages,
- analyze the tradeoffs of programming languages and their appropriateness for implementation tasks,
- demonstrate familiarity with new programming languages that are not studied in the first three programming courses in the CS curriculum and, as a by-product, become more proficient in the languages already studied.

Other goals I hope we met

- develop the skills necessary to learn new programming languages quickly
- master specific language concepts and recognize them in strange guises

What next?

- Follow-on courses:
  - Compilers (Implementation of Programming Languages)
  - Semantics (Definition of Programming Languages)
  - ...unfortunately, we don’t have them at GW
- Stay in touch
  - Tell me when 4233 helps you out with future courses (or jobs!)
  - Ask me cool PL questions
  - Drop by to tell me about the rest of your time in CS (and beyond)!
  - I really do like to know

Finally

- The most important idea of this course:
  - complicated artifacts can be broken down into small pieces
  - you can then study those small pieces and understand how they work in isolation
  - then you can understand why their aggregation achieves some goals
- That kind of analysis is applicable anywhere, not just PL.
THE END