Seminar in Programming Languages

Shuly Wintner
Fall 2010-11

Course web site: http://cs.haifa.ac.il/~shuly/teaching/10/PLseminar/
Course Goals

❉ Programming Language Concepts
- A language is a “conceptual universe” (Perlis)
  - Framework for problem-solving
  - Useful concepts and programming methods
- Understand the languages you use, by comparison
- Appreciate history, diversity of ideas in programming
- Be prepared for new programming methods, paradigms, tools

❉ Critical thought
- Identify properties of language, not syntax or sales pitch

❉ Language and implementation
- Every convenience has its cost
  - Recognize the cost of presenting an abstract view of machine
  - Understand trade-offs in programming language design
Language goals and trade-offs
What’s new in programming languages

◆ Commercial trend over past 5 years
  - Increasing use of type-safe languages: Java, C#, ...
  - Scripting languages, other languages for web applications

◆ Teaching trends
  - Java replaces C as most common intro language
    - Less emphasis on how data, control represented in machine

◆ Research and development trends
  - Modularity
    - Java, C++: standardization of new module features
  - Program analysis
    - Automated error detection, programming env, compilation
  - Isolation and security
    - Sandboxing, language-based security, ...
  - Web 2.0
    - Increasing client-side functionality, mashup isolation problems
What’s worth studying?

- Dominant languages and paradigms
  - C, C++, Java
  - Imperative and Object-oriented languages
  - Explosion of programming technologies for the web

- Important implementation ideas

- Performance challenges
  - Concurrency

- Design tradeoffs

- Concepts that research community is exploring for new programming languages and tools
Organization of the course

• Traditional Algol, Pascal constructs
  – Block structure, activation records
  – Scope, control structures, function calls
  – Types and type systems, ...

• Lisp/Scheme concepts
  – higher-order functions and closures, tail recursion
  – exceptions, continuations

• Modularity and program structure

• Specific emphasis on OOP
  – Smalltalk vs C++ vs Java
  – Language design and implementation

• Concurrent and distributed programming
Languages in common use

Compiled by François Labelle from statistics on open-source projects at SourceForge
Languages in common use

From http://www.tiobe.com/index.php/content/paperinfo/tpci/index.html
Language groups

- **Multi-purpose languages**
  - C, C++, Java
  - Visual Basic
  - Object Pascal: Delphi, Kylix, ...
  - Lisp, Scheme, ML

- **Scripting languages**
  - Perl, PHP, Python
  - Shell

- **Special-purpose languages**
  - SQL
  - Prolog
Structure of the course

- Lisp (chapter 3)
- Foundations (chapter 4)
- ML/Algol language summary (chapter 5)
- Types and type inference (chapter 6)
- Scope, functions and storage management (chapter 7)
- Control in sequential languages (chapter 8)
- Modularity and data abstraction (chapter 9)
- Object-oriented languages (chapter 10)
- Simula and Smalltalk (chapter 11)
- C++ (chapter 12)
- Java (chapter 13)
- Concurrent and distributed programming (chapter 14)
- TBD
Methodology

- **Read ahead**
  - You should read each chapter before it is presented

- **Each session will be moderated by one student**

- **The moderator is responsible for running the show**

- **Presentation slides are available! Feel free to modify them as you see fit**

- **The material covered must follow the book chapter, not the slides**
Administration

◆ **Grade based on**
  - Comprehension and preparation of the chapter you moderate
  - Preparation for other sessions
  - Contribution to class discussion

◆ **Office hours**
  - Wednesday 16:00-17:00, Jacobs 403. Phone: (828)8180.

◆ **Assignment of chapters**
  - First come, first served
Fundamentals

Shuly Wintner
Slides courtesy of John C. Mitchell
Topics
Topics

- Computability and partial functions
- Compilers and syntax
- Lambda calculus (as an operational semantics)
- Denotational semantics
- Imperative vs. functional languages
Subtle: “undefined” is not the name of a function value ...
Foundations: Partial, Total Functions

Subtle: “undefined” is not the name of a function value ...
Foundations: Partial, Total Functions

- **Value of an expression may be undefined**
  - Undefined operation, e.g., division by zero
    - $3/0$ has no value
    - implementation may halt with error condition
  - Nontermination
    - $f(x) = \text{if } x=0 \text{ then } 1 \text{ else } f(x-2)$
    - this is a partial function: not defined on all arguments
    - cannot be detected at compile-time; this is halting problem
  - These two cases are
    - “Mathematically” equivalent
    - Operationally different

Subtle: “undefined” is not the name of a function value ...
Partial and Total Functions

- **Total function**: $f(x)$ has a value for every $x$
- **Partial function**: $g(x)$ does not have a value for every $x$
Partial and Total Functions

- Total function: $f(x)$ has a value for every $x$
- Partial function: $g(x)$ does not have a value for every $x$
Mathematics: a function is a set of ordered pairs (graph of function)
Partial and Total Functions

- **Total function** $f: A \to B$ is a subset $f \subseteq A \times B$ with
  - For every $x \in A$, there is some $y \in B$ with $\langle x, y \rangle \in f$ \hspace{1cm} \text{(total)}
  - If $\langle x, y \rangle \in f$ and $\langle x, z \rangle \in f$ then $y = z$ \hspace{1cm} \text{(single-valued)}

- **Partial function** $f: A \to B$ is a subset $f \subseteq A \times B$ with
  - If $\langle x, y \rangle \in f$ and $\langle x, z \rangle \in f$ then $y = z$ \hspace{1cm} \text{(single-valued)}

Programs define partial functions for two reasons
- partial operations (like division)
- nontermination
  \[ f(x) = \text{if } x = 0 \text{ then } 1 \text{ else } f(x-2) \]
Computability

Definition

Function $f$ is computable if some program $P$ computes it:
For any input $x$, the computation $P(x)$ halts with output $f(x)$
Halting function

Decide whether program halts on input

- Given program $P$ and input $x$ to $P$,

$$\text{Halt } (P,x) = \begin{cases} 
\text{yes} & \text{if } P(x) \text{ halts} \\
\text{no} & \text{otherwise}
\end{cases}$$

Clarifications

- Assume program $P$ requires one string input $x$
- Write $P(x)$ for output of $P$ when run in input $x$
- Program $P$ is string input input to Halt

Fact: There is no program for Halt
Suppose P solves variant of halting problem
• On input Q, assume
  \[ P(Q) = \begin{cases} 
    \text{yes} & \text{if } Q(Q) \text{ halts} \\
    \text{no} & \text{otherwise}
  \end{cases} \]

Build program D
• D(Q) = \begin{cases} 
  \text{run forever} & \text{if } Q(Q) \text{ halts} \\
  \text{halt} & \text{if } Q(Q) \text{ runs forever}
\end{cases}

Does this make sense? What can D(D) do?
• If D(D) halts, then D(D) runs forever.
• If D(D) runs forever, then D(D) halts.
• CONTRADICTION: program P must not exist.
Main points about computability

◆ Some functions are computable, some are not
  • Halting problem

◆ Programming language implementation
  • Can report error if program result is undefined due to division by zero, other undefined basic operation
  • Cannot report error if program will not terminate
Syntax and Semantics of Programs
Syntax and Semantics of Programs

❖ Syntax
  • The symbols used to write a program

❖ Semantics
  • The actions that occur when a program is executed

❖ Programming language implementation
  • Syntax → Semantics
  • Transform program syntax into machine instructions that can be executed to cause the correct sequence of actions to occur
Interpreter vs Compiler

Input → Interpreter → Output

Input → Compiler → Target Program → Output
Typical Compiler

Source Program

Lexical Analyzer

Syntax Analyzer

Semantic Analyzer

Intermediate Code Generator

Code Optimizer

Code Generator

Target Program

See summary in course text, compiler books
Brief look at syntax

Grammar

\[ e ::= n | e+e | e-e \]
\[ n ::= d | nd \]
\[ d ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 \]

Expressions in language

\[ e \rightarrow e-e \rightarrow e-e+e \rightarrow n-n+n \rightarrow nd-d+d \rightarrow dd-d+d \]
\[ \rightarrow ... \rightarrow 27 - 4 + 3 \]

Grammar defines a language
Expressions in language derived by sequence of productions
Derivation represented by tree

\[ e \rightarrow e - e \rightarrow e - e + e \rightarrow n - n + n \rightarrow nd - d + d \rightarrow dd - d + d \]

\[ \rightarrow ... \rightarrow 27 - 4 + 3 \]

Tree shows parenthesization of expression
Parsing

- **Given expression find tree**
- **Ambiguity**
  - Expression \(27 - 4 + 3\) can be parsed two ways
  - Problem: \(27 - (4 + 3) \neq (27 - 4) + 3\)
- **Ways to resolve ambiguity**
  - **Precedence**
    - Group * before +
    - Parse \(3 \times 4 + 2\) as \((3 \times 4) + 2\)
  - **Associativity**
    - Parenthesize operators of equal precedence to left (or right)
    - Parse \(3 - 4 + 5\) as \((3 - 4) + 5\)

See book for more info
Chapter assignment

- Lisp (25.10)
- Foundations (1.11)
- ML/Algol language summary (8.11)
- Types and type inference (15.11)
- Scope, functions and storage management (22.11)
- Control in sequential languages (29.11)
- Modularity and Object-oriented languages (6.12 x 2)
- Simula and Smalltalk (13.12)
- C++ (20.12)
- Java (27.12 x 2)
- Concurrent and distributed programming (3.1 x 2)
- TBD (10.1)