

ICS/CSE 141: Programming Languages

Introduction

School of Information and Computer Science
University of California, Irvine

Chris Stork
`cstork@ics.uci.edu`

Welcome!

Your instructor is **Chris Stork**:

- Masters in Mathematics, Heinrich-Heine-Universität Düsseldorf, Germany, 1998
- One or two quarters away from finishing my PhD thesis in computer science (*Compressed Abstract Syntax Trees as a Mobile Code Format*)
- Interests: programming languages, compiler construction, computer architecture, security, peer-to-peer, . . .

Overview

1. The *What, Why, and How* of this course
2. Administrivia
3. Survey
4. Motivation
5. Preliminaries
6. Functions
7. Computability
8. Scheme

The *What*, *Why*, and *How* of this course

What

What will we learn in this course?

- design space for programming languages
- evaluate programming languages
- programming concepts & paradigms
- a bit of history

Why

Why learn about programming languages?

- choose an **appropriate** language for a given problem
- improve your **programming skills**
- understand the **benefits** and **cost** of particular language features
- appreciate the **diversity** of programming languages

How

How will we learn?

- look at a lot of code in different languages
- program ourselves
 - implement a programming languages (or features)
 - use different programming languages
- reason about programs
- read some papers

Prerequisites

- enjoy programming
- know Java or C++
- have an idea about
 - ICS23/22: computer architecture, memory management, pointers, assembly programming,
 - ICS51: ADTs, recursion, simple understanding of object-oriented programming and functional programming
 - propositional and first order logic
- Unix knowledge might be helpful. . .

Administriva

<http://www.ics.uci.edu/~cstork/ics141/>

People

- Instructor: Chris Stork, Cand. Ph.D., UCI
Email: cstork@ics.uci.edu
Office hours: after class and by appointment
- Teaching Assistant: Chris Fensch, Cand. Ph.D., UCI
Email: ics141ta@ics.uci.edu
Office hours: Lab sessions and by appointment

Grades, Assignments,...

Grades are on a 0–100 points scale which will be translated to A,B,... at end of session.

- **3 quizzes** on Mondays (30%)
- **4 homework/programming assignments** due on Wednesdays (40%)
- **final exam** (30%)

Some **reading assignments**

Estimated time requirements: \approx **30 hours/week**

Discussions

Attend the discussion sessions!

Monitor the EEE course **mailing list** for my announcements!

Also use it for out-of-class discussions that are of interest to everybody.

Did you receive the test mail?

If not, go to <http://eee.uci.edu/start/> to get yourself set up.

Homework Submission

All homeworks are **submitted electronically** via the Checkmate system: <http://checkmate.ics.uci.edu/>

Non-programming homework is due as **L^AT_EX** and **PDF** documents.
(Details later.)

Non-ICS students: You need an ICS account (in addition to UCInetID)! Come to CS 364 with proof on enrollment and picture ID and talk to lab assistant.

Check out <http://www.ics.uci.edu/~lab> for details and general lab policies.

Required Books

**John C. Mitchell: *Concepts in Programming Languages*,
Cambridge University Press, 2003.**

- Standard textbook structure, however quite recent and written by an active researcher in the area.
- Emphasizes functional programming and type systems, a little more theoretical than other texts.
- The course does **not** cover everything in the book, and the book does **not** cover everything in the course, especially in the beginning.
- Terminology will be based on this book.

Recommended Books

Robert Sebesta: *Concepts of Programming Languages*, 6th edition, Addison-Wesley, 2003.

- Standard @ UCI
- lots of history
- outdated treatment of functional and logical PLs

Ravi Sethi: *Programming Languages: Concepts and Constructs*, 2nd edition, Addison-Wesley, 1996.

- Just discovered. . .
- seems well done
- but a little older (lacks Java & Haskell)

Other Books: Interesting Alternatives

Daniel Friedman, Mitchell Wand, Christopher Haynes:
***Essentials of Programming Languages*, 2nd edition, MIT Press,**
2001.

- Implement interpreter for everything in Scheme
- Tough! Good for after this course.

Michael L. Scott: *Programming Language Pragmatics*, Morgan Kaufmann, 2000.

- Good mix of programming languages, computer architecture, and compiler construction

(See course web page for more!)

Collaboration, etc.

Collaboration in form of discussions is **strongly encouraged**, but they must be **acknowledged**.

No collaboration on code! **Don't even look** at other's code.

Researching solutions on the Web or in the library is good, but must be acknowledged as well.

[UCI's cheating policy](#) applies and will be enforced!

How is this course different from others?

Mix of two traditional approaches:

- **single-language/interpreter-based/features**
 - Everything in Scheme (or even Java)
 - Friedman/Wand/Haynes or Ramsey/Kamin
- **multi-language/historical/social**
 - Sebesta, Ghezzi/Jazayeri, or Sethi

Should prepare you for next generation of PLs!

Survey

Motivation

Philosophy

That which can be said, can be said clearly.

Wittgenstein

The purpose of language is simply to convey meaning.

Confucius

Linguistics

The limits of my language are the limits of my world.

Wittgenstein

Sapir-Whorf Hypothesis (Linguistic Determinism):

“All thoughts are constraint by language.”

George Orwell’s novel “1984”:

newspeak vs. oldspeak

Programming Languages, what for?

- Communicate

Programming Languages, what for?

- Communicate
 - human \rightarrow computer... obviously!

Programming Languages, what for?

- Communicate
 - human → computer... obviously!
 - human → human... e.g., comments, literate programming

Programming Languages, what for?

- Communicate
 - human \rightarrow computer... obviously!
 - human \rightarrow human... e.g., comments, literate programming
 - (computer \rightarrow computer)

Programming Languages, what for?

- Communicate
 - human \rightarrow computer... obviously!
 - human \rightarrow human... e.g., comments, literate programming
 - (computer \rightarrow computer)
- Model the world
 - provide concepts, e.g., bits, objects, functions, rules
 - protect us from mistakes (the good newspeak!)

Programming Languages, what for?

- Communicate
 - human \rightarrow computer... obviously!
 - human \rightarrow human... e.g., comments, literate programming
 - (computer \rightarrow computer)
- Model the world
 - provide concepts, e.g., bits, objects, functions, rules
 - protect us from mistakes (the good newspeak!)
- Support creation, analysis, and easy manipulation of programs

Abstraction vs. Efficiency

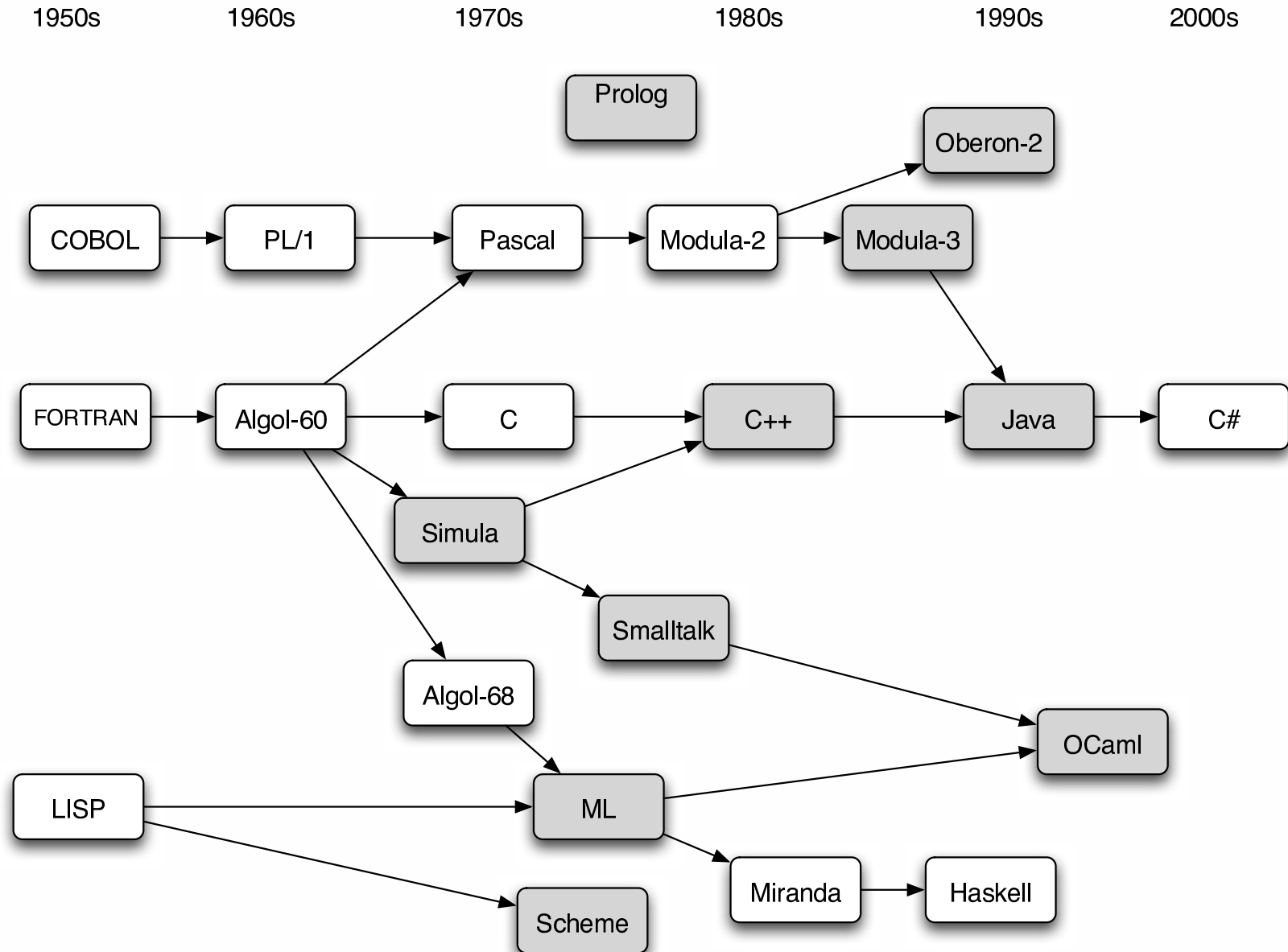
By relieving the brain of all unnecessary work, a good notation sets it free to concentrate on more advanced problems, and, in effect, increases the mental power of the race.

Alfred North Whitehead (1861–1947)

At the beginning, **efficiency** was the overarching concern in the design of programming languages.

But given the ever increasing computing power, raising the **abstraction** level of programming languages becomes feasible and, in the face of the human cost, even a necessity.

Evolution of (Some) Programming Languages



War and Revolution

“Language Wars”

- **Tcl vs. Guile** (search for "Why you should not use Tcl" on groups.google.com)
- **Perl vs. Python** (Pie-thon contest at OSCON 2004
<http://www.oreillynet.com/pub/a/oscon2004/friday/in>)
- **C++ vs. Java vs. C#**

War and Revolution

“Language Wars”

- **Tcl vs. Guile** (search for "Why you should not use Tcl" on groups.google.com)
- **Perl vs. Python** (Pie-thon contest at OSCON 2004
<http://www.oreillynet.com/pub/a/oscon2004/friday/in>)
- **C++ vs. Java vs. C#**

“Language Revolutions”

- What was the last big language revolution? (Java? C#? What was new?)
- What will be the next one? (Aspects, web services? XML? Language at all? Pervasive Computing? Seamless computing?)

Preliminaries

Programming Domains

- **Scientific Applications** (arrays, loops, floating point arithmetic):
Fortran
- **Business Applications** (reports, decimal arithmetic):
Cobol
- **Artificial Intelligence** (symbolic computation, lists/trees):
LISP, Scheme
- **Systems Programming** (fast, low-level):
PL/I, Extended ALGOL, C
- **Scripting Languages** (glueing, CGI, rapid prototyping):
sh, awk, VisualBasic, tc1, Perl, Python, JavaScript, PHP
- **Special-Purpose Languages** . . .

What makes a Programming Languages?

- **Definition**

- Syntax (EBNF)
- Static semantics (Attribute grammar, type checking)
- Dynamic semantics (operational, denotational, or axiomatic)

- **Implementation**

- Interpreter or compiler
- Libraries
- Other tools: profiler, debugger, IDE

PL Evaluation (Buzzwords)

- **Readability** (not only syntax!)
 - Perl: “There’s more than one way to do it”
 - Python: “There should be only one way to do it”
- **Writability**
 - Perl: Write-once language ;-)
- **Simplicity**
 - Language definition short and easy to read (Scheme, Oberon)
- **Orthogonality**
 - small number of primitive constructs can be freely combined

(complete taxonomy in Sebesta)

PL Evaluation (continued)

- **Reliability**

- type checking
- exception handling
- contracts
- aliasing

- **Costs**

- programs resource consumption (memory & time)
- implementation time (human programmer most expensive!)

Programming Language Communities

Social aspects play a role too! Every language supports certain ways of thinking and attracts different kinds of people.

Programming language (online) communities can be very different.

PHP vs. Perl vs. Python vs. Java vs. Scheme vs. Haskell vs. . . .

Some languages live in distinctive niches, e.g. Lua (scripting within gaming community)

Related Topics: SE & CA

- **Software Engineering**

- How should we build, test, optimize, and maintain programs?
- For example, Java's `assert` statement
- Support for refactoring

- **Computer Architecture**

- We have to run our software on actual machines after all!
- von Neumann Architecture
- The LISP machine

Related Topics: Compilers & Interpreters

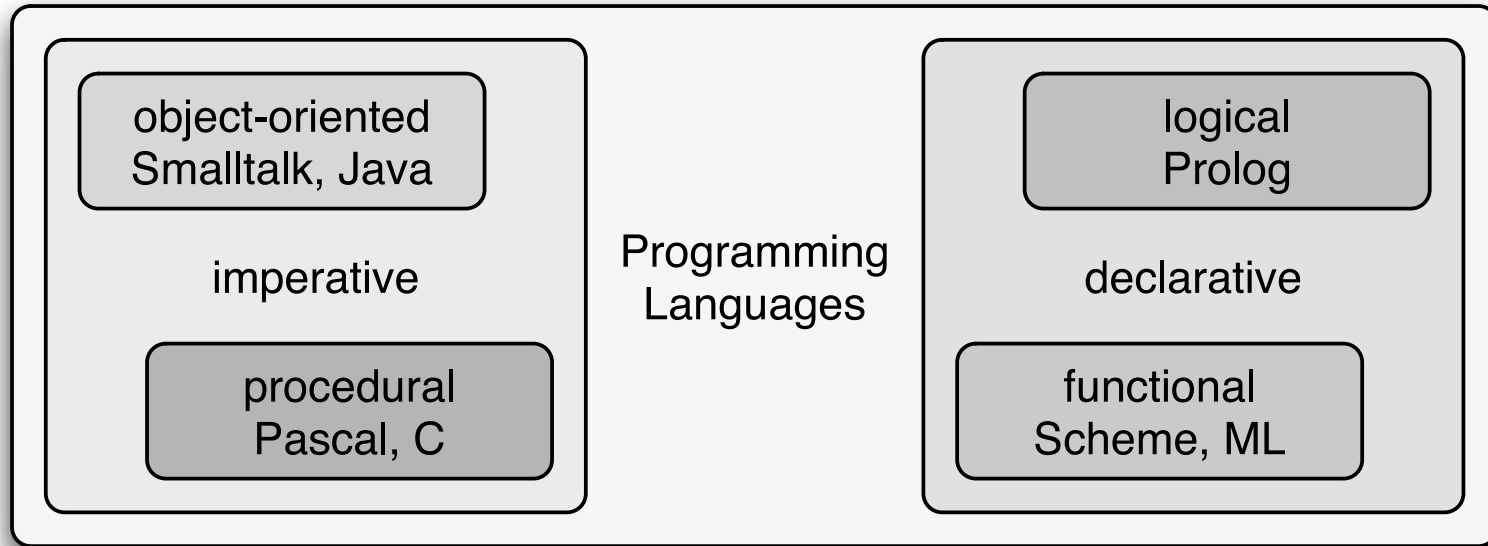
Compilers transform programs

- Parser: Source \Rightarrow Abstract Syntax Tree (AST)
- Type checker: AST \Rightarrow Intermediate Representation (IR)
- Optimizers: IR \Rightarrow IR
- Code Generator / Assembler: IR \Rightarrow Machine Code

Interpreters execute programs

Language design and implementation influence each other

Classification of Programming Languages



This classification is a little old-fashioned and simplistic.

- What about functional objects? Where do modules fit?
Components?

Functions

Functions: Intro

- For a function $f : A \rightarrow B$ we call
 - A the **domain** of f
 - B the **range** or **codomain** of f
- $f : A \rightarrow B$ is a set of **ordered pairs** $f \subseteq A \times B$ with
 - $(x, y) \in f \wedge (x, z) \in f \Rightarrow y = z$ “at most one value”
 - $\forall x \in A : \exists y \in B : (x, y) \in f$ “at least one value”
- Functions can take **multiple** arguments
 - $f : A_1 \times A_2 \times \dots \times A_n \rightarrow B$

Total and Partial Functions

- A function as defined above is called **total**.
 - It has one value for **every** argument.
- Certain functions are **undefined** for some arguments. They are called **partial**.
 - $f : \mathbf{R} \rightarrow \mathbf{R}$ defined by $f(x) = 1/x$
 - $f : \mathbf{N} \rightarrow \mathbf{N}$ defined by $f(x) = x - 1$
- More precisely, we should write
 - $f : \mathbf{R} \setminus \{0\} \rightarrow \mathbf{R}$ defined by $f(x) = 1/x$
 - $f : \mathbf{N} \setminus \{0, 1\} \rightarrow \mathbf{N}$ defined by $f(x) = x - 2$

Computability

Functions and Programs

- We can view **programs** as **functions**
 - “Given some **input**, a program computes some **output**”
- Programs are **partial** functions
 - for some inputs, runtime errors can occur **error termination**
 - for some inputs, the program may run forever
non-termination
- Compare two examples (in ML):
 - `fun f(x) = x div 0`
 - `fun f(x) = if x = 0 then 0 else x + f(x-2)`

Definition

A partial function $f : A \rightarrow B$ is **computable** if there exists a program P that

- given an argument x as input
- yields $y = f(x)$ as output.

Definition

A partial function $f : A \rightarrow B$ is **computable** if there exists a program P that

- given an argument x as input
- yields $y = f(x)$ as output.

Does this definition depend on the programming language in which P is written?

Definition

A partial function $f : A \rightarrow B$ is **computable** if there exists a program P that

- given an argument x as input
- yields $y = f(x)$ as output.

Does this definition depend on the programming language in which P is written?

Are there PLs that can compute “more” functions than others?

Church's Thesis

All sufficiently capable machines or languages can (theoretically) compute the same class of functions.

“Sufficiently capable” is defined by examples:

- Turing machine (an idealized, very simple computer)
- λ -calculus (we'll see!)
- ...

Therefore computability is **not** a distinguishing feature of programming languages!

Partial Functions and Composability

Partial Functions are bad for programming since they hinder composability of functions:

$$y = f(g(x))$$

defined if and only if

$$y' = g(x) \text{ and } f(y') \text{ are defined}$$

Can't we automatically determine which inputs are allowed and have the computer check that for us?

Partial Functions and Termination

Assuming that the partial function f is defined by the program P , we need to be able to tell for which inputs x the output of $P(x)$ is defined.

Runtime Error Return an error code or throw an exception.

Conceptually, error codes or exceptions can be added to the codomain.

Non-Termination Hmm, find out for which inputs x the program P terminates. . .

Halting Problem

Given a program P and an input x , does $P(x)$ terminate?

So, we are looking for a program $H(P, x)$ that terminates and returns either “yes, P halts on x ” or “no, P does not halt on x ”

Surprising Answer: No such program exists, the halting problem is **undecidable!** Proof by contradiction (see Mitchell, p. 14).

Implication: Compilers (and therefore PLs) are restricted in the kind of checks that they can perform for the programmer.

Halting Problem: Proof I

Assume such a program H exists. That means

$$H(P, x) \text{ returns } \begin{cases} \text{“halts”} & \text{if } P(x) \text{ terminates} \\ \text{“does not halt”} & \text{if } P(x) \text{ does not terminate} \end{cases}$$

Now, build a new program $D(P)$, which

- terminates if $H(P, P)$ returns “does not halt”
- does not terminate if $H(P, P)$ returns “halts”

Halting Problem: Proof II

Does $D(D)$ terminate or not?

- If it terminates then $H(D, D)$ returned “does not halt”
- If it does not terminate then $H(D, D)$ returned “halts”

This is a *contradiction to our original assumption* that there exists a program H that solves the halting problem. Therefore no such H can exist!

Lisp & Scheme

LISP

LISP (LISt Processing) developed around 1960 by John McCarthy

- Symbolic computation, e.g. integration of (mathematical) functions
- Predominant language for AI applications (still?)
- “Common Lisp, mud ball of strength, is the acting patriarch of the Lisp family.”—[The Tao of Recursion](#)
- Plethora of dialects.

Scheme

Scheme developed around 1975 by Guy Steele (same guy who worked on Java)

- Language design, e.g. making Lisp better, theoretical foundations
- Cleaned-up and simplified LISP: It's complete standard, the [Revised⁵ Scheme Report \(R5RS\)](#) has only 50 pages (Common Lisp reference has over 1000 pages!)
- We'll do Scheme!
- Don't be confused by all these dialects out there! Stick to R5SR.

Lists

As in Lisp, everything in Scheme consists of lists (that's a lie!):

Data (1 2 3 4)
 ("text string" 100 sym (3 4))

Code (f x y)
 (+ 1 2)

Prefix Syntax

Simple expressions (“ \Rightarrow ” means “evaluates to”):

- $(+ 1 2) \Rightarrow 3$
- $(* 2 2 2) \Rightarrow 8$
- $(+ (* 1 2) (* 3 4)) \Rightarrow 14$
- $(< 1 2) \Rightarrow \#t$ (“true”)
- $(< 2 1) \Rightarrow \#f$ (“false”)
- $(< 1 3 3) \Rightarrow \#f$
- $(<= 1 3 3) \Rightarrow \#t$

quote & eval

- Using `quote` to avoid evaluation

– `(+ 1 2)` \Rightarrow 3

– `(quote (+ 1 2))` \Rightarrow `(+ 1 2)`

– `'(+ 1 2)` \Rightarrow `(+ 1 2)` (shorthand)

- Using `eval` to evaluate a list

– `'(+ 1 2)` \Rightarrow `(+ 1 2)`

– `(eval '(+ 1 2))` \Rightarrow 3

- Code **is** data and data **is** code! Do that in C++ or Java...

Building Lists and taking them apart

- Building lists using `cons`

- `() ⇒ ()` (“empty list”)
- `(cons 'a ()) ⇒ (a)`
- `(cons 'b (cons 'a ())) ⇒ (b a)`
- `(cons 'b '(a)) ⇒ (b a)`

- Taking lists apart using `car` and `cdr`

- `(car '(a b c)) ⇒ a` (“first element in list”)
- `(cdr '(a b c)) ⇒ (b c)` (“rest of list”)
- `(car '())` or `(cdr '()) ⇒ error`

DrScheme

Nice and flexible environment for Scheme hacking (and then some)

- Version 207 installed on lab machines
- Graphical interface, shows control flow and data flow
- Available for lots of platforms, install on your own machine?
- Check out <http://www.drscheme.org/> for more...

Important!

- Select “R5RS” language for this course!
- If an example handed out does not work, check language setting!

Dotted Pairs

Why was that a lie? Because the only real basic data structure in Scheme is the **dotted pair** or **cons cell**, written as

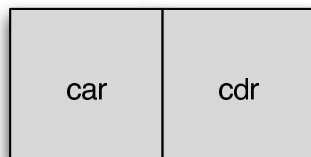
$$(a . b)$$

Where `a` and `b` are called the `car` and `cdr`, respectively. (These are historical names.)

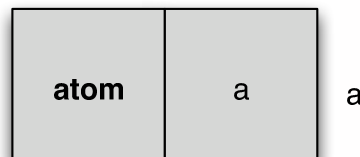
$$(\text{car } '(a . b)) \Rightarrow a$$
$$(\text{cdr } '(a . b)) \Rightarrow b$$

Pairs & Lists

cons cell



'a



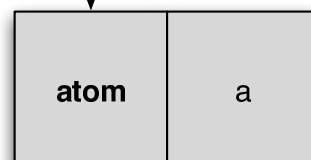
a

(cons 'a ())

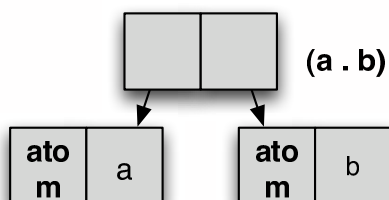


(a . ())
(a)

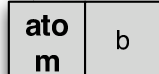
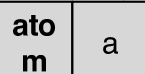
o



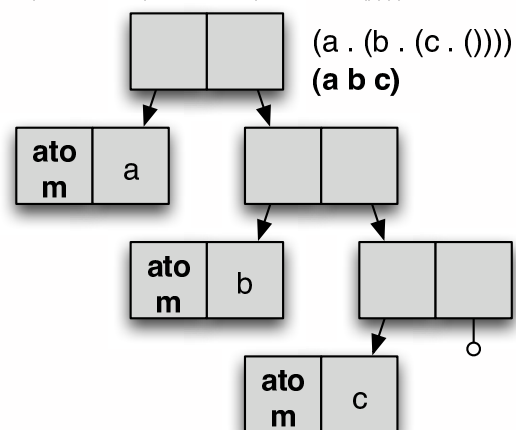
(cons 'a 'b)



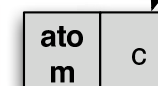
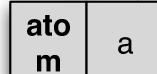
(a . b)



(cons 'a (cons 'b (cons 'c ())))



(a . (b . (c . ())))
(a b c)



o

Conditionals

- Basic **if** construct (“ \Rightarrow ” means “evaluates to”):
 - Syntax: `(if <test> <consequent> <alternate>)`
 - `(if (> 3 2) 'yes 'no) \Rightarrow yes`
 - `(if (> 2 3) 'yes 'no) \Rightarrow no`
- The much more common **cond** construct:
 - Syntax: `(cond <clause> <clause> ...)`, each clause has the form `(<test> <expression>)` in turn.
 - `(cond ((< 2 1) 10) (#t 42)) \Rightarrow 42`
 - `(cond ((< 1 2) 10) (#t 42)) \Rightarrow 10`

Booleans

- Boolean expressions (“ \Rightarrow ” means “evaluates to”):
 - `(and (= 2 2) (> 2 1))` \Rightarrow `#t`
 - `(and (= 2 2) (= 1 1) (> 1 2))` \Rightarrow `#f`
 - `(and #t (/ 3 0))` \Rightarrow **error**
 - `(and #f (/ 3 0))` \Rightarrow `#f` (“short circuit”)
 - `(or (= 2 2) (> 2 1))` \Rightarrow `#t`
 - `(or (= 2 2) (= 1 1) (> 1 2))` \Rightarrow `#t`
 - `(or #t (/ 3 0))` \Rightarrow `#t` (“short circuit”)
 - `(not #t)` \Rightarrow `#f`

Debugging

```
(display <info>)
```

```
(newline)
```

at well chosen places

Assignments

- Install and play with DrScheme
- Read chapters 2 & 3, and sections 4.1 & 4.2 in Mitchell
- Take a look at [Revised⁵ Scheme Report \(R5RS\)](#)
- Very good hands-on Scheme intro:
<http://www.math.grinnell.edu/~stone/scheme-wel>
- For the historical interest see J. McCarthy,
[Recursive functions of symbolic expressions and their computation by machine](#),
Comm. ACM 3, 4 (1960) 184-195.

Thank You!