PAUL G. ALLEN SCHOOL
OF COMPUTER SCIENCE & ENGINEERING

CSE341: Programming Languages

Lecture 1
Course Mechanics
ML Variable Bindings

Dan Grossman

Spring 2019

Welcome!

We have 10 weeks to learn *the fundamental concepts* of programming languages

With hard work, patience, and an open mind, this course makes you a much better programmer

- Even in languages we won't use
- Learn the core ideas around which *every* language is built, despite countless surface-level differences and variations
- *Poor* course summary: "Uses ML, Racket, and Ruby"

Today's class:

- Course mechanics
- *[A rain-check on motivation]*
- Dive into ML: Homework 1 due Wednesday of next week

Concise to-do list

In the next 24-48 hours:

1. Read course web page: http://courses.cs.washington.edu/courses/cse341/19sp/
2. Read all course policies (4 short documents on web page)
3. Adjust class email-list settings as necessary
4. Complete Homework 0 (survey worth 0 points)

5. Get set up using Emacs [optional; recommended] and ML
   - Installation/configuration/use instructions on web page
   - Essential; non-intellectual
     - No reason to delay!

Who: Course Staff

Dan Grossman: Faculty, 341 my favorite course / area of expertise

Seven (!!) *great* TAs

Get to know us!

Staying in touch

- Course email list: **cse341a_sp19@u.washington.edu**
  - Students and staff already subscribed
  - You must get announcements sent there
  - Fairly low traffic
- Course staff: **cse341-staff@cs.washington.edu**
  plus individual emails
- Message Board
  - For appropriate discussions; TAs will monitor
  - Optional/encouraged, won't use for important announcements
- Anonymous feedback link on webpage
  - For good and bad: If you don't tell me, I don't know

Lecture: Dan

- Slides, code, and reading notes / videos posted
  - May be revised after class
  - *Take notes*: materials may not describe everything
    - Slides in particular are *visual aids* for me to use
- Ask questions, focus on key ideas
- Engage actively
  - Arrive *punctually* (beginning matters most!) and well-rested
    - Just like you will for the midterm!
  - *Write* down ideas and code as we go
  - If attending and paying attention is a poor use of your time, one of us is doing something wrong

Section

- Required: will usually cover new material
- Sometimes more language or environment details
- Sometimes main ideas needed for homework
- *Will* meet this week: using Emacs and ML

Material often also covered in reading notes / videos

Reading Notes and Videos

- Posted for each “course unit”
  - Go over most (all?) of the material (and some extra stuff?)

- So why come to class?
  - Materials let us make class-time much more useful and interactive
    - Answer questions without being rushed because *occasionally* “didn’t get to X; read/watch about it”
    - Can point to optional topics/videos
    - Can try different things in class, not just recite things

- Don’t need other textbooks – I’ve roughly made one myself

Office hours

- Regular hours and locations on course web
  - Changes as necessary announced on email list

- Use them
  - *Please visit me*
  - Ideally not *just* for homework questions (but that's good too)

Homework

- Seven total
- To be done individually
- Doing the homework involves:
  1. Understanding the concepts being addressed
  2. Writing code demonstrating understanding of the concepts
  3. Testing your code to ensure you understand and have correct programs
  4. “Playing around” with variations, incorrect answers, etc.

  Only (2) is graded, but focusing on (2) makes homework harder
- Challenge problems: Low points/difficulty ratio

Note my writing style

- Homeworks tend to be worded very precisely and concisely
  - I write like a computer scientist (a good thing!)
  - Technical issues deserve precise technical writing
  - Conciseness values your time as a reader
  - You should try to be precise too
- *Skimming or not understanding why a word or phrase was chosen can make the homework harder*
- By all means ask if a problem is confusing
  - Being confused is normal and understandable
  - And I may have made a mistake

Academic Integrity

- Read the course policy carefully
  - Clearly explains how you can and cannot get/provide help on homework and projects
- Always explain any unconventional action
- I have promoted and enforced academic integrity since I was a freshman
  - Great trust with little sympathy for violations
  - Honest work is the most important feature of a university
- This course especially: Do not web-search for homework solutions! We will check!

Exams

- Midterm: Friday May 3, in class
- Final: Thursday June 13, 8:30-10:20AM
- Same concepts, but different format from homework
  - More conceptual (but write code too)
  - Will post old exams
  - Closed book/notes, but you bring one sheet with whatever you want on it

Coursera (more info in document)

- I've taught this material to thousands of people around the world
  - A lot of work and extremely rewarding
- You are not allowed to participate in that class!
- This should have little impact on you
  - Two courses are separate
  - 341 is a great class and staff is committed to this offering being the best ever
- But this is a neat thing you are likely curious about…

More Coursera

- Why did I do a MOOC?
  - Have more impact (like a textbook) for my favorite stuff!
  - Experiment with where higher-ed might be going

- So why pay tuition?
  - Personal attention from humans
  - Homeworks/exams with open-ended questions
  - Class will adjust as needed
  - We can be sure you actually learned
  - Course is part of a coherent curriculum
  - Beyond the classroom: job fairs, advisors, social, …

Has Coursera help/hurt 341?

- Biggest risks
  - Becomes easier to cheat – don't! (And I've changed things)
  - I become too resistant to change – hope not!

- Benefits too
  - More resources: videos, grading scripts, …
  - Way fewer typos
  - Taking the "VIP version" of a more well-known course
  - Change the world to be more 341-friendly

Questions?

Anything I forgot about course mechanics before we discuss, you know, programming languages?

What this course is about

- Many essential concepts relevant in any programming language
  - And how these pieces fit together
- Use ML, Racket, and Ruby languages:
  - They let many of the concepts “shine”
  - Using multiple languages shows how the same concept can “look different” or actually be slightly different
  - In many ways simpler than Java
- Big focus on *functional programming*
  - Not using *mutation* (assignment statements) (!)
  - Using *first-class functions* (can’t explain that yet)
  - But many other topics too

Why learn this?

This is the “normal” place for course motivation

- Why learn this material?

But we don’t have enough shared vocabulary/experience yet

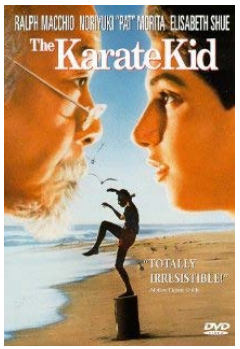
- So 3-4 week delay on motivation for functional programming
- I promise full motivation: delay is worth it
- (Will motivate immutable data at end of “Unit 1”)

My claim

Learning to think about software in this "PL" way will make you a better programmer even if/when you go back to old ways

It will also give you the mental tools and experience you need for a lifetime of confidently picking up new languages and ideas

[Somewhat in the style of *The Karate Kid* movies (1984, 2010)]



A strange environment

- Next 4-5 weeks will use
  - ML language
  - Emacs editor
  - Read-eval-print-loop (REPL) for evaluating programs
- Need to get things installed and configured
  - Either in the Allen School labs or your own machine
  - We've written thorough instructions (questions welcome)
- Only then can you focus on the content of Homework 1
- Working in strange environments is a CSE life skill

Mindset

- "Let go" of all programming languages you already know

- For now, treat ML as a "totally new thing"
  - Time later to compare/contrast to what you know
  - For now, "oh that seems kind of like this thing in [Java]" will confuse you, slow you down, and you will learn less

- Start from a blank file…

A very simple ML program

[The same program we just wrote in Emacs; here for convenience if reviewing the slides]

```
(* My first ML program *)

val x = 34;

val y = 17;

val z = (x + y) + (y + 2);

val q = z + 1;

val abs_of_z = if z < 0 then 0 - z else z;

val abs_of_z_simpler = abs z
```

A variable binding

```
val z = (x + y) + (y + 2); (* comment *)
```

More generally:

```
val x = e;
```

- *Syntax*:
  - *Keyword* **`val`** and *punctuation* **`=`** and **`;`**
  - *Variable* ***`x`***
  - *Expression* ***`e`***
    - Many forms of these, most containing *subexpressions*

The semantics

- Syntax is just how you write something

- Semantics is what that something means
  - Type-checking (before program runs)
  - Evaluation (as program runs)

- For variable bindings:
  - Type-check expression and extend static environment
  - Evaluate expression and extend dynamic environment

So what is the precise syntax, type-checking rules, and evaluation rules for various expressions? Good question!

ML, carefully, so far

- A program is a sequence of *bindings*
- *Type-check* each binding in order using the *static environment* produced by the previous bindings
- *Evaluate* each binding in order using the *dynamic environment* produced by the previous bindings
  - Dynamic environment holds *values*, the results of evaluating expressions
- So far, the only kind of binding is a *variable binding*
  - More soon

Expressions

- We have seen many kinds of expressions:

  `34` **`true`** **`false`** **`x`** ***`e1+e2`*** ***`e1<e2`***

  `if` ***`e1`*** **`then`** ***`e2`*** **`else`** ***`e3`***

- Can get arbitrarily large since any subexpression can contain subsubexpressions, etc.

- Every kind of expression has
  1. Syntax
  2. Type-checking rules
     - Produces a type or fails (with a bad error message ☹)
     - Types so far: **`int`** **`bool`** **`unit`**
  3. Evaluation rules (used only on things that type-check)
     - Produces a value (or exception or infinite-loop)

Variables

- Syntax:

  sequence of letters, digits, _, not starting with digit

- Type-checking:

  Look up type in current static environment

  – If not there fail

- Evaluation:

  Look up value in current dynamic environment

Addition

- Syntax:

  e1 **+** ***e2*** where ***e1*** and ***e2*** are expressions

- Type-checking:

  If ***e1*** and ***e2*** have type **`int`**,

  then ***e1*** **+** ***e2*** has type **`int`**

- Evaluation:

  If ***e1*** evaluates to **`v1`** and ***e2*** evaluates to **`v2`**,

  then ***e1*** **+** ***e2*** evaluates to sum of **`v1`** and **`v2`**

Values

- All values are expressions
- Not all expressions are values
- A value “evaluates to itself” in “zero steps”
- Examples:
  - **`34`**, **`17`**, **`42`** have type **`int`**
  - **`true`**, **`false`** have type **`bool`**
  - **`()`** has type **`unit`**

Slightly tougher ones

What are the syntax, typing rules, and evaluation rules for conditional expressions?

What are the syntax, typing rules, and evaluation rules for less-than expressions?

The foundation we need

We have many more types, expression forms, and binding forms to learn before we can write “anything interesting”

Syntax, typing rules, evaluation rules will guide us the whole way!

For Homework 1: functions, pairs, conditionals, lists, options, and local bindings

- Earlier problems require less

Will not add (or need):

- Mutation (a.k.a. assignment): use new bindings instead
- Statements: everything is an expression
- Loops: use recursion instead

PAUL G. ALLEN SCHOOL
OF COMPUTER SCIENCE & ENGINEERING

CSE341: Programming Languages

Lecture 2
Functions, Pairs, Lists

Dan Grossman

Spring 2019

Function definitions

Functions: the most important building block in the whole course

- Like Java methods, have arguments and result
- But no classes, **`this`**, **`return`**, etc.

Example *function binding*:

```
(* Note: correct only if y>=0 *)

fun pow (x : int, y : int) =
  if y=0
  then 1
  else x * pow(x,y-1)
```

Note: The *body* includes a (recursive) *function call*: **`pow(x,y-1)`**

Example, extended

```
fun pow (x : int, y : int) =
  if y=0
  then 1
  else x * pow(x,y-1)

fun cube (x : int) =
  pow (x,3)

val sixtyfour = cube 4

val fortytwo = pow(2,2+2) + pow(4,2) + cube(2) + 2
```

Some gotchas

Three common “gotchas”

- Bad error messages if you mess up function-argument syntax
- The use of **`*`** in type syntax is not multiplication
  - Example: **`int * int -> int`**
  - In expressions, **`*`** is multiplication: **`x * pow(x,y-1)`**
- Cannot refer to later function bindings
  - That’s simply ML’s rule
  - Helper functions must come before their uses
  - Need special construct for *mutual recursion* (later)

Recursion

- If you're not yet comfortable with recursion, you will be soon ☺
  - Will use for most functions taking or returning lists

- "Makes sense" because calls to same function solve "simpler" problems

- Recursion more powerful than loops
  - We won't use a single loop in ML
  - Loops often (not always) obscure simple, elegant solutions

Function bindings: 3 questions

- Syntax: `fun x0 (x1 : t1, … , xn : tn) = e`
  - (Will generalize in later lecture)

- Evaluation: ***A function is a value!*** (No evaluation yet)
  - Adds **`x0`** to environment so *later* expressions can *call* it
  - (Function-call semantics will also allow recursion)

- Type-checking:
  - Adds binding **`x0 : (t1 * … * tn) -> t`** if:
  - Can type-check body **`e`** to have type **`t`** in the static environment containing:
    - "Enclosing" static environment (earlier bindings)
    - **`x1 : t1, …, xn : tn`** (arguments with their types)
    - **`x0 : (t1 * … * tn) -> t`** (for recursion)

More on type-checking

```
fun x0 (x1 : t1, … , xn : tn) = e
```

- New kind of type: **`(t1 * … * tn) -> t`**
  - Result type on right
  - The overall type-checking result is to give **`x0`** this type in rest of program (unlike Java, not for earlier bindings)
  - Arguments can be used only in **`e`** (unsurprising)

- Because evaluation of a call to **`x0`** will return result of evaluating **`e`**, the return type of **`x0`** is the type of **`e`**

- The type-checker "magically" figures out **`t`** if such a **`t`** exists
  - Later lecture: Requires some cleverness due to recursion
  - More magic after hw1: Later can omit argument types too

Function Calls

A new kind of expression: 3 questions

Syntax: ***e0 (e1,…,en)***

- (Will generalize later)
- Parentheses optional if there is exactly one argument

Type-checking:

If:

- **e0** has some type **(t1 * … * tn) -> t**
- **e1** has type **t1**, …, **en** has type **tn**

Then:

- **e0(e1,…,en)** has type **t**

Example: **pow(x,y-1)** in previous example has type **int**

Function-calls continued

e0(e1,…,en)

Evaluation:

1. (Under current dynamic environment,) evaluate **e0** to a function **fun *x0* (*x1* : *t1*, … , *xn* : *tn*) = *e***
   - Since call type-checked, result *will be* a function
2. (Under current dynamic environment,) evaluate arguments to values **v1, …, vn**
3. Result is evaluation of ***e*** in an environment extended to map **x1** to **v1**, …, **xn** to **vn**
   - (“An environment” is actually the environment where the function was defined, and includes **x0** for recursion)

Tuples and lists

So far: numbers, booleans, conditionals, variables, functions

- Now ways to build up data with multiple parts
- This is essential
- Java examples: classes with fields, arrays

Now:

- *Tuples*: fixed “number of pieces” that may have different types

Then:

- *Lists*: any “number of pieces” that all have the same type

Later:

- Other more general ways to create compound data

Pairs (2-tuples)

Need a way to *build* pairs and a way to *access* the pieces

Build:

- Syntax: **`(e1,e2)`**
- Evaluation: Evaluate **`e1`** to **`v1`** and **`e2`** to **`v2`**; result is **`(v1,v2)`**
  - A pair of values is a value
- Type-checking: If **`e1`** has type **`ta`** and **`e2`** has type **`tb`**, then the pair expression has type **`ta * tb`**
  - A new kind of type

Pairs (2-tuples)

Need a way to *build* pairs and a way to *access* the pieces

Access:

- Syntax: `#1 e` and `#2 e`
- Evaluation: Evaluate **`e`** to a pair of values and return first or second piece
  - Example: If ***`e`*** is a variable **`x`**, then look up **`x`** in environment
- Type-checking: If ***`e`*** has type **`ta * tb`**, then **`#1 e`** has type **`ta`** and **`#2 e`** has type **`tb`**

Examples

Functions can take and return pairs

```
fun swap (pr : int*bool) =
  (#2 pr, #1 pr)

fun sum_two_pairs (pr1 : int*int, pr2 : int*int) =
  (#1 pr1) + (#2 pr1) + (#1 pr2) + (#2 pr2)

fun div_mod (x : int, y : int) =
  (x div y, x mod y)

fun sort_pair (pr : int*int) =
  if (#1 pr) < (#2 pr)
  then pr
  else (#2 pr, #1 pr)
```

Tuples

Actually, you can have *tuples* with more than two parts

- A new feature: a generalization of pairs

- `(e1,e2,…,en)`
- `ta * tb * … * tn`
- `#1 e, #2 e, #3 e, …`

Homework 1 uses triples of type `int*int*int` a lot

Nesting

Pairs and tuples can be nested however you want

- Not a new feature: implied by the syntax and semantics

```
val x1 = (7,(true,9)) (* int * (bool*int) *)

val x2 = #1 (#2 x1)   (* bool *)

val x3 = (#2 x1)      (* bool*int *)

val x4 = ((3,5),((4,8),(0,0)))
         (* (int*int)*((int*int)*(int*int)) *)
```

Lists

- Despite nested tuples, the type of a variable still "commits" to a particular "amount" of data

In contrast, a list:

  – Can have any number of elements
  – But all list elements have the same type

Need ways to *build* lists and *access* the pieces…

Building Lists

- The empty list is a value:

```
[]
```

- In general, a list of values is a value; elements separated by commas:

```
[v1,v2,…,vn]
```

- If **`e1`** evaluates to **`v`** and **`e2`** evaluates to a list **`[v1,…,vn]`**, then **`e1::e2`** evaluates to **`[v,…,vn]`**

```
e1::e2 (* pronounced "cons" *)
```

Accessing Lists

Until we learn pattern-matching, we will use three standard-library functions

- **`null e`** evaluates to **`true`** if and only if **`e`** evaluates to **`[]`**

- If **`e`** evaluates to **`[v1,v2,…,vn]`** then **`hd e`** evaluates to **`v1`**
  - (raise exception if **`e`** evaluates to **`[]`**)

- If **`e`** evaluates to **`[v1,v2,…,vn]`** then **`tl e`** evaluates to **`[v2,…,vn]`**
  - (raise exception if **`e`** evaluates to **`[]`**)
  - Notice result is a list

Type-checking list operations

Lots of new types: For any type **`t`**, the type **`t list`** describes lists where all elements have type **`t`**

- Examples: **`int list`** **`bool list`** **`int list list`** **`(int * int) list`** **`(int list * int) list`**

- So **`[]`** can have type **`t list`** list for *any* type
  - SML uses type **`'a list`** to indicate this ("quote a" or "alpha")
- For **`e1::e2`** to type-check, we need a **`t`** such that **`e1`** has type **`t`** and **`e2`** has type **`t list`**. Then the result type is **`t list`**
- **`null : 'a list -> bool`**
- **`hd : 'a list -> 'a`**
- **`tl : 'a list -> 'a list`**

Example list functions

```
fun sum_list (xs : int list) =
  if null xs
  then 0
  else hd(xs) + sum_list(tl(xs))

fun countdown (x : int) =
  if x=0
  then []
  else x :: countdown (x-1)

fun append (xs : int list, ys : int list) =
  if null xs
  then ys
  else hd (xs) :: append (tl(xs), ys)
```

Recursion again

Functions over lists are usually recursive

- Only way to “get to all the elements”

- What should the answer be for the empty list?
- What should the answer be for a non-empty list?
  - Typically in terms of the answer for the tail of the list!

Similarly, functions that produce lists of potentially any size will be recursive

- You create a list out of smaller lists

Lists of pairs

Processing lists of pairs requires no new features. Examples:

```
fun sum_pair_list (xs : (int*int) list) =
  if null xs
  then 0
  else #1(hd xs) + #2(hd xs) + sum_pair_list(tl xs)

fun firsts (xs : (int*int) list) =
  if null xs
  then []
  else #1(hd xs) :: firsts(tl xs)

fun seconds (xs : (int*int) list) =
  if null xs
  then []
  else #2(hd xs) :: seconds(tl xs)

fun sum_pair_list2 (xs : (int*int) list) =
 (sum_list (firsts xs)) + (sum_list (seconds xs))
```

PAUL G. ALLEN SCHOOL
OF COMPUTER SCIENCE & ENGINEERING

CSE341: Programming Languages

Lecture 3
Local Bindings;
Options;
Benefits of No Mutation

Dan Grossman

Spring 2019

Review

Huge progress already on the core pieces of ML:

- Types: **`int bool unit  t1*…*tn  t list  t1*…*tn->t`**
  - Types "nest" (each **`t`** above can be itself a compound type)
- Variables, environments, and basic expressions
- Functions
  - Build: **`fun x0 (x1:t1, …, xn:tn) = e`**
  - Use: **`e0 (e1, …, en)`**
- Tuples
  - Build: **`(e1, …, en)`**
  - Use: **`#1 e, #2 e, …`**
- Lists
  - Build: **`[]  e1::e2`**
  - Use: **`null e  hd e  tl e`**

Today

- The big thing we need: local bindings
  - For style and convenience
  - A big but natural idea: nested function bindings
  - For efficiency (***not*** "just a little faster")

- One last feature for Problem 11 of Homework 1: options

- Why not having mutation (assignment statements) is a valuable language feature
  - No need for you to keep track of sharing/aliasing, which Java programmers must obsess about

Let-expressions

3 questions:

- Syntax: `let b1 b2 … bn in e end`
  - Each ***bi*** is any *binding* and ***e*** is any *expression*

- Type-checking: Type-check each ***bi*** and ***e*** in a static environment that includes the previous bindings.
  Type of whole let-expression is the type of ***e***.

- Evaluation: Evaluate each ***bi*** and ***e*** in a dynamic environment that includes the previous bindings.
  Result of whole let-expression is result of evaluating ***e***.

It is an expression

A let-expression is ***just an expression***, so we can use it ***anywhere*** an expression can go

Silly examples

```
fun silly1 (z : int) =
    let val x = if z > 0 then z else 34
        val y = x+z+9
    in
        if x > y then x*2 else y*y
    end
fun silly2 () =
    let val x = 1
    in
        (let val x = 2 in x+1 end) +
        (let val y = x+2 in y+1 end)
    end
```

`silly2` is poor style but shows let-expressions are expressions

- Can also use them in function-call arguments, if branches, etc.
- Also notice shadowing

What's new

- What's new is ***scope***: where a binding is in the environment
  - *In* later bindings and body of the let-expression
    - (Unless a later or nested binding shadows it)
  - *Only in* later bindings and body of the let-expression

- *Nothing else is new:*
  - Can put any binding we want, even function bindings
  - Type-check and evaluate just like at "top-level"

Any binding

According to our rules for let-expressions, we can define functions inside any let-expression

```
let  b1 b2 … bn  in  e  end
```

This is a natural idea, and often good style

(Inferior) Example

```
fun countup_from1 (x : int) =
    let fun count (from : int, to : int) =
            if from = to
            then to :: []
            else from :: count(from+1,to)
    in
        count (1,x)
    end
```

- This shows how to use a local function binding, but:
  - Better version on next slide
  - **`count`** might be useful elsewhere

Better:

```
fun countup_from1_better (x : int) =
    let fun count (from : int) =
            if from = x
            then x :: []
            else from :: count(from+1)
    in
        count 1
    end
```

- Functions can use bindings in the environment where they are defined:
  - Bindings from “outer” environments
    - Such as parameters to the outer function
  - Earlier bindings in the let-expression
- Unnecessary parameters are usually bad style
  - Like **`to`** in previous example

Nested functions: style

- Good style to define helper functions inside the functions they help if they are:
  - Unlikely to be useful elsewhere
  - Likely to be misused if available elsewhere
  - Likely to be changed or removed later

- A fundamental trade-off in code design: reusing code saves effort and avoids bugs, but makes the reused code harder to change later

Avoid repeated recursion

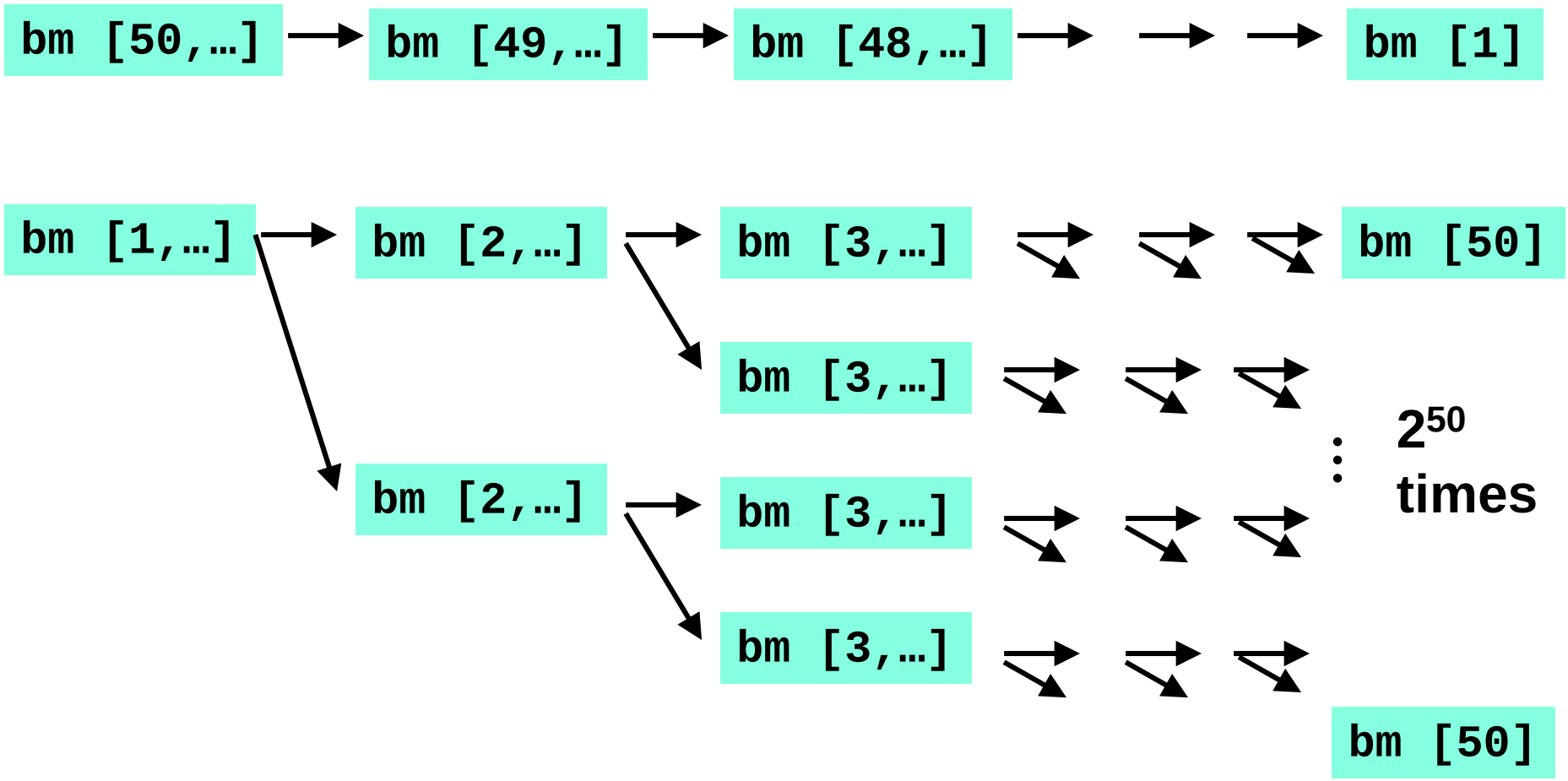
Consider this code and the recursive calls it makes

- Don't worry about calls to **`null`**, **`hd`**, and **`tl`** because they do a small constant amount of work

```
fun bad_max (xs : int list) =
    if null xs
    then 0 (* horrible style; fix later *)
    else if null (tl xs)
    then hd xs
    else if hd xs > bad_max (tl xs)
    then hd xs
    else bad_max (tl xs)

let x = bad_max [50,49,…,1]
let y = bad_max [1,2,…,50]
```

Fast vs. unusable

```
if hd xs > bad_max (tl xs)
then hd xs
else bad_max (tl xs)
```



Math never lies

Suppose one **`bad_max`** call's if-then-else logic and calls to **`hd`**, **`null`**, **`tl`** take 10^{-7} seconds

- Then **`bad_max [50,49,…,1]`** takes 50 x 10^{-7} seconds
- And **`bad_max [1,2,…,50]`** takes 1.12 x 10^{8} seconds
  - (over 3.5 years)
  - **`bad_max [1,2,…,55]`** takes over 1 century
  - Buying a faster computer won't help much ☺

The key is not to do repeated work that might do repeated work that might do…

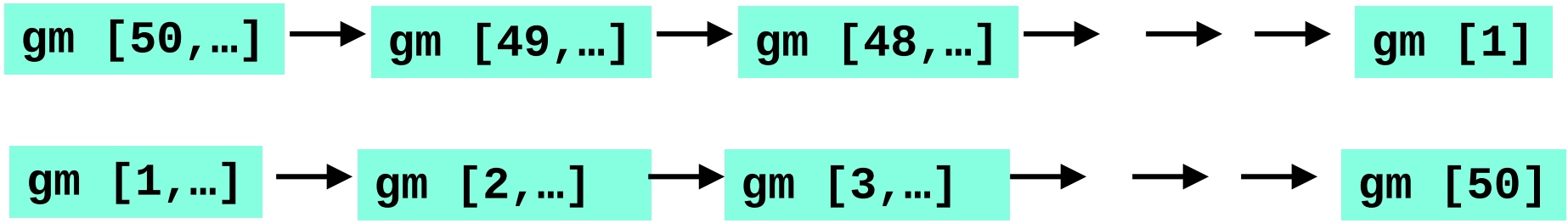
- Saving recursive results in local bindings is essential…

Efficient max

```
fun good_max (xs : int list) =
    if null xs
    then 0 (* horrible style; fix later *)
    else if null (tl xs)
    then hd xs
    else
         let val tl_ans = good_max(tl xs)
         in
             if hd xs > tl_ans
             then hd xs
             else tl_ans
         end
```

Fast vs. fast

```
let val tl_ans = good_max(tl xs)
in
    if hd xs > tl_ans
    then hd xs
    else tl_ans
end
```



Options

- **`t option`** is a type for any type **`t`**
  - (much like **`t list`**, but a different type, not a list)

Building:

- **`NONE`** has type **`'a option`** (much like **`[]`** has type **`'a list`**)
- **`SOME e`** has type **`t option`** if **`e`** has type **`t`** (much like **`e::[]`**)

Accessing:

- **`isSome`** has type **`'a option -> bool`**
- **`valOf`** has type **`'a option -> 'a`** (exception if given **`NONE`**)

Example

```
fun better_max (xs : int list) =
    if null xs
    then NONE
    else
        let val tl_ans = better_max(tl xs)
        in
            if isSome tl_ans
               andalso valOf tl_ans > hd xs
            then tl_ans
            else SOME (hd xs)
        end
```

```
val better_max = fn : int list -> int option
```

- Nothing wrong with this, but as a matter of style might prefer not to do so much useless "**`valOf`**" in the recursion

Example variation

```
fun better_max2 (xs : int list) =
    if null xs
    then NONE
    else let (* ok to assume xs nonempty b/c local *)
             fun max_nonempty (xs : int list) =
               if null (tl xs)
               then hd xs
               else
                  let val tl_ans = max_nonempty(tl xs)
                  in
                    if hd xs > tl_ans
                    then hd xs
                    else tl_ans
                  end
         in
            SOME (max_nonempty xs)
         end
```

Cannot tell if you copy

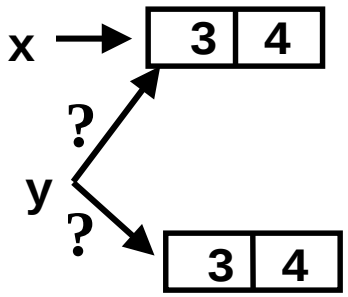
```
fun sort_pair (pr : int * int) =
  if #1 pr < #2 pr
  then pr
  else (#2 pr, #1 pr)

fun sort_pair (pr : int * int) =
  if #1 pr < #2 pr
  then (#1 pr, #2 pr)
  else (#2 pr, #1 pr)
```

In ML, these two implementations of **`sort_pair`** are indistinguishable

- But only because tuples are immutable
- The first is better style: simpler and avoids making a new pair in the then-branch
- In languages with mutable compound data, these are different!

Suppose we had mutation…

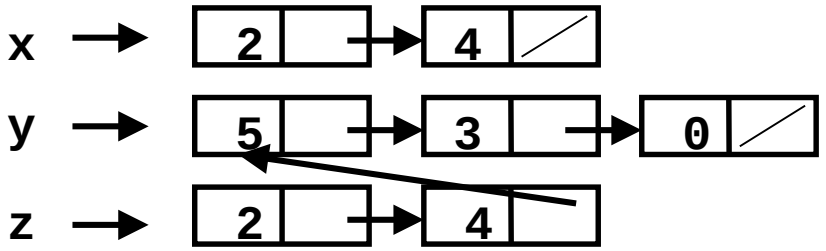
```
val x = (3,4)
val y = sort_pair x

somehow mutate #1 x to hold 5

val z = #1 y
```



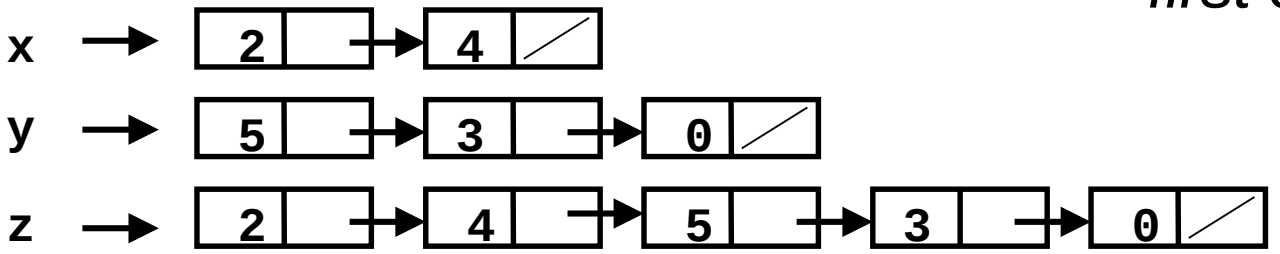
- What is **`z`**?
  - Would depend on how we implemented **`sort_pair`**
    - Would have to decide carefully and document **`sort_pair`**
  - But without mutation, we can implement "either way"
    - No code can ever distinguish aliasing vs. identical copies
    - No need to think about aliasing: focus on other things
    - Can use aliasing, which saves space, without danger

An even better example

```
fun append (xs : int list, ys : int list) =
    if null xs
    then ys
    else hd (xs) :: append (tl(xs), ys)
val x = [2,4]
val y = [5,3,0]
val z = append(x,y)
```



or



(can't tell, but it's the first one)

ML vs. Imperative Languages

- 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 languages with mutable data (e.g., 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
    - Consider a Java example…

Java security nightmare (bad code)

```
class ProtectedResource {
   private Resource theResource = ...;
   private String[] allowedUsers = ...;
   public String[] getAllowedUsers() {
      return allowedUsers;
   }
   public String currentUser() { ... }
   public void useTheResource() {
      for(int i=0; i < allowedUsers.length; i++) {
         if(currentUser().equals(allowedUsers[i])) {
            ... // access allowed: use it
            return;
         }
      }
      throw new IllegalAccessException();
   }
}
```

Have to make copies

The problem:

```
p.getAllowedUsers()[0] = p.currentUser();
p.useTheResource();
```

The fix:

```
public String[] getAllowedUsers() {
   … return a copy of allowedUsers …
}
```

Reference (alias) vs. copy doesn't matter if code is immutable!

PAUL G. ALLEN SCHOOL
OF COMPUTER SCIENCE & ENGINEERING

CSE341: Programming Languages

Lecture 4
Records, Datatypes, Case Expressions

Dan Grossman

Spring 2019

Five different things

1. Syntax: How do you write language constructs?
2. Semantics: What do programs mean? (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, code formatter, …)
   - Not actually part of the language

These are 5 separate issues

- In practice, all are essential for good programmers
- Many people confuse them, but shouldn't

Our Focus

This course focuses on semantics and idioms

- Syntax is usually uninteresting
  - A fact to learn, like “The American Civil War ended in 1865”
  - People obsess over subjective preferences

- Libraries and tools crucial, but often learn new ones “on the job”
  - We are learning semantics and how to use that knowledge to understand all software and employ appropriate idioms
  - By avoiding most libraries/tools, our languages may look “silly” but so would *any* language used this way

How to build bigger types

- Already know:
  - Have various *base types* like **`int bool unit char`**
  - Ways to build (nested) *compound types*: tuples, lists, options
- Coming soon: more ways to build compound types
- First: 3 most important type building blocks in *any* language
  - "Each of": A **`t`** value contains *values of each of* **`t1 t2 … tn`**
  - "One of": A **`t`** value contains *values of one of* **`t1 t2 … tn`**
  - "Self reference": A **`t`** value can refer to other **`t`** values

  Remarkable: A lot of data can be described with just these building blocks

  Note: These are not the common names for these concepts

Examples

- Tuples build each-of types
  - **`int * bool`** contains an **`int`** *and* a **`bool`**
- Options build one-of types
  - **`int option`** contains an **`int`** *or* it contains no data
- Lists use all three building blocks
  - **`int list`** contains an **`int`** *and* another **`int list`** *or* it contains no data
- And of course we can nest compound types
  - **`((int * int) option * (int list list)) option`**

Rest of this Lecture

- Another way to build each-of types in ML
  - *Records*: have named *fields*
  - Connection to tuples and idea of *syntactic sugar*

- A way to build and use our own one-of types in ML
  - For example, a type that contains an **`int`** or a **`string`**
  - Will lead to *pattern-matching*, one of ML's coolest and strangest-to-Java-programmers features

- Later in course: How OOP does one-of types
  - Key contrast with procedural and functional programming

Records

Record values have fields (any name) holding values

```
{f1 = v1, …, fn = vn}
```

Record types have fields (and name) holding types

```
{f1 : t1, …, fn : tn}
```

The order of fields in a record value or type never matters

- REPL alphabetizes fields just for consistency

Building records:

```
{f1 = e1, …, fn = en}
```

Accessing components:

```
#myfieldname e
```

(Evaluation rules and type-checking as expected)

Example

```
{name = "Matai", id = 4 - 3}
```

Evaluates to

```
{id = 1, name = "Matai"}
```

And has type

```
{id : int, name : string}
```

If some expression such as a variable **x** has this type, then get fields with: `#id x` `#name x`

Note we did not have to declare any record types

- The same program could also make a `{id=true,ego=false}` of type `{id:bool,ego:bool}`

By name vs. by position

- Little difference between **`(4,7,9)`** and **`{f=4,g=7,h=9}`**
  - Tuples a little shorter
  - Records a little easier to remember "what is where"
  - Generally a matter of taste, but for many (6? 8? 12?) fields, a record is usually a better choice

- A common decision for a construct's syntax is whether to refer to things *by position* (as in tuples) or *by some (field) name* (as with records)
  - A common hybrid is like with Java method arguments (and ML functions as used so far):
    - Caller uses *position*
    - Callee uses *variables*
    - Could totally do it differently; some languages have

The truth about tuples

Previous lecture gave tuples syntax, type-checking rules, and evaluation rules

But we could have done this instead:

- Tuple syntax is just a different way to write certain records
- **`(e1,…,en)`** is another way of writing **`{1=e1,…,n=en}`**
- **`t1*…*tn`** is another way of writing **`{1:t1,…,n:tn}`**
- In other words, records with field names 1, 2, …

In fact, this is how ML actually defines tuples

- Other than special syntax in programs and printing, they don't exist
- You really can write **`{1=4,2=7,3=9}`**, but it's bad style

Syntactic sugar

"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 ☺

Will see many more examples of syntactic sugar

- They simplify *understanding* the language
- They simplify *implementing* the language

Why? Because there are fewer semantics to worry about even though we have the syntactic convenience of tuples

Another example we saw: **`andalso`** and **`orelse`** vs. **`if then else`**

Datatype bindings

A “strange” (?) and totally awesome (!) way to make one-of types:

- A **`datatype`** binding

```
datatype mytype = TwoInts of int * int
                | Str of string
                | Pizza
```

- Adds a new type **`mytype`** to the environment
- Adds *constructors* to the environment: **`TwoInts`**, **`Str`**, and **`Pizza`**
- A constructor is (among other things), a function that makes values of the new type (or is a value of the new type):
  - **`TwoInts : int * int -> mytype`**
  - **`Str : string -> mytype`**
  - **`Pizza : mytype`**

The values we make

```
datatype mytype = TwoInts of int * int
                | Str of string
                | Pizza
```

- Any value of type **`mytype`** is made from *one of* the constructors
- The value contains:
  - A "tag" for "which constructor" (e.g., **`TwoInts`**)
  - The corresponding data (e.g., **`(7,9)`**)
- Examples:
  - **`TwoInts(3+4,5+4)`** evaluates to **`TwoInts(7,9)`**
  - **`Str(if true then "hi" else "bye")`** evaluates to **`Str("hi")`**
  - **`Pizza`** is a value

Using them

So we know how to *build* datatype values; need to *access* them

There are *two* aspects to accessing a datatype value

1. Check what *variant* it is (what constructor made it)
2. Extract the *data* (if that variant has any)

Notice how our other one-of types used functions for this:

- **`null`** and **`isSome`** check variants
- **`hd`**, **`tl`**, and **`valOf`** extract data (raise exception on wrong variant)

ML *could* have done the same for datatype bindings

- For example, functions like “isStr” and “getStrData”
- Instead it did something better

Case

ML combines the two aspects of accessing a one-of value with a *case expression* and *pattern-matching*

- Pattern-matching much more general/powerful (Lecture 5)

Example:

```
fun f x = (* f has type mytype -> int *)
    case x of
        Pizza => 3
      | TwoInts(i1,i2) => i1+i2
      | Str s => String.size s
```

- A multi-branch conditional to pick branch based on variant
- Extracts data and binds to variables local to that branch
- Type-checking: all branches must have same type
- Evaluation: evaluate between **`case`** … **`of`** and the right branch

Patterns

In general the syntax is:

```
case e0 of
     p1 => e1
   | p2 => e2
     …
   | pn => en
```

For today, each *pattern* is a constructor name followed by the right number of variables (i.e., **`C`** or **`C x`** or **`C(x,y)`** or …)

- Syntactically most patterns (all today) look like expressions
- But patterns are not expressions
  - We do not evaluate them
  - We see if the result of **`e0`** *matches* them

Why this way is better

0. You can use pattern-matching to write your own testing and data-extractions functions if you must
   - But do not do that on your homework

1. You cannot forget a case (inexhaustive pattern-match warning)
2. You cannot duplicate a case (a type-checking error)
3. You will not forget to test the variant correctly and get an exception (like **hd [ ]**)
4. Pattern-matching can be generalized and made more powerful, leading to elegant and concise code

PAUL G. ALLEN SCHOOL
OF COMPUTER SCIENCE & ENGINEERING

CSE341: Programming Languages

Lecture 5
More Datatypes and Pattern-Matching

Dan Grossman

Spring 2019

Useful examples

Let's fix the fact that our only example datatype so far was silly…

- Enumerations, including carrying other data

```
datatype suit = Club | Diamond | Heart | Spade
datatype card_value = Jack | Queen | King
                    | Ace | Num of int
```

- Alternate ways of identifying real-world things/people

```
datatype id = StudentNum of int
            | Name of string
                      * (string option)
                      * string
```

Don't do this

Unfortunately, bad training and languages that make one-of types inconvenient lead to common *bad style* where each-of types are used where one-of types are the right tool

```
(* use the studen_num and ignore other
   fields unless the student_num is ~1 *)
{ student_num : int,
  first       : string,
  middle      : string option,
  last        : string }
```

- Approach gives up all the benefits of the language enforcing every value is one variant, you don't forget branches, etc.
- And makes it less clear what you are doing

That said…

But if instead the point is that every “person” in your program has a name and maybe a student number, then each-of is the way to go:

```
{ student_num : int option,
  first       : string,
  middle      : string option,
  last        : string }
```

Expression Trees

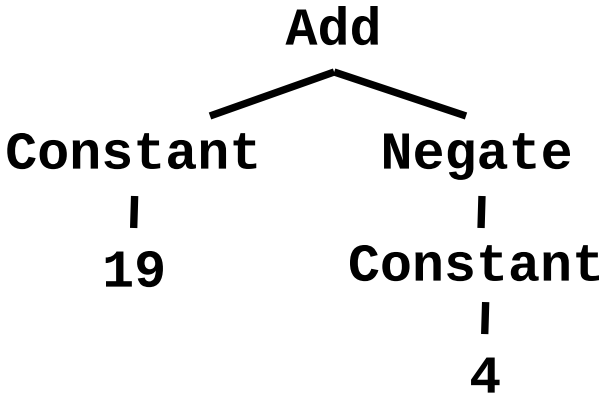
A more exciting (?) example of a datatype, using self-reference

```
datatype exp = Constant of int
             | Negate   of exp
             | Add      of exp * exp
             | Multiply of exp * exp
```

An expression in ML of type **`exp`**:

```
Add (Constant (10+9), Negate (Constant 4))
```

How to picture the resulting value in your head:



Recursion

Not surprising:

Functions over recursive datatypes are usually recursive

```
fun eval e =
    case e of
         Constant i       => i
       | Negate e2        => ~ (eval e2)
       | Add(e1,e2)       => (eval e1) + (eval e2)
       | Multiply(e1,e2) => (eval e1) * (eval e2)
```

Putting it together

```
datatype exp = Constant of int
             | Negate   of exp
             | Add      of exp * exp
             | Multiply of exp * exp
```

Let's define **`max_constant : exp -> int`**

Good example of combining several topics as we program:

- Case expressions
- Local helper functions
- Avoiding repeated recursion
- Simpler solution by using library functions

See the **`.sml`** file…

Careful definitions

When a language construct is “new and strange,” there is *more* reason to define the evaluation rules precisely…

… so let’s review datatype bindings and case expressions “so far”

- *Extensions* to come but won’t invalidate the “so far”

Datatype bindings

```
datatype t = C1 of t1 | C2 of t2 | … | Cn of tn
```

Adds type **`t`** and constructors **`Ci`** of type **`ti->t`**

- **`Ci v`** is a value, i.e., the result "includes the tag"

Omit "**`of t`**" for constructors that are just tags, no underlying data

- Such a **`Ci`** is a value of type **`t`**

Given an expression of type **`t`**, use *case expressions* to:

- See which variant (tag) it has
- Extract underlying data once you know which variant

Datatype bindings

```
case e of p1 => e1 | p2 => e2 | … | pn => en
```

- As usual, can use a case expressions anywhere an expression goes
  - Does not need to be whole function body, but often is
- Evaluate **`e`** to a value, call it **`v`**
- If **`pi`** is the first *pattern* to *match* **`v`**, then result is evaluation of **`ei`** in environment "extended by the match"
- Pattern **`Ci(x1,…,xn)`** matches value **`Ci(v1,…,vn)`** and extends the environment with **`x1`** to **`v1`** … **`xn`** to **`vn`**
  - For "no data" constructors, pattern **`Ci`** matches value **`Ci`**

Recursive datatypes

Datatype bindings can describe recursive structures

- Have seen arithmetic expressions
- Now, linked lists:

```
datatype my_int_list = Empty
                     | Cons of int * my_int_list

val x = Cons(4,Cons(23,Cons(2008,Empty)))

fun append_my_list (xs,ys) =
   case xs of
       Empty => ys
     | Cons(x,xs') => Cons(x, append_my_list(xs',ys))
```

Options are datatypes

Options are just a predefined datatype binding

- **`NONE`** and **`SOME`** are *constructors*, not just functions
- So use pattern-matching not **`isSome`** and **`valOf`**

```
fun inc_or_zero intoption =
    case intoption of
        NONE => 0
      | SOME i => i+1
```

Lists are datatypes

Do not use **`hd`**, **`tl`**, or **`null`** either

- **`[]`** and **`::`** are constructors too
- (strange syntax, particularly *infix*)

```
fun sum_list xs =
   case xs of
        [] => 0
      | x::xs' => x + sum_list xs'

fun append (xs,ys) =
   case xs of
        [] => ys
      | x::xs' => x :: append (xs',ys)
```

Why pattern-matching

- Pattern-matching is better for options and lists for the same reasons as for all datatypes
  - No missing cases, no exceptions for wrong variant, etc.

- We just learned the other way first for pedagogy
  - Do not use **`isSome`**, **`valOf`**, **`null`**, **`hd`**, **`tl`** on Homework 2

- So why are **`null`**, **`tl`**, etc. predefined?
  - For passing as arguments to other functions (next week)
  - Because sometimes they are convenient
  - But not a big deal: could define them yourself

Excitement ahead…

Learn some deep truths about “what is really going on”

- Using much more syntactic sugar than we realized

- Every val-binding and function-binding uses pattern-matching

- Every function in ML takes exactly one argument

First need to extend our definition of pattern-matching…

Each-of types

So far have used pattern-matching for one of types because we *needed* a way to access the values

Pattern matching also works for records and tuples:

- The pattern **`(x1,…,xn)`**
  matches the tuple value **`(v1,…,vn)`**
- The pattern **`{f1=x1, …, fn=xn}`**
  matches the record value **`{f1=v1, …, fn=vn}`**
  (and fields can be reordered)

Example

This is poor style, but based on what I told you so far, the only way to use patterns

- Works but poor style to have one-branch cases

```
fun sum_triple triple =
   case triple of
     (x, y, z) => x + y + z

fun full_name r =
   case r of
     {first=x, middle=y, last=z} =>
         x ^ " " ^ y ^ " " ^ z
```

Val-binding patterns

- New feature: A val-binding can use a pattern, not just a variable
  - (Turns out variables are just one kind of pattern, so we just told you a half-truth in Lecture 1)

```
val p = e
```

- Great for getting (all) pieces out of an each-of type
  - Can also get only parts out (not shown here)

- Usually poor style to put a constructor pattern in a val-binding
  - Tests for the one variant and raises an exception if a different one is there (like **`hd`**, **`tl`**, and **`valOf`**)

Better example

This is okay style

- Though we will improve it again next
- Semantically identical to one-branch case expressions

```
fun sum_triple triple =
    let val (x, y, z) = triple
    in
        x + y + z
    end

fun full_name r =
    let val {first=x, middle=y, last=z} = r
    in
        x ^ " " ^ y ^ " " ^ z
    end
```

Function-argument patterns

A function argument can also be a pattern

- Match against the argument in a function call

```
fun f p = e
```

Examples (great style!):

```
fun sum_triple (x, y, z) =
    x + y + z

fun full_name {first=x, middle=y, last=z} =
    x ^ " " ^ y ^ " " ^ z
```

A new way to go

- For Homework 2:
  - Do not use the # character
  - Do not need to write down any explicit types

Hmm

A function that takes one triple of type **`int*int*int`** and returns an **`int`** that is their sum:

```
fun sum_triple (x, y, z) =
    x + y + z
```

A function that takes three **`int`** arguments and returns an **`int`** that is their sum

```
fun sum_triple (x, y, z) =
    x + y + z
```

See the difference? (Me neither.) ☺

The truth about functions

- In ML, every function takes exactly one argument (*)
- What we call multi-argument functions are just functions taking one tuple argument, implemented with a tuple pattern in the function binding
  - Elegant and flexible language design
- Enables cute and useful things you cannot do in Java, e.g.,

```
fun rotate_left (x, y, z) = (y, z, x)
fun rotate_right t = rotate_left (rotate_left t)
```

* “Zero arguments” is the unit pattern **( )** matching the unit value **( )**

PAUL G. ALLEN SCHOOL
OF COMPUTER SCIENCE & ENGINEERING

CSE341: Programming Languages

Lecture 6
Nested Patterns
Exceptions
Tail Recursion

Dan Grossman

Spring 2019

Nested patterns

- We can nest patterns as deep as we want
  - Just like we can nest expressions as deep as we want
  - Often avoids hard-to-read, wordy nested case expressions

- So the full meaning of pattern-matching is to compare a pattern against a value for the “same shape” and bind variables to the “right parts”
  - More precise recursive definition coming after examples

Useful example: zip/unzip 3 lists

```
fun zip3 lists =
    case lists of
         ([],[],[]) => []
       | (hd1::tl1,hd2::tl2,hd3::tl3) =>
              (hd1,hd2,hd3)::zip3(tl1,tl2,tl3)
       | _ => raise ListLengthMismatch

fun unzip3 triples =
    case triples of
         [] => ([],[],[])
       | (a,b,c)::tl =>
           let val (l1, l2, l3) = unzip3 tl
           in
               (a::l1,b::l2,c::l3)
           end
```

More examples in **`.sml`** files

Style

- Nested patterns can lead to very elegant, concise code
  - Avoid nested case expressions if nested patterns are simpler and avoid unnecessary branches or let-expressions
    - Example: **`unzip3`** and **`nondecreasing`**
  - A common idiom is matching against a tuple of datatypes to compare them
    - Examples: **`zip3`** and **`multsign`**

- Wildcards are good style: use them instead of variables when you do not need the data
  - Examples: **`len`** and **`multsign`**

(Most of) the full definition

The semantics for pattern-matching takes a pattern *p* and a value *v* and decides (1) does it match and (2) if so, what variable bindings are introduced.

Since patterns can nest, the definition is elegantly recursive, with a separate rule for each kind of pattern. Some of the rules:

- If *p* is a variable *x*, the match succeeds and *x* is bound to *v*
- If *p* is _, the match succeeds and no bindings are introduced
- If *p* is *(p1,…,pn)* and *v* is *(v1,…,vn)*, the match succeeds if and only if *p1* matches *v1*, …, *pn* matches *vn*. The bindings are the union of all bindings from the submatches
- If *p* is *C p1*, the match succeeds if *v* is *C v1* (i.e., the same constructor) and *p1* matches *v1*. The bindings are the bindings from the submatch.
- … (there are several other similar forms of patterns)

Examples

- Pattern **`a::b::c::d`** matches all lists with >= 3 elements
- Pattern **`a::b::c::[]`** matches all lists with 3 elements
- Pattern **`((a,b),(c,d))::e`** matches all non-empty lists of pairs of pairs

Exceptions

An exception binding introduces a new kind of exception

```
exception MyUndesirableCondition
exception MyOtherException of int * int
```

The **`raise`** primitive raises (a.k.a. throws) an exception

```
raise MyUndesirableException
raise (MyOtherException (7,9))
```

A handle expression can handle (a.k.a. catch) an exception

- If doesn't match, exception continues to propagate

```
e1 handle MyUndesirableException => e2
e1 handle MyOtherException(x,y) => e2
```

Actually…

Exceptions are a lot like datatype constructors…

- Declaring an exception adds a constructor for type **`exn`**
- Can pass values of **`exn`** anywhere (e.g., function arguments)
  - Not too common to do this but can be useful
- **`handle`** can have multiple branches with patterns for type **`exn`**

Recursion

Should now be comfortable with recursion:

- No harder than using a loop (whatever that is ☺)
- Often much easier than a loop
  - When processing a tree (e.g., evaluate an arithmetic expression)
  - Examples like appending lists
  - Avoids mutation even for local variables
- Now:
  - How to reason about *efficiency* of recursion
  - The importance of *tail recursion*
  - Using an *accumulator* to achieve tail recursion
  - [No new language features here]

Call-stacks

While a program runs, there is a *call stack* of function calls that have started but not yet returned

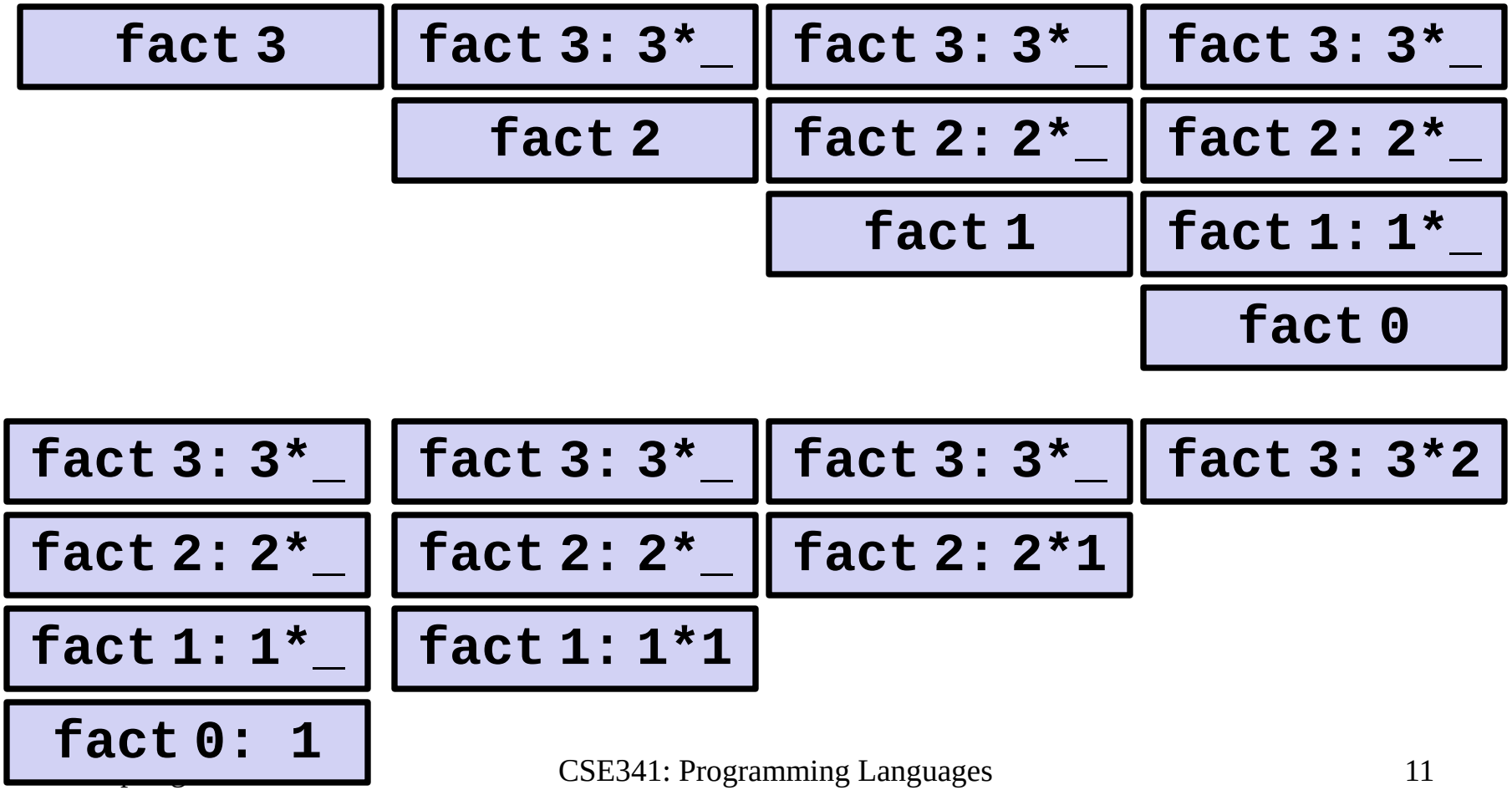
- Calling a function **f** pushes an instance of **f** on the stack
- When a call to **f** finishes, it is popped from the stack

These stack-frames store information like the value of local variables and "what is left to do" in the function

Due to recursion, multiple stack-frames may be calls to the same function

Example

```
fun fact n = if n=0 then 1 else n*fact(n-1)

val x = fact 3
```

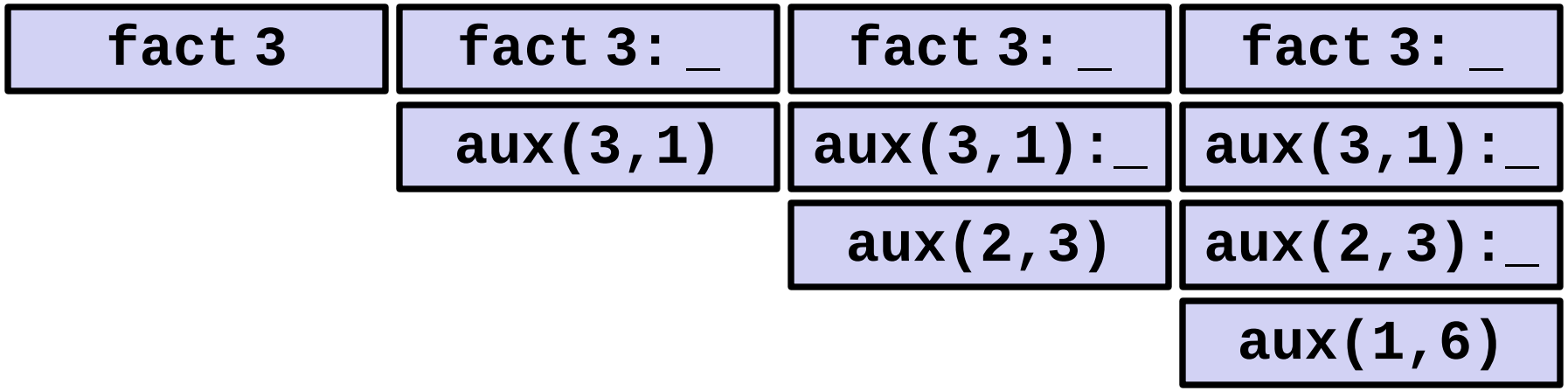


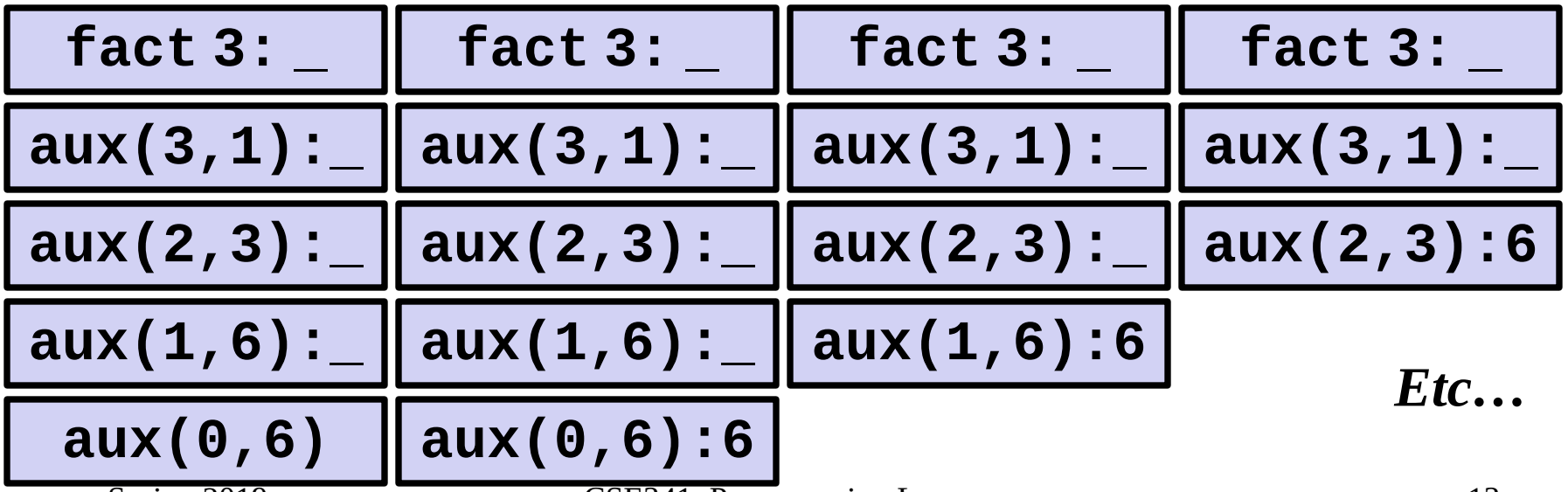
Example Revised

```
fun fact n =
    let fun aux(n,acc) =
            if n=0
            then acc
            else aux(n-1,acc*n)
    in
       aux(n,1)
    end

val x = fact 3
```

Still recursive, more complicated, but the result of recursive calls *is* the result for the caller (no remaining multiplication)

The call-stacks





An optimization

It is unnecessary to keep around a stack-frame just so it can get a callee's result and return it without any further evaluation

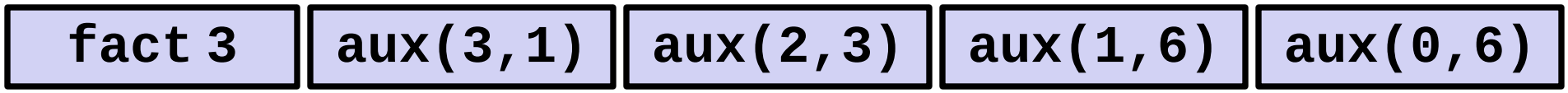
ML recognizes these *tail calls* in the compiler and treats them differently:

- Pop the caller *before* the call, allowing callee to *reuse* the same stack space
- (Along with other optimizations,) as efficient as a loop

Reasonable to assume all functional-language implementations do tail-call optimization

What really happens

```
fun fact n =
    let fun aux(n,acc) =
            if n=0
            then acc
            else aux(n-1,acc*n)
    in
       aux(n,1)
    end

val x = fact 3
```



Moral of tail recursion

- Where reasonably elegant, feasible, and important, rewriting functions to be *tail-recursive* can be much more efficient
  - Tail-recursive: recursive calls are tail-calls

- There is a methodology that can often guide this transformation:
  - Create a helper function that takes an *accumulator*
  - Old base case becomes initial accumulator
  - New base case becomes final accumulator

Methodology already seen

```
fun fact n =
    let fun aux(n,acc) =
            if n=0
            then acc
            else aux(n-1,acc*n)
    in
       aux(n,1)
    end

val x = fact 3
```

fact 3	aux(3,1)	aux(2,3)	aux(1,6)	aux(0,6)

Another example

```
fun sum xs =
   case xs of
       [] => 0
     | x::xs' => x + sum xs'
```

```
fun sum xs =
    let fun aux(xs,acc) =
            case xs of
               [] => acc
             | x::xs' => aux(xs',x+acc)
    in
       aux(xs,0)
    end
```

And another

```
fun rev xs =
   case xs of
       [] => []
     | x::xs' => (rev xs') @ [x]
```

```
fun rev xs =
    let fun aux(xs,acc) =
            case xs of
               [] => acc
             | x::xs' => aux(xs',x::acc)
    in
       aux(xs,[])
    end
```

Actually much better

```
fun rev xs =
   case xs of
        [] => []
      | x::xs' => (rev xs') @ [x]
```

- For **`fact`** and **`sum`**, tail-recursion is faster but both ways linear time
- Non-tail recursive **`rev`** is quadratic because each recursive call uses append, which must traverse the first list
  - And 1+2+…+(length-1) is almost length*length/2
  - Moral: beware list-append, especially within outer recursion
- Cons constant-time (and fast), so accumulator version much better

Always tail-recursive?

There are certainly cases where recursive functions cannot be evaluated in a constant amount of space

Most obvious examples are functions that process trees

In these cases, the natural recursive approach is the way to go

- You could get one recursive call to be a tail call, but rarely worth the complication

Also beware the wrath of premature optimization

- Favor clear, concise code
- But do use less space if inputs may be large

What is a tail-call?

The “nothing left for caller to do” intuition usually suffices

- If the result of **`f x`** is the “immediate result” for the enclosing function body, then **`f x`** is a tail call

But we can define “tail position” recursively

- Then a “tail call” is a function call in “tail position”

…

Precise definition

A *tail call* is a function call in *tail position*

- If an expression is not in tail position, then no subexpressions are

- In **`fun f p = e`**, the body **`e`** is in tail position
- If **`if e1 then e2 else e3`** is in tail position, then **`e2`** and **`e3`** are in tail position (but **`e1`** is not). (Similar for case-expressions)
- If **`let b1 … bn in e end`** is in tail position, then **`e`** is in tail position (but no binding expressions are)
- Function-call *arguments* **`e1 e2`** are not in tail position
- …

PAUL G. ALLEN SCHOOL
OF COMPUTER SCIENCE & ENGINEERING

CSE341: Programming Languages

Lecture 7
First-Class Functions

Dan Grossman

Spring 2019

What is functional programming?

"*Functional programming*" can mean a few different things:

1. Avoiding mutation in most/all cases (done and ongoing)
2. Using functions as values (this unit)

…

- Style encouraging recursion and recursive data structures
- Style closer to mathematical definitions
- Programming idioms using *laziness* (later topic, briefly)
- Anything not OOP or C? (not a good definition)

Not sure a definition of "*functional language*" exists beyond "makes functional programming easy / the default / required"

  – No clear yes/no for a particular language

First-class functions

- *First-class functions*: Can use them *wherever* we use values
  - Functions are values too
  - Arguments, results, parts of tuples, bound to variables, carried by datatype constructors or exceptions, …

```
fun double x = 2*x
fun incr x = x+1
val a_tuple = (double, incr, double(incr 7))
```

- Most common use is as an argument / result of another function
  - Other function is called a *higher-order function*
  - Powerful way to *factor out* common functionality

Function Closures

- *Function closure*: Functions can use bindings from outside the function definition (in scope where function is defined)
  - Makes first-class functions *much* more powerful
  - Will get to this feature in a bit, after simpler examples

- Distinction between terms *first-class functions* and *function closures* is not universally understood
  - Important conceptual distinction even if terms get muddled

Onward

The next week:

- How to use first-class functions and closures
- The precise semantics
- Multiple powerful idioms

Functions as arguments

- We can pass one function as an argument to another function
  - Not a new feature, just never thought to do it before

```
fun f (g,…) = … g (…) …
fun h1 … = …
fun h2 … = …
…   f(h1,…) … f(h2,…) …
```

- Elegant strategy for factoring out common code
  - Replace *N* similar functions with calls to 1 function where you pass in *N* different (short) functions as arguments

[See the code file for this lecture]

Example

Can reuse **`n_times`** rather than defining many similar functions

- Computes **`f(f(…f(x)))`** where number of calls is **`n`**

```
fun n_times (f,n,x) =
   if n=0
   then x
   else f (n_times(f,n-1,x))

fun double x = x + x
fun increment x = x + 1
val x1 = n_times(double,4,7)
val x2 = n_times(increment,4,7)
val x3 = n_times(tl,2,[4,8,12,16])

fun double_n_times (n,x) = n_times(double,n,x)
fun nth_tail (n,x) = n_times(tl,n,x)
```

Relation to types

- Higher-order functions are often so “generic” and “reusable” that they have polymorphic types, i.e., types with type variables
- But there are higher-order functions that are not polymorphic
- And there are non-higher-order (first-order) functions that are polymorphic
- Always a good idea to understand the type of a function, especially a higher-order function

Types for example

```
fun n_times (f,n,x) =
   if n=0
   then x
   else f (n_times(f,n-1,x))
```

- **`val n_times : ('a -> 'a) * int * 'a -> 'a`**
  - Simpler but less useful: **`(int -> int) * int * int -> int`**

- Two of our examples *instantiated* **`'a`** with **`int`**
- One of our examples *instantiated* **`'a`** with **`int list`**
- This *polymorphism* makes **`n_times`** more useful

- Type is *inferred* based on how arguments are used (later lecture)
  - Describes which types must be exactly something (e.g., **`int`**) and which can be anything but the same (e.g., **`'a`**)

Polymorphism and higher-order functions

- Many higher-order functions are polymorphic because they are so reusable that some types, “can be anything”

- But some polymorphic functions are not higher-order
  - Example: **`len : 'a list -> int`**

- And some higher-order functions are not polymorphic
  - Example: **`times_until_0 : (int -> int) * int -> int`**

```
fun times_until_zero (f,x) =
   if x=0 then 0 else 1 + times_until_zero(f, f x)
```

Note: Would be better with tail-recursion

Toward anonymous functions

- Definitions unnecessarily at top-level are still poor style:

```
fun trip x = 3*x
fun triple_n_times (f,x) = n_times(trip,n,x)
```

- So this is better (but not the best):

```
fun triple_n_times (f,x) =
  let fun trip y = 3*y
  in
     n_times(trip,n,x)
  end
```

- And this is even smaller scope
  - It makes sense but looks weird (poor style; see next slide)

```
fun triple_n_times (f,x) =
  n_times(let fun trip y = 3*y in trip end, n, x)
```

Anonymous functions

- This does not work: A function *binding* is not an *expression*

```
fun triple_n_times (f,x) =
  n_times((fun trip y = 3*y), n, x)
```

- This is the best way we were building up to: an expression form for *anonymous functions*

```
fun triple_n_times (f,x) =
  n_times((fn y => 3*y), n, x)
```

  - Like all expression forms, can appear anywhere
  - Syntax:
    - **`fn`** not **`fun`**
    - **`=>`** not **`=`**
    - no function name, just an argument pattern

Using anonymous functions

- Most common use: Argument to a higher-order function
  - Don't need a name just to pass a function

- But: Cannot use an anonymous function for a recursive function
  - Because there is no name for making recursive calls
  - If not for recursion, **`fun`** bindings would be syntactic sugar for **`val`** bindings and anonymous functions

```
fun triple x = 3*x

val triple = fn y => 3*y
```

A style point

Compare:

```
if x then true else false
```

With:

```
(fn x => f x)
```

So don’t do this:

```
n_times((fn y => tl y),3,xs)
```

When you can do this:

```
n_times(tl,3,xs)
```

Map

```
fun map (f,xs) =
    case xs of
      [] => []
    | x::xs' => (f x)::(map(f,xs'))
```

```
val map : ('a -> 'b) * 'a list -> 'b list
```

Map is, without doubt, in the "higher-order function hall-of-fame"

- The name is standard (for any data structure)
- You use it *all the time* once you know it: saves a little space, but more importantly, *communicates what you are doing*
- Similar predefined function: **`List.map`**
  - But it uses currying (coming soon)

Filter

```
fun filter (f,xs) =
    case xs of
      [] => []
    | x::xs' => if f x
                then x::(filter(f,xs'))
                else filter(f,xs')
```

```
val filter : ('a -> bool) * 'a list -> 'a list
```

Filter is also in the hall-of-fame

- So use it whenever your computation is a filter
- Similar predefined function: **`List.filter`**
  - But it uses currying (coming soon)

Generalizing

Our examples of first-class functions so far have all:

- Taken one function as an argument to another function
- Processed a number or a list

But first-class functions are useful anywhere for any kind of data

- Can pass several functions as arguments
- Can put functions in data structures (tuples, lists, etc.)
- Can return functions as results
- Can write higher-order functions that traverse your own data structures

Useful whenever you want to abstract over "what to compute with"

- No new language features

Returning functions

- Remember: Functions are first-class values
  - For example, can return them from functions

- Silly example:

```
fun double_or_triple f =
    if f 7
    then fn x => 2*x
    else fn x => 3*x
```

Has type **`(int -> bool) -> (int -> int)`**

But the REPL prints **`(int -> bool) -> int -> int`**
because it never prints unnecessary parentheses and
`t1 -> t2 -> t3 -> t4` means **`t1->(t2->(t3->t4))`**

Other data structures

- Higher-order functions are not just for numbers and lists
- They work great for common recursive traversals over your own data structures (datatype bindings) too
- Example of a higher-order *predicate*:
  - Are all constants in an arithmetic expression even numbers?
  - Use a more general function of type
    `(int -> bool) * exp -> bool`
  - And call it with **`(fn x => x mod 2 = 0)`**

PAUL G. ALLEN SCHOOL
OF COMPUTER SCIENCE & ENGINEERING

CSE341: Programming Languages

Lecture 8
Lexical Scope and Function Closures

Dan Grossman

Spring 2019

Very important concept

- We know function bodies can use any bindings in scope
- But now that functions can be passed around: In scope where?

Where the function was defined
(not where it was called)

- This semantics is called *lexical scope*
- There are lots of good reasons for this semantics (why)
  - Discussed after explaining what the semantics is (what)
  - Later in course: implementing it (how)
- Must “get this” for homework, exams, and competent programming

Example

Demonstrates lexical scope even without higher-order functions:

```
(* 1 *) val x = 1
(* 2 *) fun f y = x + y
(* 3 *) val x = 2
(* 4 *) val y = 3
(* 5 *) val z = f (x + y)
```

- Line 2 defines a function that, when called, evaluates body **x+y** in environment where **x** maps to **1** and **y** maps to the argument
- Call on line 5:
  - Looks up **f** to get the function defined on line 2
  - Evaluates **x+y** in current environment, producing **5**
  - Calls the function with **5**, which evaluates the body in the old environment, producing **6**

Closures

How can functions be evaluated in old environments that aren't around anymore?

- The language implementation keeps them around as necessary

Can define the semantics of functions as follows:

- A function value has two parts
  - The code (obviously)
  - The environment that was current when the function was defined
- This is a "pair" but unlike ML pairs, you cannot access the pieces
- All you can do is call this "pair"
- This pair is called a *function closure*
- A call evaluates the code part in the environment part (extended with the function argument)

Example

```
(* 1 *) val x = 1
(* 2 *) fun f y = x + y
(* 3 *) val x = 2
(* 4 *) val y = 3
(* 5 *) val z = f (x + y)
```

- Line 2 creates a closure and binds **f** to it:
  - Code: "take **y** and have body **x+y**"
  - Environment: "**x** maps to **1**"
    - (Plus whatever else is in scope, including **f** for recursion)

- Line 5 calls the closure defined in line 2 with **5**
  - So body evaluated in environment "**x** maps to **1**" extended with "**y** maps to **5**"

Coming up:

Now you know the rule: *lexical scope*

Next steps:

- (Silly) examples to demonstrate how the rule works with higher-order functions
- Why the other natural rule, *dynamic scope*, is a bad idea
- Powerful *idioms* with higher-order functions that use this rule
  - Passing functions to iterators like **`filter`**
  - Next lecture: Several more idioms

The rule stays the same

A function body is evaluated in the environment where the function was defined (created)

- Extended with the function argument

Nothing changes to this rule when we take and return functions

- But “the environment” may involve nested let-expressions, not just the top-level sequence of bindings

Makes first-class functions much more powerful

- Even if may seem counterintuitive at first

Example: Returning a function

```
(* 1 *) val x = 1
(* 2 *) fun f y =
(* 2a *)     let val x = y+1
(* 2b *)     in fn z => x+y+z end
(* 3 *) val x = 3
(* 4 *) val g = f 4
(* 5 *) val y = 5
(* 6 *) val z = g 6
```

- Trust the rule: Evaluating line 4 binds to **g** to a closure:
  - Code: "take **z** and have body **x+y+z**"
  - Environment: "**y** maps to **4**, **x** maps to **5** (shadowing), …"
  - So this closure will always add **9** to its argument
- So line 6 binds **15** to **z**

Example: Passing a function

```
(* 1 *) fun f g = (* call arg with 2 *)
(* 1a *)     let val x = 3
(* 1b *)     in g 2 end
(* 2 *) val x = 4
(* 3 *) fun h y = x + y
(* 4 *) val z = f h
```

- Trust the rule: Evaluating line 3 binds **h** to a closure:
  - Code: "take **y** and have body **x+y**"
  - Environment: "**x** maps to **4**, **f** maps to a closure, …"
  - So this closure will always add **4** to its argument
- So line 4 binds **6** to **z**
  - Line 1a is as stupid and irrelevant as it should be

Why lexical scope

- *Lexical scope*: use environment where function is defined
- *Dynamic scope*: use environment where function is called

Decades ago, both might have been considered reasonable, but now we know lexical scope makes much more sense

Here are three precise, technical reasons

  – Not a matter of opinion

Why lexical scope?

1. Function meaning does not depend on variable names used

Example: Can change body of **f** to use **q** everywhere instead of **x**

- Lexical scope: it cannot matter
- Dynamic scope: depends how result is used

```
fun f y =
    let val x = y+1
    in fn z => x+y+z end
```

Example: Can remove unused variables

- Dynamic scope: but maybe some **g** uses it (weird)

```
fun f g =
    let val x = 3
    in g 2 end
```

Why lexical scope?

2. Functions can be type-checked and reasoned about where defined

Example: Dynamic scope tries to add a string and an unbound variable to **6**

```
val x = 1
fun f y =
    let val x = y+1
    in fn z => x+y+z end
val x = "hi"
val g = f 7
val z = g 4
```

Why lexical scope?

3. Closures can easily store the data they need
   - Many more examples and idioms to come

```
fun greaterThanX x = fn y => y > x

fun filter (f,xs) =
   case xs of
     [] => []
   | x::xs => if f x
              then x::(filter(f,xs))
              else filter(f,xs)

fun noNegatives xs = filter(greaterThanX ~1, xs)
fun allGreater (xs,n) = filter(fn x => x > n, xs)
```

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., Racket) have special ways to do it
  - But most do not bother

- If you squint some, exception handling is more like dynamic scope:
  - **`raise e`** transfers control to the current innermost handler
  - Does not have to be syntactically inside a handle expression (and usually is not)

When things evaluate

Things we know:

- A function body is not evaluated until the function is called
- A function body is evaluated every time the function is called
- A variable binding evaluates its expression when the binding is evaluated, not every time the variable is used

With closures, this means we can avoid repeating computations that do not depend on function arguments

- Not so worried about performance, but good example to emphasize the semantics of functions

Recomputation

These both work and rely on using variables in the environment

```
fun allShorterThan1 (xs,s) =
    filter(fn x => String.size x < String.size s,
           xs)

fun allShorterThan2 (xs,s) =
    let val i = String.size s
    in filter(fn x => String.size x < i, xs) end
```

The first one computes **`String.size`** once per element of **`xs`**

The second one computes **`String.size s`** once per list

- Nothing new here: let-bindings are evaluated when encountered and function bodies evaluated when *called*

Another famous function: Fold

`fold` (and synonyms / close relatives reduce, inject, etc.) is another very famous iterator over recursive structures

Accumulates an answer by repeatedly applying **`f`** to answer so far

- **`fold(f,acc,[x1,x2,x3,x4])`** computes **`f(f(f(f(acc,x1),x2),x3),x4)`**

```
fun fold (f,acc,xs) =
   case xs of
     []    => acc
   | x::xs => fold(f, f(acc,x), xs)
```

- This version "folds left"; another version "folds right"
- Whether the direction matters depends on **`f`** (often not)

```
val fold = fn : ('a * 'b -> 'a) * 'a * 'b list -> 'a
```

Why iterators again?

- These “iterator-like” functions are not built into the language
  - Just a programming pattern
  - Though many languages have built-in support, which often allows stopping early without resorting to exceptions

- This pattern separates recursive traversal from data processing
  - Can reuse same traversal for different data processing
  - Can reuse same data processing for different data structures
  - In both cases, using common vocabulary concisely communicates intent

Examples with fold

These are useful and do not use “private data”

```
fun f1 xs = fold((fn (x,y) => x+y), 0, xs)
fun f2 xs = fold((fn (x,y) => x andalso y>=0),
                 true, xs)
```

These are useful and do use “private data”

```
fun f3 (xs,hi,lo) =
    fold(fn (x,y) =>
            x + (if y >= lo andalso y <= hi
                 then 1
                 else 0)),
         0, xs)
fun f4 (g,xs) = fold(fn (x,y) => x andalso g y),
                     true, xs)
```

Iterators made better

- Functions like **`map`**, **`filter`**, and **`fold`** are *much* more powerful thanks to closures and lexical scope
- Function passed in can use any “private” data in its environment
- Iterator “doesn’t even know the data is there” or what type it has

PAUL G. ALLEN SCHOOL
OF COMPUTER SCIENCE & ENGINEERING

CSE341: Programming Languages

Lecture 9
Function-Closure Idioms

Dan Grossman

Spring 2019

More idioms

- We know the rule for lexical scope and function closures
  - Now what is it good for

A partial but wide-ranging list:

- Pass functions with private data to iterators: Done
- Combine functions (e.g., composition)
- Currying (multi-arg functions and partial application)
- Callbacks (e.g., in reactive programming)
- Implementing an ADT with a record of functions (optional)

Combine functions

Canonical example is function composition:

```
fun compose (f,g) = fn x => f (g x)
```

- Creates a closure that "remembers" what **f** and **g** are bound to
- Type **('b -> 'c) * ('a -> 'b) -> ('a -> 'c)**
  but the REPL prints something *equivalent*

- ML standard library provides this as infix operator **o**
- Example (third version best):

```
fun sqrt_of_abs i = Math.sqrt(Real.fromInt(abs i))
fun sqrt_of_abs i = (Math.sqrt o Real.fromInt o abs) i
val sqrt_of_abs = Math.sqrt o Real.fromInt o abs
```

Left-to-right or right-to-left

```
val sqrt_of_abs = Math.sqrt o Real.fromInt o abs
```

As in math, function composition is "right to left"

- "take absolute value, convert to real, and take square root"
- "square root of the conversion to real of absolute value"

"Pipelines" of functions are common in functional programming and many programmers prefer left-to-right

- Can define our own infix operator
- This one is very popular (and predefined) in F#

```
infix |>
fun x |> f = f x

fun sqrt_of_abs i =
   i |> abs |> Real.fromInt |> Math.sqrt
```

Another example

- "Backup function"

```
fun backup1 (f,g) =
    fn x => case f x of
              NONE => g x
            | SOME y => y
```

- As is often the case with higher-order functions, the types hint at what the function does

```
('a -> 'b option) * ('a -> 'b) -> 'a -> 'b
```

More idioms

- We know the rule for lexical scope and function closures
  - Now what is it good for

A partial but wide-ranging list:

- Pass functions with private data to iterators: Done
- Combine functions (e.g., composition)
- Currying (multi-arg functions and partial application)
- Callbacks (e.g., in reactive programming)
- Implementing an ADT with a record of functions (optional)

Currying

- Recall every ML function takes exactly one argument
- Previously encoded *n* arguments via one *n*-tuple
- Another way: Take one argument and return a function that takes another argument and…
  - Called “currying” after famous logician Haskell Curry

Example

```
val sorted3 = fn x => fn y => fn z =>
                  z >= y andalso y >= x

val t1 = ((sorted3 7) 9) 11
```

- Calling **`(sorted3 7)`** returns a closure with:
  - Code **`fn y => fn z => z >= y andalso y >= x`**
  - Environment maps **`x`** to **`7`**
- Calling *that* closure with **`9`** returns a closure with:
  - Code **`fn z => z >= y andalso y >= x`**
  - Environment maps **`x`** to **`7`**, **`y`** to **`9`**
- Calling *that* closure with **`11`** returns **`true`**

Syntactic sugar, part 1

```
val sorted3 = fn x => fn y => fn z =>
                  z >= y andalso y >= x

val t1 = ((sorted3 7) 9) 11
```

- In general, **`e1 e2 e3 e4 …,`**
  means **`(…((e1 e2) e3) e4)`**
- So instead of **`((sorted3 7) 9) 11`**,
  can just write **`sorted3 7 9 11`**
- Callers can just think “multi-argument function with spaces instead of a tuple expression”
  - Different than tupling; caller and callee must use same technique

Syntactic sugar, part 2

```
val sorted3 = fn x => fn y => fn z =>
                  z >= y andalso y >= x

val t1 = ((sorted3 7) 9) 11
```

- In general, **`fun f p1 p2 p3 … = e`**,
  means **`fun f p1 = fn p2 => fn p3 => … => e`**
- So instead of **`val sorted3 = fn x => fn y => fn z => …`**
  or **`fun sorted3 x = fn y => fn z => …`**,
  can just write **`fun sorted3 x y z = x >=y andalso y >= x`**
- Callees can just think "multi-argument function with spaces instead of a tuple pattern"
  - Different than tupling; caller and callee must use same technique

Final version

```
fun sorted3 x y z = z >= y andalso y >= x

val t1 = sorted3 7 9 11
```

As elegant syntactic sugar (even fewer characters than tupling) for:

```
val sorted3 = fn x => fn y => fn z =>
                  z >= y andalso y >= x

val t1 = ((sorted3 7) 9) 11
```

Curried fold

A more useful example and a call to it

- Will improve call next

```
fun fold f acc xs =
    case xs of
      []     => acc
    | x::xs' => fold f (f(acc,x)) xs'

fun sum xs = fold (fn (x,y) => x+y) 0 xs
```

Note: **`foldl`** in ML standard-library has **`f`** take arguments in opposite order

"Too Few Arguments"

- Previously used currying to simulate multiple arguments
- But if caller provides "too few" arguments, we get back a closure "waiting for the remaining arguments"
  - Called partial application
  - Convenient and useful
  - Can be done with any curried function
- No new semantics here: a pleasant idiom

Example

```
fun fold f acc xs =
    case xs of
      []     => acc
    | x::xs' => fold f (f(acc,x)) xs'

fun sum_inferior xs = fold (fn (x,y) => x+y) 0 xs

val sum = fold (fn (x,y) => x+y) 0
```

As we already know, **`fold (fn (x,y) => x+y) 0`**

evaluates to a closure that given **`xs`**, evaluates the case-expression with **`f`** bound to **`fold (fn (x,y) => x+y)`** and **`acc`** bound to **`0`**

Unnecessary function wrapping

```
fun sum_inferior xs = fold (fn (x,y) => x+y) 0 xs

val sum = fold (fn (x,y) => x+y) 0
```

- Previously learned not to write `fun f x = g x` when we can write `val f = g`

- This is the same thing, with `fold (fn (x,y) => x+y) 0` in place of `g`

Iterators

- Partial application is particularly nice for iterator-like functions
- Example:

```
fun exists predicate xs =
    case xs of
      []     => false
    | x::xs' => predicate x
                orelse exists predicate xs'

val no = exists (fn x => x=7) [4,11,23]
val hasZero = exists (fn x => x=0)
```

- For this reason, ML library functions of this form usually curried
  - Examples: **`List.map`**, **`List.filter`**, **`List.foldl`**

The Value Restriction Appears ☹

If you use partial application to *create a polymorphic function*, it may not work due to the value restriction

- Warning about "type vars not generalized"
  - And won't let you call the function
- This should surprise you; you did nothing wrong ☺ but you still must change your code
- See the code for workarounds
- Can discuss a bit more when discussing type inference

More combining functions

- What if you want to curry a tupled function or vice-versa?
- What if a function's arguments are in the wrong order for the partial application you want?

Naturally, it is easy to write higher-order wrapper functions

  – And their types are neat logical formulas

```
fun other_curry1 f = fn x => fn y => f y x
fun other_curry2 f x y = f y x
fun curry f x y = f (x,y)
fun uncurry f (x,y) = f x y
```

Efficiency

So which is faster: tupling or currying multiple-arguments?

- They are both constant-time operations, so it doesn't matter in most of your code – "plenty fast"
  - Don't program against an *implementation* until it matters!

- For the small (zero?) part where efficiency matters:
  - It turns out SML/NJ compiles tuples more efficiently
  - But many other functional-language implementations do better with currying (OCaml, F#, Haskell)
    - So currying is the "normal thing" and programmers read **`t1 -> t2 -> t3 -> t4`** as a 3-argument function that also allows partial application

More idioms

- We know the rule for lexical scope and function closures
  - Now what is it good for

A partial but wide-ranging list:

- Pass functions with private data to iterators: Done
- Combine functions (e.g., composition)
- Currying (multi-arg functions and partial application)
- Callbacks (e.g., in reactive programming)
- Implementing an ADT with a record of functions (optional)

ML has (separate) mutation

- Mutable data structures are okay in some situations
  - When “update to state of world” is appropriate model
  - But want most language constructs truly immutable
- ML does this with a separate construct: references
- Introducing now because will use them for next closure idiom
- Do not use references on your homework
  - You need practice with mutation-free programming
  - They will lead to less elegant solutions

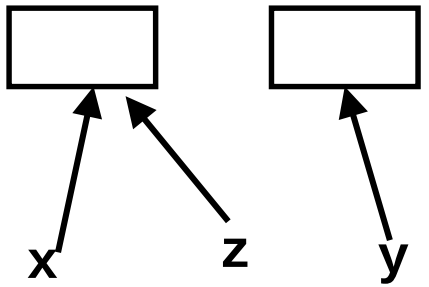
References

- New types: **`t ref`** where **`t`** is a type

- New expressions:
  - **`ref e`** to create a reference with initial contents **`e`**
  - **`e1 := e2`** to update contents
  - **`!e`** to retrieve contents (not negation)

References example

```
val x = ref 42
val y = ref 42
val z = x
val _ = x := 43
val w = (!y) + (!z) (* 85 *)
(* x + 1 does not type-check *)
```



- A variable bound to a reference (e.g., **x**) is still immutable: it will always refer to the same reference
- But the contents of the reference may change via **:=**
- And there may be aliases to the reference, which matter a lot
- References are first-class values
- Like a one-field mutable object, so **:=** and **!** don't specify the field

Callbacks

A common idiom: Library takes functions to apply later, when an *event* occurs – examples:

- When a key is pressed, mouse moves, data arrives
- When the program enters some state (e.g., turns in a game)

A library may accept multiple callbacks

- Different callbacks may need different private data with different types
- Fortunately, a function's type does not include the types of bindings in its environment
- (In OOP, objects and private fields are used similarly, e.g., Java Swing's event-listeners)

Mutable state

While it's not absolutely necessary, mutable state is reasonably appropriate here

- We really do want the "callbacks registered" to *change* when a function to register a callback is called

Example call-back library

Library maintains mutable state for "what callbacks are there" and provides a function for accepting new ones

- A real library would also support removing them, etc.
- In example, callbacks have type **`int->unit`**

So the entire public library interface would be the function for registering new callbacks:

```
val onKeyEvent : (int -> unit) -> unit
```

(Because callbacks are executed for side-effect, they may also need mutable state)

Library implementation

```
val cbs : (int -> unit) list ref = ref []

fun onKeyEvent f =  cbs := f :: (!cbs)

fun onEvent i =
    let fun loop fs =
            case fs of
              []     => ()
            | f::fs' => (f i; loop fs')
    in loop (!cbs) end
```

Clients

Can only register an **`int -> unit`**, so if any other data is needed, must be in closure's environment

- And if need to "remember" something, need mutable state

Examples:

```
val timesPressed = ref 0
val _ = onKeyEvent (fn _ =>
            timesPressed := (!timesPressed) + 1)

fun printIfPressed i =
   onKeyEvent (fn j =>
       if i=j
       then print ("pressed " ^ Int.toString i)
       else ())
```

More idioms

- We know the rule for lexical scope and function closures
  - Now what is it good for

A partial but wide-ranging list:

- Pass functions with private data to iterators: Done
- Combine functions (e.g., composition)
- Currying (multi-arg functions and partial application)
- Callbacks (e.g., in reactive programming)
- Implementing an ADT with a record of functions (optional)

Optional: Implementing an ADT

As our last idiom, closures can implement abstract data types

- Can put multiple functions in a record
- The functions can share the same private data
- Private data can be mutable or immutable
- Feels a lot like objects, emphasizing that OOP and functional programming have some deep similarities

See code for an implementation of immutable integer sets with operations *insert*, *member*, and *size*

The actual code is advanced/clever/tricky, but has no new features

- Combines lexical scope, datatypes, records, closures, etc.
- Client use is not so tricky

PAUL G. ALLEN SCHOOL
OF COMPUTER SCIENCE & ENGINEERING

CSE341: Programming Languages

Lecture 10
ML Modules

Dan Grossman

Spring 2019

Modules

For larger programs, one "top-level" sequence of bindings is poor

- Especially because a binding can use *all* earlier (non-shadowed) bindings

So ML has *structures* to define *modules*

```
structure MyModule = struct bindings end
```

Inside a module, can use earlier bindings as usual

- Can have any kind of binding (val, datatype, exception, ...)

Outside a module, refer to earlier modules' bindings via **`ModuleName.bindingName`**

- Just like **`List.foldl`** and **`Char.toLower`**; now you can define your own modules

Example

```
structure MyMathLib =
struct

fun fact x =
    if x=0
    then 1
    else x * fact(x-1)

val half_pi = Math.pi / 2

fun doubler x = x * 2

end
```

Namespace management

- *So far*, this is just *namespace management*
  - Giving a hierarchy to names to avoid shadowing
  - Allows different modules to reuse names, e.g., **`map`**
  - Very important, but not very interesting

Optional: Open

- Can use **`open ModuleName`** to get "direct" access to a module's bindings
  - Never necessary; just a convenience; often bad style
  - Often better to create local val-bindings for just the bindings you use a lot, e.g., **`val map = List.map`**
    - But doesn't work for patterns
    - And **`open`** can be useful, e.g., for testing code

Signatures

- A *signature* is a type for a module
  - What bindings does it have and what are their types
- Can define a signature and ascribe it to modules – example:

```
signature MATHLIB =
sig
val fact : int -> int
val half_pi : real
val doubler : int -> int
end

structure MyMathLib :> MATHLIB =
struct
fun fact x = …
val half_pi = Math.pi / 2.0
fun doubler x = x * 2
end
```

In general

- Signatures

```
signature SIGNAME =
sig types-for-bindings end
```

  - Can include variables, types, datatypes, and exceptions defined in module

- Ascribing a signature to a module

```
structure MyModule :> SIGNAME =
struct bindings end
```

  - Module will not type-check unless it matches the signature, meaning it has all the bindings at the right types

  - Note: SML has other forms of ascription; we will stick with these [opaque signatures]

Hiding things

Real value of signatures is to to *hide* bindings and type definitions

- So far, just documenting and checking the types

Hiding implementation details is the most important strategy for writing correct, robust, reusable software

So first remind ourselves that functions already do well for some forms of hiding…

Hiding with functions

These three functions are totally equivalent: no client can tell which we are using (so we can change our choice later):

```
fun double x = x*2
fun double x = x+x
val y = 2
fun double x = x*y
```

Defining helper functions locally is also powerful

- Can change/remove functions later and know it affects no other code

Would be convenient to have "private" top-level functions too

- So two functions could easily share a helper function
- ML does this via signatures that omit bindings…

Example

Outside the module, **`MyMathLib.doubler`** is simply unbound

- So cannot be used [directly]
- Fairly powerful, very simple idea

```
signature MATHLIB =
sig
val fact : int -> int
val half_pi : real
end

structure MyMathLib :> MATHLIB =
struct
fun fact x = …
val half_pi = Math.pi / 2.0
fun doubler x = x * 2
end
```

A larger example [mostly see the code]

Now consider a module that defines an Abstract Data Type (ADT)

- A type of data and operations on it

Our example: rational numbers supporting **`add`** and **`toString`**

```
structure Rational1 =
struct
datatype rational = Whole of int | Frac of int*int
exception BadFrac

(*internal functions gcd and reduce not on slide*)

fun make_frac (x,y) = …
fun add (r1,r2) = …
fun toString r = …
end
```

Library spec and invariants

Properties [externally visible guarantees, up to library writer]

- Disallow denominators of 0
- Return strings in reduced form (“4” not “4/1”, “3/2” not “9/6”)
- No infinite loops or exceptions

Invariants [part of the implementation, not the module’s spec]

- All denominators are greater than 0
- All **`rational`** values returned from functions are reduced

More on invariants

Our code maintains the invariants and relies on them

Maintain:

- **`make_frac`** disallows 0 denominator, removes negative denominator, and reduces result
- **`add`** assumes invariants on inputs, calls **`reduce`** if needed

Rely:

- **`gcd`** does not work with negative arguments, but no denominator can be negative
- **`add`** uses math properties to avoid calling **`reduce`**
- **`toString`** assumes its argument is already reduced

A first signature

With what we know so far, this signature makes sense:

- **`gcd`** and **`reduce`** not visible outside the module

```
signature RATIONAL_A =
sig
datatype rational = Whole of int | Frac of int*int
exception BadFrac
val make_frac : int * int -> rational
val add : rational * rational -> rational
val toString : rational -> string
end

structure Rational1 :> RATIONAL_A = …
```

The problem

By revealing the datatype definition, we let clients violate our invariants by directly creating values of type **`Rational1.rational`**

- At best a comment saying "must use **`Rational1.make_frac`**"

```
signature RATIONAL_A =
sig
datatype rational = Whole of int | Frac of int*int
…
```

Any of these would lead to exceptions, infinite loops, or wrong results, which is why the module's code would never return them

- **`Rational1.Frac(1,0)`**
- **`Rational1.Frac(3,~2)`**
- **`Rational1.Frac(9,6)`**

So hide more

Key idea: An ADT must hide the concrete type definition so clients cannot create invariant-violating values of the type directly

Alas, this attempt doesn't work because the signature now uses a type **`rational`** that is not known to exist:

```
signature RATIONAL_WRONG =
sig
exception BadFrac
val make_frac : int * int -> rational
val add : rational * rational -> rational
val toString : rational -> string
end

structure Rational1 :> RATIONAL_WRONG = …
```

Abstract types

So ML has a feature for exactly this situation:

In a signature:

`type foo`

means the type exists, but clients do not know its definition

```
signature RATIONAL_B =
sig
type rational
exception BadFrac
val make_frac : int * int -> rational
val add : rational * rational -> rational
val toString : rational -> string
end

structure Rational1 :> RATIONAL_B = …
```

This works! (And is a Really Big Deal)

```
signature RATIONAL_B =
sig
type rational
exception BadFrac
val make_frac : int * int -> rational
val add : rational * rational -> rational
val toString : rational -> string
end
```

Nothing a client can do to violate invariants and properties:

- Only way to make first rational is **`Rational1.make_frac`**
- After that can use only **`Rational1.make_frac`**, **`Rational1.add`**, and **`Rational1.toString`**
- Hides constructors and patterns – don't even know whether or not **`Rational1.rational`** is a datatype
- But clients can still pass around fractions in any way

Two key restrictions

So we have two powerful ways to use signatures for hiding:

1. Deny bindings exist (val-bindings, fun-bindings, constructors)

2. Make types abstract (so clients cannot create values of them or access their pieces directly)

(Later we will see a signature can also make a binding's type more specific than it is within the module, but this is less important)

A cute twist

In our example, exposing the **`Whole`** constructor is no problem

In SML we can expose it as a function since the datatype binding in the module does create such a function

- Still hiding the rest of the datatype
- Still does not allow using **`Whole`** as a pattern

```
signature RATIONAL_C =
sig
type rational
exception BadFrac
val Whole : int -> rational
val make_frac : int * int -> rational
val add : rational * rational -> rational
val toString : rational -> string
end
```

Signature matching

Have so far relied on an informal notion of, "does a module type-check given a signature?" As usual, there are precise rules…

`structure Foo :> BAR` is allowed if:

- Every non-abstract type in **BAR** is provided in **Foo**, as specified
- Every abstract type in **BAR** is provided in **Foo** in some way
  - Can be a datatype or a type synonym
- Every val-binding in **BAR** is provided in **Foo**, possibly with a *more general* and/or *less abstract* internal type
  - Discussed "more general types" earlier in course
  - Will see example soon
- Every exception in **BAR** is provided in **Foo**

Of course **Foo** can have more bindings (implicit in above rules)

Equivalent implementations

A key purpose of abstraction is to allow *different implementations* to be *equivalent*

- *No* client can tell which you are using
- So can improve/replace/choose implementations later
- Easier to do if you *start* with more abstract signatures (reveal only what you must)

Now:

Another structure that can also have signature **`RATIONAL_A`**, **`RATIONAL_B`**, or **`RATIONAL_C`**

- But only *equivalent* under **`RATIONAL_B`** or **`RATIONAL_C`**
  (ignoring overflow)

Next:

A third equivalent structure implemented very differently

Equivalent implementations

Example (see code file):

- **`structure Rational2`** does not keep rationals in reduced form, instead reducing them "at last moment" in **`toString`**
  - Also make **`gcd`** and **`reduce`** local functions

- Not equivalent under **`RATIONAL_A`**
  - **`Rational1.toString(Rational1.Frac(9,6)) = "9/6"`**
  - **`Rational2.toString(Rational2.Frac(9,6)) = "3/2"`**

- Equivalent under **`RATIONAL_B`** or **`RATIONAL_C`**
  - Different invariants, but same properties
  - Essential that type **`rational`** is abstract

More interesting example

Given a signature with an abstract type, different structures can:

- Have that signature
- But implement the abstract type differently

Such structures might or might not be equivalent

Example (see code):

- **`type rational = int * int`**
- Does *not* have signature **`RATIONAL_A`**
- *Equivalent* to both previous examples under **`RATIONAL_B`** or **`RATIONAL_C`**

More interesting example

```
structure Rational3 =
struct
type rational = int * int
exception BadFrac

fun make_frac (x,y) = …
fun Whole i = (i,1) (* needed for RATIONAL_C *)
fun add ((a,b)(c,d)) = (a*d+b*c,b*d)
fun toString r = … (* reduce at last minute *)
end
```

Some interesting details

- Internally **`make_frac`** has type **`int * int -> int * int`**, but externally **`int * int -> rational`**
  - Client cannot tell if we return argument unchanged
  - Could give type **`rational -> rational`** in signature, but this is awful: makes entire module unusable – why?

- Internally **`Whole`** has type **`'a -> 'a * int`** but externally **`int -> rational`**
  - This matches because we can specialize **`'a`** to **`int`** and then abstract **`int * int`** to **`rational`**
  - **`Whole`** cannot have types **`'a -> int * int`** or **`'a -> rational`** (must specialize all **`'a`** uses)
  - Type-checker figures all this out for us

Can't mix-and-match module bindings

Modules with the *same signatures* still define *different types*

So things like this do not type-check:

- **`Rational1.toString(Rational2.make_frac(9,6))`**
- **`Rational3.toString(Rational2.make_frac(9,6))`**

This is a crucial feature for type system and module properties:

- Different modules have different internal invariants!
- In fact, they have different type definitions
  - **`Rational1.rational`** looks like **`Rational2.rational`**, but clients and the type-checker do not know that
  - **`Rational3.rational`** is **`int*int`** not a datatype!

PAUL G. ALLEN SCHOOL
OF COMPUTER SCIENCE & ENGINEERING

CSE341: Programming Languages

Lecture 11
Type Inference

Dan Grossman

Spring 2019

Type-checking

- (Static) type-checking can reject a program before it runs to prevent the possibility of some errors
  - A feature of statically typed languages

- Dynamically typed languages do little (none?) such checking
  - So might try to treat a number as a function at run-time

- Will study relative advantages after some Racket
  - Racket, Ruby (and Python, Javascript, …) dynamically typed

- ML (and Java, C#, Scala, C, C++) is statically typed
  - Every binding has one type, determined "at compile-time"

Implicitly typed

- ML is statically typed
- ML is implicitly typed: rarely need to write down types

```
fun f x = (* infer val f : int -> int *)
    if x > 3
    then 42
    else x * 2

fun g x = (* report type error *)
    if x > 3
    then true
    else x * 2
```

- Statically typed: Much more like Java than Javascript!

Type inference

- Type inference problem: Give every binding/expression a type such that type-checking succeeds
  - Fail if and only if no solution exists

- In principle, could be a pass before the type-checker
  - But often implemented together

- Type inference can be easy, difficult, or *impossible*
  - Easy: Accept all programs
  - Easy: Reject all programs
  - Subtle, elegant, and *not magic*: ML

Overview

- Will describe ML type inference via several examples
  - General algorithm is a slightly more advanced topic
  - Supporting nested functions also a bit more advanced

- Enough to help you “do type inference in your head”
  - And appreciate it is not magic

Key steps

- Determine types of bindings in order
  - (Except for mutual recursion)
  - So you cannot use later bindings: will not type-check
- For each **`val`** or **`fun`** binding:
  - Analyze definition for all necessary facts (constraints)
  - Example: If see **`x > 0`**, then **`x`** must have type **`int`**
  - Type error if no way for all facts to hold (over-constrained)
- Afterward, use type variables (e.g., **`'a`**) for any unconstrained types
  - Example: An unused argument can have any type
- (Finally, enforce the *value restriction*, discussed later)

Very simple example

After this example, will go much more step-by-step

- Like the automated algorithm does

```
val x = 42 (* val x : int *)

fun f (y, z, w) =
    if y (* y must be bool *)
    then z + x (* z must be int *)
    else 0 (* both branches have same type *)
(* f must return an int
   f must take a bool * int * ANYTHING
   so val f : bool * int * 'a -> int
 *)
```

Relation to Polymorphism

- Central feature of ML type inference: it can infer types with type variables
  - Great for code reuse and understanding functions

- But remember there are two orthogonal concepts
  - Languages can have type inference without type variables
  - Languages can have type variables without type inference

Key Idea

- Collect all the facts needed for type-checking
- These facts constrain the type of the function
- See code and/or reading notes for:
  - Two examples without type variables
  - And one example that does not type-check
  - Then examples for polymorphic functions
    - Nothing changes, just under-constrained: some types can “be anything” but may still need to be the same as other types

Material after here is optional,
but is an important part of the full story

Two more topics

- ML type-inference story so far is too lenient
  - Value restriction limits where polymorphic types can occur
  - See why and then what

- ML is in a “sweet spot”
  - Type inference more difficult without polymorphism
  - Type inference more difficult with subtyping

Important to “finish the story” but these topics are:

- A bit more advanced
- A bit less elegant
- Will not be on the exam

The Problem

As presented so far, the ML type system is *unsound*!

- Allows putting a value of type **`t1`** (e.g., **`int`**) where we expect a value of type **`t2`** (e.g., **`string`**)

A combination of polymorphism and mutation is to blame:

```
val r = ref NONE (* val r : 'a option ref *)

val _ = r := SOME "hi"

val i = 1 + valOf (!r)
```

- Assignment type-checks because (infix) **`:=`** has type **`'a ref * 'a -> unit`**, so instantiate with **`string`**
- Dereference type-checks because **`!`** has type **`'a ref -> 'a`**, so instantiate with **`int`**

What to do

To restore soundness, need a stricter type system that rejects at least one of these three lines

```
val r = ref NONE (* val r : 'a option ref *)

val _ = r := SOME "hi"

val i = 1 + valOf (!r)
```

- And cannot make special rules for reference types because type-checker cannot know the definition of all type synonyms
  - Due to module system

```
type 'a foo = 'a ref
val f = ref (* val f : 'a -> 'a foo *)
val r = f NONE
```

The fix

```
val r = ref NONE (* val r : ?.X1 option ref *)
val _ = r := SOME "hi"
val i = 1 + valOf (!r)
```

- Value restriction: a variable-binding can have a polymorphic type only if the expression is a variable or value
  - Function calls like **`ref NONE`** are neither

- Else get a warning and unconstrained types are filled in with dummy types (basically unusable)

- Not obvious this suffices to make type system sound, but it does

The downside

As we saw previously, the value restriction can cause problems when it is unnecessary because we are not using mutation

```
val pairWithOne = List.map (fn x => (x,1))
(* does not get type 'a list -> ('a*int) list *)
```

The type-checker does not know **`List.map`** is not making a mutable reference

Saw workarounds in previous segment on partial application

- Common one: wrap in a function binding

```
fun pairWithOne xs = List.map (fn x => (x,1)) xs
(* 'a list -> ('a*int) list *)
```

A local optimum

- Despite the value restriction, ML type inference is elegant and fairly easy to understand

- More difficult *without* polymorpism
  - What type should length-of-list have?

- More difficult *with* subtyping
  - Suppose pairs are supertypes of wider tuples
  - Then **`val (y,z) = x`** constrains **`x`** to have at least two fields, not exactly two fields
  - Depending on details, languages can support this, but types often more difficult to infer and understand

  - Will study subtyping later, but not with type inference

PAUL G. ALLEN SCHOOL
OF COMPUTER SCIENCE & ENGINEERING

CSE341: Programming Languages

Lecture 12
Equivalence

Dan Grossman

Spring 2019

Last Topic of Unit

More careful look at what "two pieces of code are equivalent" means

- Fundamental software-engineering idea
- Made easier with
  - Abstraction (hiding things)
  - Fewer side effects

Not about any "new ways to code something up"

Equivalence

Must reason about "are these equivalent" *all the time*

- The more precisely you think about it the better

- *Code maintenance:* Can I simplify this code?
- *Backward compatibility:* Can I add new features without changing how any old features work?
- *Optimization:* Can I make this code faster?
- *Abstraction:* Can an external client tell I made this change?

To focus discussion: When can we say two functions are equivalent, even without looking at all calls to them?

- May not know all the calls (e.g., we are editing a library)

A definition

Two functions are equivalent if they have the same "observable behavior" no matter how they are used anywhere in any program

Given equivalent arguments, they:

- Produce equivalent results
- Have the same (non-)termination behavior
- Mutate (non-local) memory in the same way
- Do the same input/output
- Raise the same exceptions

Notice it is much easier to be equivalent if:

- There are fewer possible arguments, e.g., with a type system and abstraction
- We avoid *side-effects*: mutation, input/output, and exceptions

Example

Since looking up variables in ML has no side effects, these two functions are equivalent:

```
fun f x = x + x
```

$=$

```
val y = 2
fun f x = y * x
```

But these next two are not equivalent in general: it depends on what is passed for **f**

- Are equivalent *if* argument for **f** has no side-effects

```
fun g (f,x) =
   (f x) + (f x)
```

$\neq$

```
val y = 2
fun g (f,x) =
   y * (f x)
```

- Example: `g ((fn i => print "hi" ; i), 7)`
- Great reason for "pure" functional programming

Another example

These are equivalent *only if* functions bound to **g** and **h** do not raise exceptions or have side effects (printing, updating state, etc.)

- Again: pure functions make more things equivalent

```
fun f x =
    let
      val y = g x
      val z = h x
    in
      (y,z)
    end
```

≠

```
fun f x =
    let
      val z = h x
      val y = g x
    in
      (y,z)
    end
```

- Example: **g** divides by **0** and **h** mutates a top-level reference
- Example: **g** writes to a reference that **h** reads from

One that really matters

Once again, turning the left into the right is great but only if the functions are pure:

```
map f (map g xs)
```

```
map (f o g) xs
```

Syntactic sugar

Using or not using syntactic sugar is always equivalent

- By definition, else not syntactic sugar

Example:

```
fun f x =
    x andalso g x
```

$=$

```
fun f x =
    if x
    then g x
    else false
```

But be careful about evaluation order

```
fun f x =
    x andalso g x
```

$\neq$

```
fun f x =
    if g x
    then x
    else false
```

Standard equivalences

Three general equivalences that always work for functions

- In any (?) decent language

1. Consistently rename bound variables and uses

```
val y = 14
fun f x = x+y+x
```

$\equiv$

```
val y = 14
fun f z = z+y+z
```

But notice you can't use a variable name already used in the function body to refer to something else

```
val y = 14
fun f x = x+y+x
```

$\not\equiv$

```
val y = 14
fun f y = y+y+y
```

```
fun f x =
  let val y = 3
  in x+y end
```

$\not\equiv$

```
fun f y =
  let val y = 3
  in y+y end
```

Standard equivalences

Three general equivalences that always work for functions

- In (any?) decent language

2. Use a helper function or do not

```
val y = 14
fun g z = (z+y+z)+z
```

$=$

```
val y = 14
fun f x = x+y+x
fun g z = (f z)+z
```

But notice you need to be careful about environments

```
val y = 14
val y = 7
fun g z = (z+y+z)+z
```

$\neq$

```
val y = 14
fun f x = x+y+x
val y = 7
fun g z = (f z)+z
```

Standard equivalences

Three general equivalences that always work for functions

- In (any?) decent language

3. Unnecessary function wrapping

```
fun f x = x+x
fun g y = f y
```

$\equiv$

```
fun f x = x+x
val g = f
```

But notice that if you compute the function to call and *that computation* has side-effects, you have to be careful

```
fun f x = x+x
fun h () = (print "hi";
            f)
fun g y = (h()) y
```

$\not\equiv$

```
fun f x = x+x
fun h () = (print "hi";
            f)
val g = (h())
```

One more

If we ignore types, then ML let-bindings can be syntactic sugar for calling an anonymous function:

```
let val x = e1
in e2 end
```

```
(fn x => e2) e1
```

- These both evaluate **e1** to **v1**, then evaluate **e2** in an environment extended to map **x** to **v1**
- So *exactly* the same evaluation of expressions and result

But in ML, there is a type-system difference:

- **x** on the left can have a polymorphic type, but not on the right
- Can always go from right to left
- If **x** need not be polymorphic, can go from left to right

What about performance?

According to our definition of equivalence, these two functions are equivalent, but we learned one is awful

- (Actually we studied this before pattern-matching)

```
fun max xs =
  case xs of
    [] => raise Empty
  | x::[] => x
  | x::xs' =>
      if x > max xs'
      then x
      else max xs'
```

```
fun max xs =
  case xs of
    [] => raise Empty
  | x::[] => x
  | x::xs' =>
      let
        val y = max xs'
      in
        if x > y
        then x
        else y
      end
```

Different definitions for different jobs

- PL Equivalence (341): given same inputs, same outputs and effects
  - Good: Lets us replace bad **`max`** with good **`max`**
  - Bad: Ignores performance in the extreme
- Asymptotic equivalence (332): Ignore constant factors
  - Good: Focus on the algorithm and efficiency for large inputs
  - Bad: Ignores "four times faster"
- Systems equivalence (333): Account for constant overheads, performance tune
  - Good: Faster means different and better
  - Bad: Beware overtuning on "wrong" (e.g., small) inputs; definition does not let you "swap in a different algorithm"

Claim: Computer scientists implicitly (?) use all three every (?) day

PAUL G. ALLEN SCHOOL
OF COMPUTER SCIENCE & ENGINEERING

CSE341: Programming Languages

Interlude: Course Motivation

Dan Grossman

Spring 2019

Course Motivation
(Did you think I forgot? ☺)

- Why learn the fundamental concepts that appear in all (most?) languages?
- Why use languages quite different from C, C++, Java, Python?
- Why focus on functional programming?
- Why use ML, Racket, and Ruby in particular?
- Not: Language X is better than Language Y

[You won't be tested on this stuff]

Summary

- No such thing as a “best” PL
- Fundamental concepts easier to teach in some (multiple) PLs
- A good PL is a relevant, elegant interface for writing software
  - There is no substitute for precise understanding of PL semantics
- Functional languages have been on the leading edge for decades
  - Ideas have been absorbed by the mainstream, but very slowly
  - First-class functions and avoiding mutation increasingly essential
  - Meanwhile, use the ideas to be a better C/Java/PHP hacker
- Many great alternatives to ML, Racket, and Ruby, but each was chosen for a reason and for how they complement each other

What is the best kind of car?

What is the best kind of shoes?

Cars / Shoes

Cars are used for rather different things:

- Winning a Formula 1 race
- 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

More on cars

- A good mechanic might have a specialty, but also understands how “cars” (not a particular make/model) work
  - The upholstery color isn’t essential (syntax)

- A good mechanical engineer really knows how cars work, how to get the most out of them, and how to design better ones
  - I don’t have a favorite kind of car or a favorite PL

- To learn how car pieces interact, it may make sense to start with a classic design rather than the latest model
  - A popular car may not be best
  - May especially not be best for learning how cars work

Why semantics and idioms

This course focuses as much as it can on semantics and idioms

- 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
  - See Java in a clearer light even if I never show you Java

Hamlet

The play *Hamlet*:

- Is a beautiful work of art
- Teaches deep, eternal truths
- Is the source of some well-known sayings
- Makes you a better person

Continues to be studied centuries later even though:

- The syntax is really annoying to many
- There are more popular movies with some of the same lessons
- Reading Hamlet will not get you a summer internship

All cars are the same

- To make it easier to rent cars, it is great that they all have steering wheels, brakes, windows, headlights, etc.
  - Yet it is still uncomfortable to learn a new one
  - Can you be a great driver if you only ever drive one car?

- And maybe PLs are more like cars, trucks, boats, and bikes

- So are all PLs really the same…

Are all languages the same?

Yes:

- Any input-output behavior implementable in language X is implementable in language Y [Church-Turing thesis]
- 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:

- The human condition vs. different cultures
  (travel to learn more about home)
- The primitive/default in one language is awkward in another
- Beware “the Turing tarpit”

Functional Programming

Why spend 60-80% of course using *functional languages*:

- Mutation is discouraged
- Higher-order functions are very convenient
- One-of types via constructs like datatypes

Because:

1. These features are invaluable for correct, elegant, efficient software (great way to think about computation)
2. Functional languages have always been ahead of their time
3. Functional languages well-suited to where computing is going

Most of course is on (1), so a few minutes on (2) and (3) …

Ahead of their time

All these were dismissed as “beautiful, worthless, slow things PL professors make you learn”

- Garbage collection (Java didn’t exist in 1995, PL courses did)
- Generics (**`List<T>`** in Java, C#), much more like SML than C++
- XML for universal data representation (like Racket/Scheme/LISP/…)
- Higher-order functions (Ruby, Javascript, C#, now Java, …)
- Type inference (C#, Scala, …)
- Recursion (a big fight in 1960 about this – I’m told ☺)
- …

The future may resemble the past

Somehow nobody notices we are right… 20 years later

- "To conquer" versus "to assimilate"
- Societal progress takes time and muddles "taking credit"
- Maybe pattern-matching, currying, hygienic macros, etc. will be next

Recent-ish Surge, Part 1

Other popular functional PLs (alphabetized, pardon omissions)

- Clojure http://clojure.org
- Erlang http://www.erlang.org
- F# http://tryfsharp.org
- Haskell http://www.haskell.org
- OCaml http://ocaml.org
- Scala http://www.scala-lang.org

Some “industry users” lists (surely more exist):

- http://www.haskell.org/haskellwiki/Haskell_in_industry
- http://ocaml.org/companies.html
- In general, see http://cufp.org

Recent-ish Surge, Part 2

Popular adoption of concepts:

- C#, LINQ (closures, type inference, …)
- Java 8 (closures)
- MapReduce / Hadoop
  - Avoiding side-effects essential for fault-tolerance here
- Scala libraries (e.g., Akka, …)
- …

Why a surge?

My best *guesses*:

- Concise, elegant, productive programming
- JavaScript, Python, Ruby helped break the Java/C/C++ hegemony
- Avoiding mutation is *the* easiest way to make concurrent and parallel programming easier
  - In general, to handle sharing in complex systems
- Sure, functional programming is still a small niche, but there is so much software in the world today even niches have room

The languages together

SML, Racket, and Ruby are a useful *combination* for us

	dynamically typed	statically typed
functional	Racket	SML
object-oriented	Ruby	Java

ML: polymorphic types, pattern-matching, abstract types & modules

Racket: dynamic typing, “good” macros, minimalist syntax, eval

Ruby: classes but not types, very OOP, mixins

[and much more]

Really wish we had more time:

Haskell: laziness, purity, type classes, monads

Prolog: unification and backtracking

[and much more]

But why not…

Instead of SML, could use similar languages easy to learn after:

- OCaml: yes indeed but would have to port all my materials 😐
  - And a few small things (e.g., second-class constructors)
- F#: yes and very cool, but needs a .Net platform
  - And a few more small things (e.g., second-class constructors, less elegant signature-matching)
- Haskell: more popular, cooler types, but lazy semantics and type classes from day 1

Admittedly, SML and its implementations are showing their age (e.g., **`andalso`** and less tool support), but it still makes for a fine foundation in statically typed, eager functional programming

But why not…

Instead of Racket, could use similar languages easy to learn after:

- Scheme, Lisp, Clojure, …

Racket has a combination of:

- A modern feel and active evolution
- "Better" macros, modules, structs, contracts, …
- A large user base and community (*not* just for education)
- An IDE tailored to education

Could easily define our own language in the Racket system

- Would rather use a good and vetted design

But why not…

Instead of Ruby, could use another language:

- Python, Perl, JavaScript are also dynamically typed, but are not as "fully" OOP, which is what I want to focus on
  - Python also does not have (full) closures
  - JavaScript also does not have classes but is OOP

- Smalltalk serves my OOP needs
  - But implementations merge language/environment
  - Less modern syntax, user base, etc.

Is this real programming?

- The way we use ML/Racket/Ruby can make them seem almost "silly" precisely because lecture and homework focus on interesting language constructs

- "Real" programming needs file I/O, string operations, floating-point, graphics, project managers, testing frameworks, threads, build systems, …
  - Many elegant languages have all that and more
    - Including Racket and Ruby
  - If we used Java the same way, Java would seem "silly" too

A note on reality

Reasonable questions when deciding to use/learn a language:

- What libraries are available for reuse?
- What tools are available?
- What can get me a job?
- What does my boss tell me to do?
- What is the de facto industry standard?
- What do I already know?

Our course by design does not deal with these questions

  - You have the rest of your life for that
  - And technology *leaders* affect the answers

PAUL G. ALLEN SCHOOL
OF COMPUTER SCIENCE & ENGINEERING

CSE341: Programming Languages

Lecture 13
Racket Introduction

Dan Grossman

Spring 2019

Racket

Next two units will use the Racket language (not ML) and the DrRacket programming environment (not Emacs)

- Installation / basic usage instructions on course website

- Like ML, functional focus with imperative features
  - Anonymous functions, closures, no return statement, etc.
  - But we will not use pattern-matching

- Unlike ML, no static type system: accepts more programs, but most errors do not occur until run-time

- Really minimalist syntax

- Advanced features like macros, modules, quoting/eval, continuations, contracts, …
  - Will do only a couple of these

Racket vs. Scheme

- Scheme and Racket are very similar languages
  - Racket “changed its name” in 2010
- Racket made some non-backward-compatible changes…
  - How the empty list is written
  - Cons cells not mutable
  - How modules work
  - Etc.

  … and many additions
- Result: A modern language used to build some real systems
  - More of a moving target: notes may become outdated
  - Online documentation, particularly “The Racket Guide”

Getting started

DrRacket “definitions window” and “interactions window” very similar to how we used Emacs and a REPL, but more user-friendly

- DrRacket has always focused on good-for-teaching
- See usage notes for how to use REPL, testing files, etc.
- Easy to learn to use on your own, but lecture demos will help

Free, well-written documentation:

- http://racket-lang.org/
- The Racket Guide especially, http://docs.racket-lang.org/guide/index.html

File structure

Start every file with a line containing only

```
#lang racket
```

(Can have comments before this, but not code)

A file is a module containing a *collection of definitions* (bindings)…

Example

```
#lang racket

(define x 3)
(define y (+ x 2))

(define cube ; function
  (lambda (x)
    (* x (* x x))))

(define pow ; recursive function
  (lambda (x y)
    (if (= y 0)
        1
        (* x (pow x (- y 1))))))
```

Some niceties

Many built-in functions (a.k.a. procedures) take any number of args

- Yes `*` is just a function
- Yes you can define your own *variable-arity* functions (not shown here)

```
(define cube
  (lambda (x)
    (* x x x)))
```

Better style for non-anonymous function definitions (just sugar):

```
(define (cube x)
  (* x x x))

(define (pow x y)
    (if (= y 0)
        1
        (* x (pow x (- y 1)))))
```

An old friend: currying

Currying is an idiom that works in any language with closures

- Less common in Racket because it has real multiple args

```
(define pow
  (lambda (x)
    (lambda (y)
      (if (= y 0)
          1
          (* x ((pow x) (- y 1)))))))

(define three-to-the (pow 3))
(define eightyone (three-to-the 4))
(define sixteen ((pow 2) 4))
```

Sugar for defining curried functions: `(define ((pow x) y) (if …`

(No sugar for calling curried functions)

Another old-friend: List processing

Empty list: **`null`**

Cons constructor: **`cons`**

Access head of list: **`car`**

Access tail of list: **`cdr`**

Check for empty: **`null?`**

Notes:

- Unlike Scheme, **`()`** doesn't work for **`null`**, but **`'()`** does
- **`(list e1 … en)`** for building lists
- Names **`car`** and **`cdr`** are a historical accident

Examples

```
(define (sum xs)
  (if (null? xs)
      0
      (+ (car xs) (sum (cdr xs)))))

(define (my-append xs ys)
  (if (null? xs)
      ys
      (cons (car xs) (my-append (cdr xs) ys))))

(define (my-map f xs)
  (if (null? xs)
      null
      (cons (f (car xs)) (my-map f (cdr xs)))))
```

Racket syntax

Ignoring a few “bells and whistles,”

Racket has an amazingly simple *syntax*

A *term* (anything in the language) is either:

- An *atom*, e.g., **`#t`**, **`#f`**, **`34`**, **`"hi"`**, **`null`**, **`4.0`**, **`x`**, …
- A *special form*, e.g., **`define`**, **`lambda`**, **`if`**
  - Macros will let us define our own
- A *sequence* of terms in parens: **`(t1 t2 … tn)`**
  - If **`t1`** a special form, semantics of sequence is special
  - Else a function call

- Example: **`(+ 3 (car xs))`**
- Example: **`(lambda (x) (if x "hi" #t))`**

Brackets

Minor note:

Can use **[** anywhere you use **(**, but must match with **]**

- Will see shortly places where **[…]** is common style
- DrRacket lets you type **)** and replaces it with **]** to match

Why is this good?

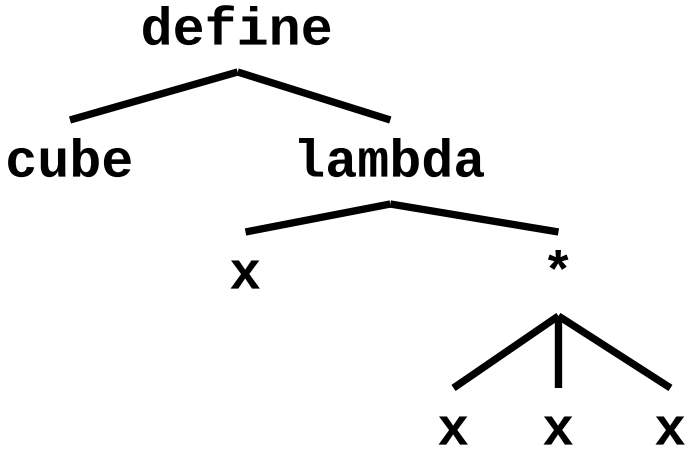
By parenthesizing everything, converting the program text into a tree representing the program (*parsing*) is trivial and unambiguous

- Atoms are leaves
- Sequences are nodes with elements as children
- (No other rules)

Also makes indentation easy

Example:

```
(define cube
  (lambda (x)
    (* x x x)))
```

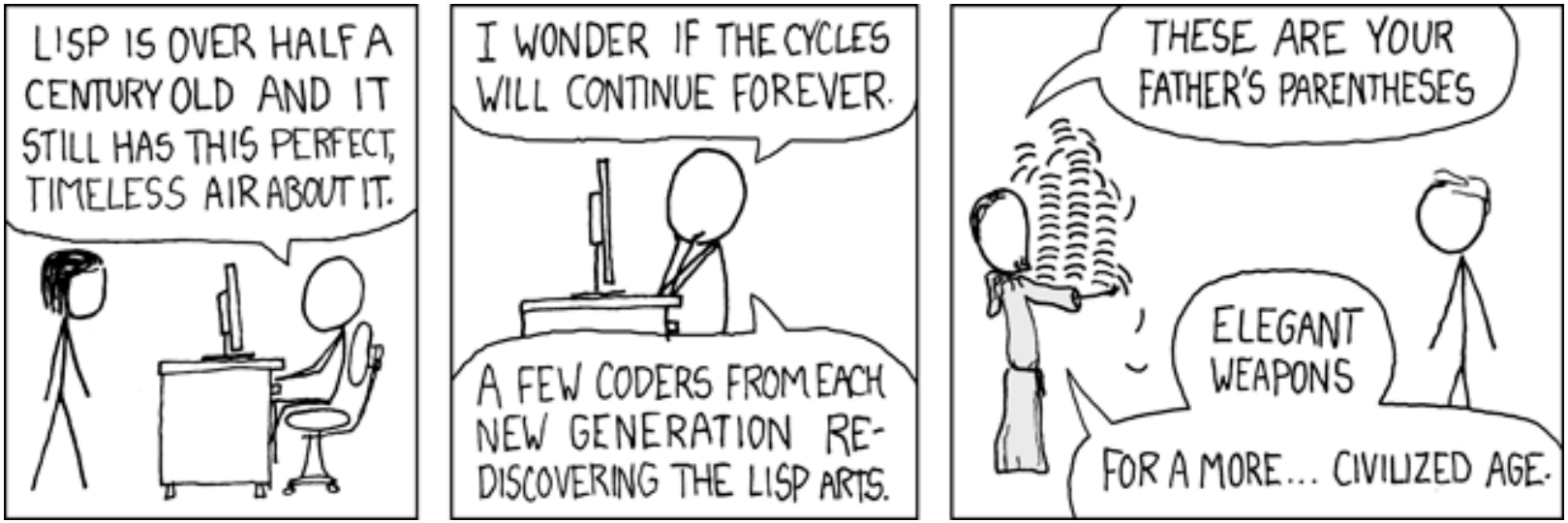


No need to discuss “operator precedence” (e.g., `x + y * z`)

Parenthesis bias

- If you look at the HTML for a web page, it takes the same approach:
  - **`(foo`** written **`<foo>`**
  - **`)`** written **`</foo>`**

- But for some reason, LISP/Scheme/Racket is the target of subjective parenthesis-bashing
  - Bizarrely, often by people who have no problem with HTML
  - You are entitled to your opinion about syntax, but a good historian wouldn't refuse to study a country where he/she didn't like people's accents



http://xkcd.com/297/

Parentheses matter

You must break yourself of one habit for Racket:

- Do not add/remove parens because you feel like it
  - Parens are never optional or meaningless!!!
- In most places **(e)** means call **e** with zero arguments
- So **((e))** means call **e** with zero arguments and call the result with zero arguments

Without static typing, often get hard-to-diagnose run-time errors

Examples (more in code)

Correct:

```
(define (fact n)(if (= n 0) 1 (* n (fact (- n 1)))))
```

Treats 1 as a zero-argument function (run-time error):

```
(define (fact n)(if (= n 0) (1)(* n (fact (- n 1)))))
```

Gives **`if`** 5 arguments (syntax error)

```
(define (fact n)(if = n 0 1 (* n (fact (- n 1)))))
```

3 arguments to define (including **`(n)`**) (syntax error)

```
(define fact (n)(if (= n 0) 1 (* n (fact (- n 1)))))
```

Treats **`n`** as a function, passing it **`*`** (run-time error)

```
(define (fact n)(if (= n 0) 1 (n * (fact (- n 1)))))
```

Dynamic typing

Major topic coming later: contrasting static typing (e.g., ML) with dynamic typing (e.g., Racket)

For now:

- Frustrating not to catch "little errors" like `(n * x)` until you test your function
- But can use very flexible data structures and code without convincing a type checker that it makes sense

Example:

- A list that can contain numbers or other lists
- Assuming *lists or numbers "all the way down,"* sum all the numbers…

Example

```
(define (sum xs)
  (if (null? xs)
      0
      (if (number? (car xs))
          (+ (car xs) (sum (cdr xs)))
          (+ (sum (car xs)) (sum (cdr xs))))))
```

- No need for a fancy datatype binding, constructors, etc.
- Works no matter how deep the lists go
- But assumes each element is a list or a number
  - Will get a run-time error if anything else is encountered

Better style

Avoid nested if-expressions when you can use cond-expressions instead

- Can think of one as sugar for the other

General syntax:
```
(cond [e1a e1b]
      [e2a e2b]
       …
      [eNa eNb])
```

- Good style: **`eNa`** should be **`#t`**

Example

```
(define (sum xs)
  (cond [(null? xs) 0]
        [(number? (car xs))
         (+ (car xs) (sum (cdr xs)))]
        [#t (+ (sum (car xs)) (sum (cdr xs)))]))
```

A variation

As before, we could change our spec to say instead of errors on non-numbers, we should just ignore them

So this version can work for any list (or just a number)

- Compare carefully, we did *not* just add a branch

```
(define (sum xs)
  (cond [(null? xs) 0]
        [(number? xs) xs]
        [(list? xs)
         (+ (sum (car xs)) (sum (cdr xs)))]
        [#t 0]))
```

What is true?

For both **`if`** and **`cond`**, test expression can evaluate to anything

- It is not an error if the result is not **`#t`** or **`#f`**
- (Apologies for the double-negative ☺)

Semantics of **`if`** and **`cond`**:

- "Treat anything other than **`#f`** as true"
- (In some languages, other things are false, not in Racket)

This feature makes no sense in a statically typed language

Some consider using this feature poor style, but it can be convenient

Local bindings

- Racket has 4 ways to define local variables
  - **`let`**
  - **`let*`**
  - **`letrec`**
  - **`define`**

- Variety is good: They have different semantics
  - Use the one most convenient for your needs, which helps communicate your intent to people reading your code
    - If any will work, use **`let`**
  - Will help us better learn scope and environments

- Like in ML, the 3 kinds of let-expressions can appear anywhere

Let

A let expression can bind any number of local variables

- Notice where all the parentheses are

The expressions are all evaluated in the environment from **before the let-expression**

- Except the body can use all the local variables of course
- This is **not** how ML let-expressions work
- Convenient for things like **`(let ([x y][y x]) …)`**

```
(define (silly-double x)
  (let ([x (+ x 3)]
        [y (+ x 2)])
    (+ x y -5)))
```

*Let**

Syntactically, a let* expression is a let-expression with 1 more character

The expressions are evaluated in the environment produced from the **previous bindings**

- Can repeat bindings (later ones shadow)
- This **is** how ML let-expressions work

```
(define (silly-double x)
  (let* ([x (+ x 3)]
         [y (+ x 2)])
    (+ x y -8)))
```

Letrec

Syntactically, a letrec expression is also the same

The expressions are evaluated in the environment that includes **all the bindings**

```
(define (silly-triple x)
  (letrec ([y (+ x 2)]
           [f (lambda(z) (+ z y w x))]
           [w (+ x 7)])
    (f -9)))
```

- Needed for mutual recursion
- But expressions are still *evaluated in order*: accessing an uninitialized binding raises an error
  - Remember function bodies not evaluated until called

More letrec

- Letrec is ideal for recursion (including mutual recursion)

```
(define (silly-mod2 x)
  (letrec
   ([even? (λ(x)(if (zero? x) #t (odd? (- x 1))))]
    [odd?  (λ(x)(if (zero? x) #f (even? (- x 1))))])
     (if (even? x) 0 1)))
```

- Do not use later bindings except inside functions
  - This example will raise an error when called

```
(define (bad-letrec x)
  (letrec ([y z]
           [z 13])
    (if x y z)))
```

Local defines

- In certain positions, like the beginning of function bodies, you can put defines
  - For defining local variables, same semantics as **`letrec`**

```
(define (silly-mod2 x)
  (define (even? x)(if (zero? x) #t (odd? (- x 1))))
  (define (odd? x) (if (zero? x) #f (even?(- x 1))))
  (if (even? x) 0 1))
```

- Local defines is preferred Racket style, but course materials will avoid them to emphasize **`let`**, **`let*`**, **`letrec`** distinction
  - You can choose to use them on homework or not

Top-level

The bindings in a file work like local defines, i.e., **`letrec`**

- Like ML, you can *refer to* earlier bindings
- Unlike ML, you can also *refer to* later bindings
- But refer to later bindings only in function bodies
  - Because bindings are *evaluated* in order
  - Get an error if try to use a not-yet-defined binding
- Unlike ML, cannot define the same variable twice in module
  - Would make no sense: cannot have both in environment

REPL

Unfortunate detail:

- REPL works slightly differently
  - Not quite **`let*`** or **`letrec`**
  - ☹
- Best to avoid recursive function definitions or forward references in REPL
  - Actually okay unless shadowing something (you may not know about) – then weirdness ensues
  - And calling recursive functions is fine of course

Optional: Actually…

- Racket has a module system
  - Each file is implicitly a module
    - Not really “top-level”
  - A module can shadow bindings from other modules it uses
    - Including Racket standard library
  - So we could redefine **+** or any other function
    - But poor style
    - Only shadows in our module (else messes up rest of standard library)

- (Optional note: Scheme is different)

Set!

- Unlike ML, Racket really has assignment statements
  - But used *only-when-really-appropriate!*

```
(set! x e)
```

- For the **x** in the current environment, subsequent lookups of **x** get the result of evaluating expression **e**
  - Any code using this **x** will be affected
  - Like **`x = e`** in Java, C, Python, etc.

- Once you have side-effects, sequences are useful:

```
(begin e1 e2 … en)
```

Example

Example uses **`set!`** at top-level; mutating local variables is similar

```
(define b 3)
(define f (lambda (x) (* 1 (+ x b))))
(define c (+ b 4)) ; 7
(set! b 5)
(define z (f 4))   ; 9
(define w c)       ; 7
```

Not much new here:

- Environment for closure determined when function is defined, but body is evaluated when function is called
- Once an expression produces a value, it is irrelevant how the value was produced

Top-level

- Mutating top-level definitions is particularly problematic
  - What if any code could do **`set!`** on anything?
  - How could we defend against this?

- A general principle: If something you need not to change might change, make a local copy of it. Example:

```
(define b 3)
(define f
  (let ([b b])
    (lambda (x) (* 1 (+ x b)))))
```

Could use a different name for local copy but do not need to

But wait…

- Simple elegant language design:
  - Primitives like **+** and ***** are just predefined variables bound to functions
  - But maybe that means they are mutable
  - Example continued:

```
(define f
  (let ([b b]
        [+ +]
        [* *])
    (lambda (x) (* 1 (+ x b)))))
```

  - Even that won't work if **f** uses other functions that use things that might get mutated – all functions would need to copy everything mutable they used

No such madness

In Racket, *you do not have to program like this*

- Each file is a module
- *If* a module does not use **`set!`** on a top-level variable, then Racket makes it constant and forbids **`set!`** outside the module
- Primitives like **+**, *****, and **`cons`** are in a module that does not mutate them

Showed you this for the *concept* of copying to defend against mutation

- Easier defense: Do not allow mutation
- Mutable top-level bindings a highly dubious idea

The truth about **cons**

cons just makes a pair

- Often called a *cons cell*
- By convention and standard library, lists are nested pairs that eventually end with **null**

```
(define pr (cons 1 (cons #t "hi"))) ; '(1 #t . "hi")
(define lst (cons 1 (cons #t (cons "hi" null))))
(define hi (cdr (cdr pr)))
(define hi-again (car (cdr (cdr lst))))
(define hi-another (caddr lst))
(define no (list? pr))
(define yes (pair? pr))
(define of-course (and (list? lst) (pair? lst)))
```

Passing an *improper list* to functions like **length** is a run-time error

The truth about **`cons`**

So why allow improper lists?

- Pairs are useful
- Without static types, why distinguish **`(e1,e2)`** and **`e1::e2`**

Style:

- Use proper lists for collections of unknown size
- But feel free to use **`cons`** to build a pair
  - Though structs (like records) may be better

Built-in primitives:

- **`list?`** returns true for proper lists, including the empty list
- **`pair?`** returns true for things made by cons
  - All improper and proper lists except the empty list

cons cells are immutable

What if you wanted to mutate the *contents* of a cons cell?

- In Racket you cannot (major change from Scheme)
- This is good
  - List-aliasing irrelevant
  - Implementation can make **`list?`** fast since listness is determined when cons cell is created

Set! does not change list contents

This does *not* mutate the contents of a cons cell:

```
(define x (cons 14 null))
(define y x)
(set! x (cons 42 null))
(define fourteen (car y))
```

- Like Java's **`x = new Cons(42,null)`**, *not* **`x.car = 42`**

mcons cells are mutable

Since mutable pairs are sometimes useful (will use them soon), Racket provides them too:

- **`mcons`**
- **`mcar`**
- **`mcdr`**
- **`mpair?`**
- **`set-mcar!`**
- **`set-mcdr!`**

Run-time error to use **`mcar`** on a cons cell or **`car`** on an mcons cell

PAUL G. ALLEN SCHOOL
OF COMPUTER SCIENCE & ENGINEERING

CSE341: Programming Languages

Lecture 14
Thunks, Laziness, Streams, Memoization

Dan Grossman

Spring 2019

Delayed evaluation

For each language construct, the semantics specifies when subexpressions get evaluated. In ML, Racket, Java, C:

- Function arguments are *eager* (call-by-value)
  - Evaluated once before calling the function
- Conditional branches are not eager

It matters: calling **`factorial-bad`** never terminates:

```
(define (my-if-bad x y z)
  (if x y z))

(define (factorial-bad n)
  (my-if-bad (= n 0)
             1
             (* n (factorial-bad (- n 1)))))
```

Thunks delay

We know how to delay evaluation: put expression in a function!

- Thanks to closures, can use all the same variables later

A zero-argument function used to delay evaluation is called a *thunk*

- As a verb: *thunk the expression*

This works (but it is silly to wrap **`if`** like this):

```
(define (my-if x y z)
  (if x (y) (z)))

(define (fact n)
    (my-if (= n 0)
           (lambda() 1)
           (lambda() (* n (fact (- n 1))))))
```

The key point

- Evaluate an expression **e** to get a result:

```
e
```

- A function that *when called*, evaluates **e** and returns result
  - Zero-argument function for "thunking"

```
(lambda () e)
```

- Evaluate **e** to some thunk and then call the thunk

```
(e)
```

- Next: Powerful idioms related to delaying evaluation and/or avoided repeated or unnecessary computations
  - Some idioms also use mutation in encapsulated ways

Avoiding expensive computations

Thunks let you skip expensive computations if they are not needed

Great if take the true-branch:

```
(define (f th)
  (if (…) 0 (…  (th) …)))
```

But worse if you end up using the thunk more than once:

```
(define (f th)
  (… (if (…) 0 (… (th) …))
     (if (…) 0 (… (th) …))
     …
     (if (…) 0 (… (th) …))))
```

In general, might not know many times a result is needed

Best of both worlds

Assuming some expensive computation has no side effects, ideally we would:

- Not compute it *until needed*
- *Remember the answer* so future uses complete immediately

Called *lazy evaluation*

Languages where most constructs, including function arguments, work this way are *lazy languages*

- Haskell

Racket predefines support for *promises*, but we can make our own

- Thunks and mutable pairs are enough

Delay and force

```
(define (my-delay th)
  (mcons #f th))

(define (my-force p)
  (if (mcar p)
      (mcdr p)
      (begin (set-mcar! p #t)
             (set-mcdr! p ((mcdr p)))
             (mcdr p))))
```

An ADT represented by a mutable pair

- `#f` in *car* means *cdr* is unevaluated thunk
  - Really a one-of type: thunk or result-of-thunk
- Ideally hide representation in a module

Using promises

```
(define (f p)
  (… (if (…) 0 (… (my-force p) …))
     (if (…) 0 (… (my-force p) …))
     …
     (if (…) 0 (… (my-force p) …))))
```

```
(f (my-delay (lambda () e)))
```

Lessons From Example

See code file for example that does multiplication using a very slow addition helper function

- With thunking second argument:
  - *Great* if first argument 0
  - Okay if first argument 1
  - *Worse* otherwise
- With precomputing second argument:
  - *Okay* in all cases
- With thunk that uses a promise for second argument:
  - *Great* if first argument 0
  - *Okay* otherwise

Streams

- A stream is an *infinite sequence* of values
  - So cannot make a stream by making all the values
  - Key idea: Use a thunk to delay creating most of the sequence
  - Just a programming idiom

A powerful concept for division of labor:

- Stream producer knows how to create any number of values
- Stream consumer decides how many values to ask for

Some examples of streams you might (not) be familiar with:

- User actions (mouse clicks, etc.)
- UNIX pipes: **`cmd1 | cmd2`** has **`cmd2`** "pull" data from **`cmd1`**
- Output values from a sequential feedback circuit

Using streams

We will represent streams using pairs and thunks

Let a stream be a thunk that *when called* returns a pair:

```
'(next-answer . next-thunk)
```

So given a stream **`s`**, the client can get any number of elements

- First: **`(car (s))`**
- Second: **`(car ((cdr (s))))`**
- Third: **`(car ((cdr ((cdr (s))))))`**

(Usually bind **`(cdr (s))`** to a variable or pass to a recursive function)

Example using streams

This function returns how many stream elements it takes to find one for which tester does not return **#f**

- Happens to be written with a tail-recursive helper function

```
(define (number-until stream tester)
  (letrec ([f (lambda (stream ans)
                (let ([pr (stream)])
                   (if (tester (car pr))
                       ans
                       (f (cdr pr) (+ ans 1)))))])
    (f stream 1)))
```

- **(stream)** generates the pair
- So recursively pass **(cdr pr)**, the thunk for the rest of the infinite sequence

Streams

Coding up a stream in your program is easy

- We will do functional streams using pairs and thunks

Let a stream be a thunk that *when called* returns a pair:

```
'(next-answer . next-thunk)
```

Saw how to use them, now how to make them…

- Admittedly mind-bending, but uses what we know

Making streams

- How can one thunk create the right next thunk? Recursion!
  - Make a thunk that produces a pair where cdr is next thunk
  - A recursive function can return a thunk where recursive call does not happen until thunk is called

```
(define ones (lambda () (cons 1 ones)))

(define nats
  (letrec ([f (lambda (x)
                (cons x (lambda () (f (+ x 1)))))])
    (lambda () (f 1))))

(define powers-of-two
  (letrec ([f (lambda (x)
                (cons x (lambda () (f (* x 2)))))])
    (lambda () (f 2))))
```

Getting it wrong

- This uses a variable before it is defined

```
(define ones-really-bad (cons 1 ones-really-bad))
```

- This goes into an infinite loop making an infinite-length list

```
(define ones-bad (lambda () cons 1 (ones-bad)))
(define (ones-bad)(cons 1 (ones-bad)))
```

- This is a stream: thunk that returns a pair with cdr a thunk

```
(define ones (lambda () (cons 1 ones)))
(define (ones)(cons 1 ones))
```

Memoization

- If a function has no side effects and does not read mutable memory, no point in computing it twice for the same arguments
  - Can keep a *cache* of previous results
  - Net win if (1) maintaining cache is cheaper than recomputing and (2) cached results are reused

- Similar to promises, but if the function takes arguments, then there are multiple "previous results"

- For recursive functions, this *memoization* can lead to *exponentially* faster programs
  - Related to algorithmic technique of dynamic programming

How to do memoization: see example

- Need a (mutable) cache that all calls using the cache share
  - So must be defined *outside* the function(s) using it

- See code for an example with Fibonacci numbers

  - Good demonstration of the idea because it is short, but, as shown in the code, there are also easier less-general ways to make **`fibonacci`** efficient

  - (An association list (list of pairs) is a simple but sub-optimal data structure for a cache; okay for our example)

assoc

- Example uses **`assoc`**, which is just a library function you could look up in the Racket reference manual:

  `(assoc v lst)` takes a list of pairs and locates the first element of **`lst`** whose car is equal to **`v`** according to **`is-equal?`**. If such an element exists, the pair (i.e., an element of **`lst`**) is returned. Otherwise, the result is **`#f`**.

- Returns **`#f`** for not found to distinguish from finding a pair with **`#f`** in cdr

PAUL G. ALLEN SCHOOL
OF COMPUTER SCIENCE & ENGINEERING

CSE341: Programming Languages

Lecture 15
Macros

Dan Grossman

Spring 2019

What is a macro

- A *macro definition* describes how to transform some new syntax into different syntax in the source language
- A macro is one way to implement syntactic sugar
  - "Replace any syntax of the form **`e1 andalso e2`** with **`if e1 then e2 else false`**"
- A *macro system* is a language (or part of a larger language) for defining macros
- *Macro expansion* is the process of rewriting the syntax for each *macro use*
  - Before a program is run (or even compiled)

Using Racket Macros

- If you define a macro **`m`** in Racket, then **`m`** becomes a new special form:
  - Use **`(m …)`** gets expanded according to definition

- Example definitions (actual definitions coming later):
  - Expand **`(my-if e1 then e2 else e3)`**
    to **`(if e1 e2 e3)`**
  - Expand **`(comment-out e1 e2)`**
    to **`e2`**
  - Expand **`(my-delay e)`**
    to **`(mcons #f (lambda () e))`**

Example uses

It is like we added keywords to our language

- Other keywords only keywords in uses of that macro
- Syntax error if keywords misused
- Rewriting ("expansion") happens before execution

```
(my-if x then y else z)  ; (if x y z)
(my-if x then y then z)  ; syntax error

(comment-out (car null) #f)

(my-delay (begin (print "hi") (foo 15)))
```

Overuse

Macros often deserve a bad reputation because they are often overused or used when functions would be better

When in doubt, resist defining a macro?

But they can be used well

Now…

- How any macro system must deal with tokens, parentheses, and scope
- How to define macros in Racket
- How macro definitions must deal with expression evaluation carefully
  - Order expressions evaluate and how many times
- The key issue of variable bindings in macros and the notion of *hygiene*
  - Racket is superior to most languages here

Tokenization

First question for a macro system: How does it tokenize?

- Macro systems generally work at the level of *tokens* not sequences of characters
  - So must know how programming language tokenizes text

- Example: "macro expand **`head`** to **`car`**"
  - Would not rewrite **`(+ headt foo)`** to **`(+ cart foo)`**
  - Would not rewrite **`head-door`** to **`car-door`**
    - But would in C where **`head-door`** is subtraction

Parenthesization

Second question for a macro system: How does associativity work?

C/C++ basic example:

```
#define ADD(x,y) x+y
```

Probably *not* what you wanted:

`ADD(1,2/3)*4` means `1+2/3*4` not `(1+2/3)*4`

So C macro writers use lots of parentheses, which is fine:

```
#define ADD(x,y) ((x)+(y))
```

Racket won't have this problem:

- Macro use: `(macro-name …)`
- After expansion: ***`something else in same place`***

Local bindings

Third question for a macro system: Can variables shadow macros?

Suppose macros also apply to variable bindings. Then:

```
(let  ([head 0][car 1]) head) ; 0
(let* ([head 0][car 1]) head) ; 0
```

Would become:

```
(let  ([car 0][car 1]) car) ; error
(let* ([car 0][car 1]) car) ; 1
```

This is why C/C++ convention is all-caps macros and non-all-caps for everything else

Racket does *not* work this way – it gets scope "right"!

Example Racket macro definitions

Two simple macros

```
(define-syntax my-if                    ; macro name
  (syntax-rules (then else)             ; other keywords
    [(my-if e1 then e2 else e3) ; macro use
     (if e1 e2 e3)]))                   ; form of expansion
```

```
(define-syntax comment-out              ; macro name
  (syntax-rules ()                      ; other keywords
    [(comment-out ignore instead) ; macro use
     instead]))                         ; form of expansion
```

If the form of the use matches, do the corresponding expansion

- In these examples, list of possible use forms has length 1
- Else syntax error

Revisiting delay and force

Recall our definition of promises from earlier

- Should we use a macro instead to avoid clients' explicit thunk?

```
(define (my-delay th)
  (mcons #f th))

(define (my-force p)
  (if (mcar p)
      (mcdr p)
      (begin (set-mcar! p #t)
             (set-mcdr! p ((mcdr p)))
             (mcdr p))))
```

```
(f (my-delay (lambda () e)))
```

```
(define (f p)
  (… (my-force p) …))
```

A delay macro

- A macro can put an expression under a thunk
  - Delays evaluation without explicit thunk
  - Cannot implement this with a function
- Now client should *no*t use a thunk (that would double-thunk)
  - Racket's pre-defined **`delay`** is a similar macro

```
(define-syntax my-delay
  (syntax-rules ()
    [(my-delay e)
     (mcons #f (lambda() e))]))
```

```
(f (my-delay e))
```

What about a force macro?

We could define **`my-force`** with a macro too

- Good macro style would be to evaluate the argument exactly once (use **`x`** below, not multiple evaluations of **`e`**)
- Which shows it is bad style to use a macro at all here!
- Do not use macros when functions do what you want

```
(define-syntax my-force
  (syntax-rules ()
    [(my-force e)
     (let([x e])
        (if (mcar x)
            (mcdr x)
            (begin (set-mcar! x #t)
                   (set-mcdr! p ((mcdr p)))
                   (mcdr p))))]))
```

Another bad macro

Any *function* that doubles its argument is fine for clients

```
(define (dbl x) (+ x x))
(define (dbl x) (* 2 x))
```

- These are equivalent to each other

So macros for doubling are bad style but instructive examples:

```
(define-syntax dbl (syntax-rules()[(dbl x)(+ x x)]))
(define-syntax dbl (syntax-rules()[(dbl x)(* 2 x)]))
```

- These are not equivalent to each other. Consider:

```
(dbl (begin (print "hi") 42))
```

More examples

Sometimes a macro *should* re-evaluate an argument it is passed

- If not, as in **`dbl`**, then use a local binding as needed:

```
(define-syntax dbl
  (syntax-rules ()
     [(dbl x)
      (let ([y x]) (+ y y))]))
```

Also good style for macros not to have surprising evaluation order

- Good rule of thumb to preserve left-to-right
- Bad example (fix with a local binding):

```
(define-syntax take
  (syntax-rules (from)
     [(take e1 from e2)
      (- e2 e1)]))
```

Local variables in macros

In C/C++, defining local variables inside macros is unwise

- When needed done with hacks like **`__strange_name34`**

Here is why with a silly example:

- Macro:

```
(define-syntax dbl
  (syntax-rules ()
     [(dbl x) (let ([y 1])
                (* 2 x y))]))
```

- Use:

```
(let ([y 7]) (dbl y))
```

- Naïve expansion:

```
(let ([y 7]) (let ([y 1])
               (* 2 y y)))
```

- But instead Racket "gets it right," which is part of *hygiene*

The other side of hygiene

This also looks like it would do the “wrong” thing

- Macro:

```
(define-syntax dbl
  (syntax-rules ()
     [(dbl x) (* 2 x)]))
```

- Use:

```
(let ([* +]) (dbl 42))
```

- Naïve expansion:

```
(let ([* +]) (* 2 42))
```

- But again Racket’s *hygienic macros* get this right!

How hygienic macros work

A hygienic macro system:

1. Secretly renames local variables in macros with fresh names
2. Looks up variables used in macros where the macro is defined

Neither of these rules are followed by the “naïve expansion” most macro systems use

- Without hygiene, macros are much more brittle (non-modular)

On rare occasions, hygiene is not what you want

- Racket has somewhat complicated support for that

More examples

See the code for macros that:

- A for loop for executing a body a fixed number of times
  - Shows a macro that purposely re-evaluates some expressions and not others

- Allow 0, 1, or 2 local bindings with fewer parens than **`let*`**
  - Shows a macro with multiple cases

- A re-implementation of **`let*`** in terms of **`let`**
  - Shows a macro taking any number of arguments
  - Shows a recursive macro

PAUL G. ALLEN SCHOOL
OF COMPUTER SCIENCE & ENGINEERING

CSE341: Programming Languages

Lecture 16
Datatype-Style Programming With Lists or Structs

Dan Grossman

Spring 2019

The Goal

In ML, we often define datatypes and write recursive functions over them – how do we do analogous things in Racket?

- First way: With lists
- Second way: With structs [a new construct]
  - Contrast helps explain advantages of structs

Life without datatypes

Racket has nothing like a datatype binding for one-of types

No need in a dynamically typed language:

- Can just mix values of different types and use primitives like **`number?`**, **`string?`**, **`pair?`**, etc. to "see what you have"
- Can use cons cells to build up any kind of data

Mixed collections

In ML, cannot have a list of “ints or strings,” so use a datatype:

```
datatype int_or_string = I of int | S of string

fun funny_sum xs = (* int_or_string list -> int *)
   case xs of
        [] => 0
      | (I i)::xs' => i + funny_sum xs'
      | (S s)::xs' => String.size s + funny_sum xs'
```

In Racket, dynamic typing makes this natural without explicit tags

- Instead, every value has a tag with primitives to check it
- So just check car of list with **`number?`** or **`string?`**

Recursive structures

More interesting datatype-programming we know:

```
datatype exp = Const of int
             | Negate of exp
             | Add of exp * exp
             | Multiply of exp * exp
```

```
fun eval_exp e =
   case e of
        Const i => i
      | Negate e2 => ~ (eval_exp e2)
      | Add(e1,e2) => (eval_exp e1) + (eval_exp e2)
      | Multiply(e1,e2)=>(eval_exp e1)*(eval_exp e2)
```

Change how we do this

- Previous version of **`eval_exp`** has type **`exp -> int`**
- From now on will write such functions with type **`exp -> exp`**
- Why? Because will be interpreting languages with multiple kinds of results (ints, pairs, functions, …)
  - Even though much more complicated for example so far
- How? See the ML code file:
  - Base case returns entire expression, e.g., **`(Const 17)`**
  - Recursive cases:
    - Check variant (e.g., make sure a **`Const`**)
    - Extract data (e.g., the number under the **`Const`**)
    - Also return an **`exp`** (e.g., create a new **`Const`**)

New way in Racket

See the Racket code file for coding up the same new kind of "**`exp -> exp`**" *interpreter*

- Using lists where car of list encodes "what kind of exp"

Key points:

- Define our own constructor, test-variant, extract-data functions
  - Just better style than hard-to-read uses of **`car`**, **`cdr`**
- Same recursive structure without pattern-matching
- With no type system, no notion of "what is an exp" except in documentation
  - But if we use the helper functions correctly, then okay
  - Could add more explicit error-checking if desired

Symbols

Will not focus on Racket *symbols* like **`'foo`**, but in brief:

- Syntactically start with quote character
- Like strings, can be almost any character sequence
- Unlike strings, compare two symbols with **eq?** which is fast

New feature

```
(struct foo (bar baz quux) #:transparent)
```

Defines a new kind of thing and introduces several new functions:

- **`(foo e1 e2 e3)`** returns "a foo" with **`bar`**, **`baz`**, **`quux`** fields holding results of evaluating **`e1`**, **`e2`**, and **`e3`**
- **`(foo? e)`** evaluates **`e`** and returns **`#t`** if and only if the result is something that was made with the **`foo`** function
- **`(foo-bar e)`** evaluates **`e`**. If result was made with the **`foo`** function, return the contents of the **`bar`** field, else an error
- **`(foo-baz e)`** evaluates **`e`**. If result was made with the **`foo`** function, return the contents of the **`baz`** field, else an error
- **`(foo-quux e)`** evaluates **`e`**. If result was made with the **`foo`** function, return the contents of the **`quux`** field, else an error

An idiom

```
(struct const (int) #:transparent)
(struct negate (e) #:transparent)
(struct add (e1 e2) #:transparent)
(struct multiply (e1 e2) #:transparent)
```

For “datatypes” like exp, create one struct for each “kind of exp”

- structs are like ML constructors!
- But provide constructor, tester, and extractor functions
  - Instead of patterns
  - E.g., **`const`**, **`const?`**, **`const-int`**
- Dynamic typing means “these are the kinds of exp” is “in comments” rather than a *type system*
- Dynamic typing means “types” of fields are also “in comments”

All we need

These structs are all we need to:

- Build trees representing expressions, e.g.,

```
(multiply (negate (add (const 2) (const 2)))
          (const 7))
```

- Build our **`eval-exp`** function (see code):

```
(define (eval-exp e)
  (cond [(const? e) e]
        [(negate? e)
         (const (- (const-int
                    (eval-exp (negate-e e)))))]
        [(add? e) …]
        [(multiply? e) …]…
```

Attributes

- `#:transparent` is an optional attribute on struct definitions
  - For us, prints struct values in the REPL rather than hiding them, which is convenient for debugging homework

- `#:mutable` is another optional attribute on struct definitions
  - Provides more functions, for example:

```
(struct card (suit rank) #:transparent #:mutable)
; also defines set-card-suit!, set-card-rank!
```

  - Can decide if each struct supports mutation, with usual advantages and disadvantages
    - As expected, we will avoid this attribute
  - **`mcons`** is just a predefined mutable struct

Contrasting Approaches

```
(struct add (e1 e2) #:transparent)
```

Versus

```
(define (add e1 e2) (list 'add e1 e2))
(define (add? e) (eq? (car e) 'add))
(define (add-e1 e) (car (cdr e)))
(define (add-e2 e) (car (cdr (cdr e))))
```

This is *not* a case of syntactic sugar

The key difference

```
(struct add (e1 e2) #:transparent)
```

- The result of calling **`(add x y)`** is *not* a list
  - And there is no list for which **`add?`** returns **`#t`**

- **`struct`** makes a new kind of thing: extending Racket with a new kind of data

- So calling **`car`**, **`cdr`**, or **`mult-e1`** on "an add" is a run-time error

List approach is error-prone

```
(define (add e1 e2) (list 'add e1 e2))
(define (add? e) (eq? (car e) 'add))
(define (add-e1 e) (car (cdr e)))
(define (add-e2 e) (car (cdr (cdr e))))
```

- Can break abstraction by using **`car`**, **`cdr`**, and list-library functions directly on "add expressions"
  - Silent likely error:

```
(define xs (list (add (const 1)(const 4)) …))
(car (car xs))
```

- Can make data that **`add?`** wrongly answers **`#t`** to

```
(cons 'add "I am not an add")
```

Summary of advantages

Struct approach:

- Is better style and more concise for *defining* data types
- Is about equally convenient for *using* data types
- But much better at timely errors when *misusing* data types
  - Cannot use accessor functions on wrong kind of data
  - Cannot confuse tester functions

More with abstraction

Struct approach is even better combined with other Racket features not discussed here:

- The *module system* lets us hide the constructor function to enforce invariants
  - List-approach cannot hide cons from clients
  - Dynamically-typed languages can have abstract types by letting modules define new types!

- The *contract system* lets us check invariants even if constructor is exposed
  - For example, fields of “an add” must also be “expressions”

Struct is special

Often we end up learning that some convenient feature could be coded up with other features

Not so with struct definitions:

- A function cannot introduce multiple bindings
- Neither functions nor macros can create a new kind of data
  - Result of constructor function returns **`#f`** for *every* other tester function: **`number?`**, **`pair?`**, other structs' tester functions, etc.

PAUL G. ALLEN SCHOOL
OF COMPUTER SCIENCE & ENGINEERING

CSE341: Programming Languages

Lecture 17
Implementing Languages Including Closures

Dan Grossman

Spring 2019

Typical workflow

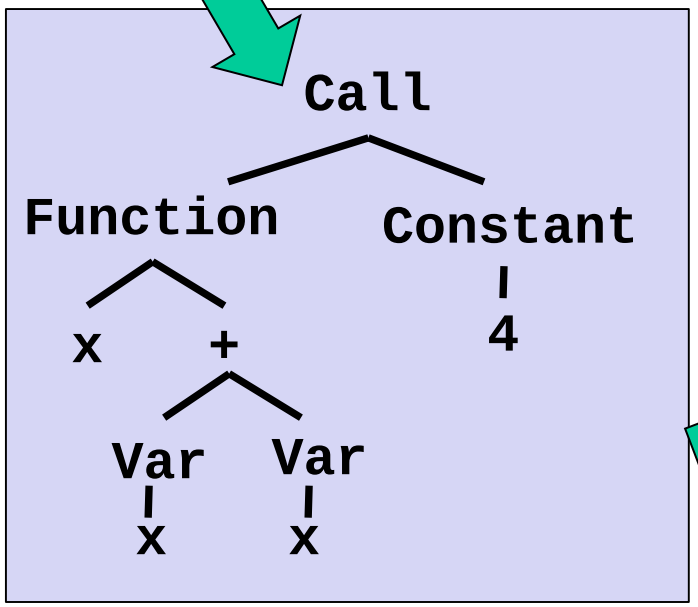
concrete syntax (string)

```
"(fn x => x + x) 4"
```

Possible errors / warnings

Parsing

abstract syntax (tree)



Possible errors / warnings

Type checking?

Rest of implementation

Interpreter or compiler

So “rest of implementation” takes the abstract syntax tree (AST) and “runs the program” to produce a result

Fundamentally, two approaches to implement a PL *B*:

- Write an interpreter in another language *A*
  - Better names: evaluator, executor
  - Take a program in *B* and produce an answer (in *B*)
- Write a compiler in another language *A* to a third language *C*
  - Better name: translator
  - Translation must *preserve meaning* (equivalence)

We call *A* the metalanguage

  - Crucial to keep *A* and *B* straight

Reality more complicated

Evaluation (interpreter) and translation (compiler) are your options

- But in modern practice have both and multiple layers

A plausible example:

- Java compiler to bytecode intermediate language
- Have an interpreter for bytecode (itself in binary), but compile frequent functions to binary at run-time
- The chip is itself an interpreter for binary
  - Well, except these days the x86 has a translator in hardware to more primitive micro-operations it then executes

DrRacket uses a similar mix

Sermon

Interpreter versus compiler versus combinations is about a particular language **implementation**, not the language **definition**

So there is no such thing as a "compiled language" or an "interpreted language"

- Programs cannot "see" how the implementation works

Unfortunately, you often hear such phrases

- "C is faster because it's compiled and LISP is interpreted"
- This is nonsense; politely correct people

- (Admittedly, languages with "eval" must "ship with some implementation of the language" in each program)

Typical workflow

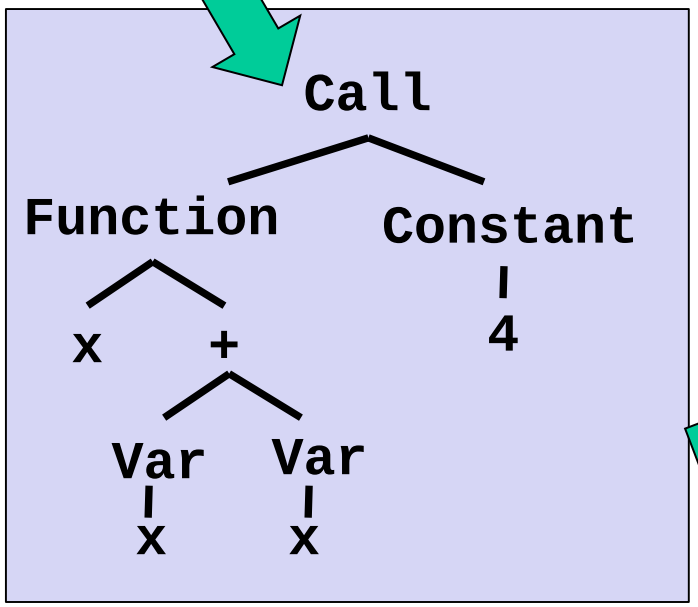
concrete syntax (string)

```
"(fn x => x + x) 4"
```

Possible errors / warnings

Parsing

abstract syntax (tree)



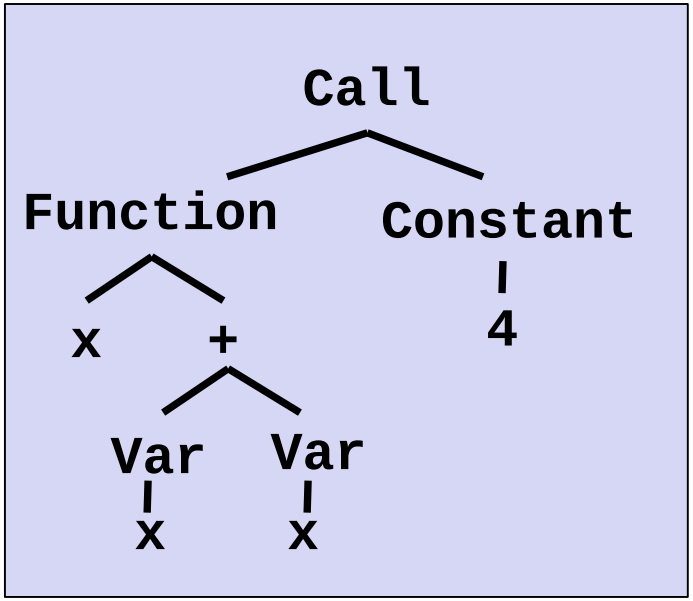
Possible errors / warnings

Type checking?

Rest of implementation

Skipping parsing

- If implementing PL *B* in PL *A*, we can skip parsing
  - Have *B* programmers write ASTs directly in PL *A*
  - Not so bad with ML constructors or Racket structs
  - Embeds *B* programs as trees in *A*




```
; define B's abstract syntax
(struct call …)
(struct function …)
(struct var …)
…
```

```
; example B program
(call (function (list "x")
                (add (var "x")
                     (var "x")))
      (const 4))
```

Already did an example!

- Let the metalanguage *A* = Racket
- Let the language-implemented *B* = "*Arithmetic Language*"
- Arithmetic programs written with calls to Racket constructors
- The interpreter is **`eval-exp`**

```
(struct const (int) #:transparent)
(struct negate (e) #:transparent)
(struct add (e1 e2) #:transparent)
(struct multiply (e1 e2) #:transparent)
```

```
(define (eval-exp e)
  (cond [(const? e) e]
        [(negate? e)
         (const (- (const-int
                    (eval-exp (negate-e e)))))]
        [(add? e) …]
        [(multiply? e) …]…
```

Racket data structure is *Arithmetic Language* program, which **`eval-exp`** runs

What we know

- Define (abstract) syntax of language *B* with Racket structs
  - *B* called MUPL in homework
- Write *B* programs directly in Racket via constructors
- Implement interpreter for *B* as a (recursive) Racket function

Now, a subtle-but-important distinction:

- Interpreter can *assume* input is a "legal AST for B"
  - Okay to give wrong answer or inscrutable error otherwise
- Interpreter *must check* that recursive results are the right kind of *value*
  - Give a good error message otherwise

Legal ASTs

- "Trees the interpreter must handle" are a subset of all the trees Racket allows as a dynamically typed language

```
(struct const (int) #:transparent)
(struct negate (e) #:transparent)
(struct add (e1 e2) #:transparent)
(struct multiply (e1 e2) #:transparent)
```

- Can assume "right types" for struct fields
  - **`const`** holds a number
  - **`negate`** holds a legal AST
  - **`add`** and **`multiply`** hold 2 legal ASTs

- Illegal ASTs can "crash the interpreter" – *this is fine*

```
(multiply (add (const 3) "uh-oh") (const 4))
(negate -7)
```

Interpreter results

- Our interpreters return expressions, but not any expressions
  - Result should always be a *value*, a kind of expression that evaluates to itself
  - If not, the interpreter has a bug
- So far, only values are from **`const`**, e.g., **`(const 17)`**
- But a larger language has more values than just numbers
  - Booleans, strings, etc.
  - Pairs of values (definition of value recursive)
  - Closures
  - …

Example

See code for language that adds booleans, number-comparison, and conditionals:

```
(struct bool (b) #:transparent)
(struct eq-num (e1 e2) #:transparent)
(struct if-then-else (e1 e2 e3) #:transparent)
```

What if the program is a legal AST, but evaluation of it tries to use the wrong kind of value?

- For example, “add a boolean”
- You should detect this and give an error message not in terms of the interpreter implementation
- Means checking a recursive result whenever a particular kind of value is needed
  - No need to check if any kind of value is okay

Dealing with variables

- Interpreters so far have been for languages without variables
  - No let-expressions, functions-with-arguments, etc.
  - Language in homework has all these things

- This segment describes in English what to do
  - Up to you to translate this to code

- Fortunately, what you have to implement is what we have been stressing since the very, very beginning of the course

Dealing with variables

- An environment is a mapping from variables (Racket strings) to values (as defined by the language)
  - Only ever put pairs of strings and values in the environment

- Evaluation takes place in an environment
  - Environment passed as argument to interpreter helper function
  - A variable expression looks up the variable in the environment
  - Most subexpressions use same environment as outer expression
  - A let-expression evaluates its body in a larger environment

The Set-up

So now a recursive helper function has all the interesting stuff:

```
(define (eval-under-env e env)
   (cond … ; case for each kind of
    ))     ; expression
```

– Recursive calls must "pass down" correct environment

Then **`eval-exp`** just calls **`eval-under-env`** with same expression and the *empty environment*

On homework, environments themselves are just Racket lists containing Racket pairs of a string (the MUPL variable name, e.g., **`"x"`**) and a MUPL value (e.g., **`(int 17)`**)

A grading detail

- Stylistically **`eval-under-env`** would be a helper function one could define locally inside **`eval-exp`**

- But do not do this on your homework
  - We have grading tests that call **`eval-under-env`** directly, so we need it at top-level

The best part

- The most interesting and mind-bending part of the homework is that the language being implemented has first-class closures
  - With lexical scope of course

- Fortunately, what you have to implement is what we have been stressing since we first learned about closures…

Higher-order functions

The “magic”: How do we use the “right environment” for lexical scope when functions may return other functions, store them in data structures, etc.?

Lack of magic: The interpreter uses a closure data structure (with two parts) to keep the environment it will need to use later

```
(struct closure (env fun) #:transparent)
```

Evaluate a function expression:

- A function is *not* a value; a closure *is* a value
  - Evaluating a function returns a closure
- Create a closure out of (a) the function and (b) the current environment when the function was evaluated

Evaluate a function call:

- …

Function calls

```
(call e1 e2)
```

- Use current environment to evaluate **`e1`** to a closure
  - Error if result is a value that is not a closure
- Use current environment to evaluate **`e2`** to a value
- Evaluate closure's function's body in the closure's environment, extended to:
  - Map the function's argument-name to the argument-value
  - And for recursion, map the function's name to the whole closure

This is the same semantics we learned a few weeks ago "coded up"

Given a closure, the code part is *only* ever evaluated using the environment part (extended), *not* the environment at the call-site

Is that expensive?

- *Time* to build a closure is tiny: a struct with two fields

- *Space* to store closures *might* be large if environment is large
  - But environments are immutable, so natural and correct to have lots of sharing, e.g., of list tails (cf. lecture 3)
  - Still, end up keeping around bindings that are not needed

- Alternative used in practice: When creating a closure, store a possibly-smaller environment holding only the variables that are free variables in the function body
  - Free variables: Variables that occur, not counting shadowed uses of the same variable name
  - A function body would never need anything else from the environment

Free variables examples

```
(lambda () (+ x y z))   ; {x, y, z}

(lambda (x) (+ x y z))  ; {y, z}

(lambda (x) (if x y z)) ; {y, z}

(lambda (x) (let ([y 0]) (+ x y z))) ; {z}

(lambda (x y z) (+ x y z)) ; {}

(lambda (x) (+ y (let ([y z]) (+ y y)))) ; {y, z}
```

Computing free variables

- So does the interpreter have to analyze the code body every time it creates a closure?

- No: Before evaluation begins, compute free variables of every function in program and store this information with the function

- Compared to naïve store-entire-environment approach, building a closure now takes more time but less space
  - And time proportional to number of free variables
  - And various optimizations are possible

- [Also use a much better data structure for looking up variables than a list]

Optional: compiling higher-order functions

- If we are compiling to a language without closures (like assembly), cannot rely on there being a "current environment"
- So compile functions by having the translation produce "regular" functions that *all* take an *extra explicit argument* called "environment"
- And compiler replaces all uses of free variables with code that looks up the variable using the environment argument
  - Can make these fast operations with some tricks
- Running program still creates closures and every function call passes the closure's environment to the closure's code

Recall…

Our approach to language implementation:

- Implementing language *B* in language *A*
- Skipping parsing by writing language *B* programs directly in terms of language *A* constructors
- An interpreter written in *A* recursively evaluates

What we know about macros:

- Extend the syntax of a language
- Use of a macro expands into language syntax before the program is run, i.e., before calling the main interpreter function

Put it together

With our set-up, we can use language *A* (i.e., Racket) *functions* that produce language *B* abstract syntax as language *B* "macros"

- Language *B* programs can use the "macros" as though they are part of language *B*
- No change to the interpreter or struct definitions
- Just a programming idiom enabled by our set-up
  - Helps teach what macros are
- See code for example "macro" definitions and "macro" uses
  - "macro expansion" happens before calling **`eval-exp`**

Hygiene issues

- Earlier we had material on hygiene issues with macros
  - (Among other things), problems with shadowing variables when using local variables to avoid evaluating expressions more than once

- The “macro” approach described here does not deal well with this

PAUL G. ALLEN SCHOOL
OF COMPUTER SCIENCE & ENGINEERING

CSE341: Programming Languages

Lecture 18
Static vs. Dynamic Typing

Dan Grossman

Spring 2019

Key differences

- Racket and ML have *much* in common

- Key differences
  - Syntax
  - Pattern-matching vs. struct-tests and accessor-functions
  - Semantics of various let-expressions
  - …

- *Biggest* difference: ML's type system and Racket's lack thereof *

* There is Typed Racket, which interacts well with Racket so you can have typed and untyped modules, but we won't study it, and it differs in interesting ways from ML

The plan

Key questions:

- What is type-checking? Static typing? Dynamic typing? Etc.
- Why is type-checking approximate?
- What are the advantages and disadvantages of type-checking?

But first to better appreciate ML and Racket:

- How could a Racket programmer describe ML?
- How could an ML programmer describe Racket?

ML from a Racket perspective

- Syntax, etc. aside, ML is like a well-defined subset of Racket

- Many of the programs it disallows have bugs ☺

```
(define (g x) (+ x x)) ; ok
(define (f y) (+ y (car y)))
(define (h z) (g (cons z 2)))
```

  - In fact, in what ML allows, I never need primitives like **`number?`**

- But other programs it disallows I may actually want to write ☹

```
(define (f x) (if (> x 0) #t (list 1 2)))
(define xs (list 1 #t "hi"))
(define y (f (car xs)))
```

Racket from an ML Perspective

One way to describe Racket is that it has "one big datatype"

- *All* values have this type

```
datatype theType = Int of int  | String of string
                 | Pair of theType * theType
                 | Fun of theType -> theType
                 | …
```

- Constructors are applied implicitly (values are *tagged*)
  - **`42`** is really like **`Int 42`**

- Primitives implicitly *check tags and extract data*, raising errors for wrong constructors

```
fun car v = case v of Pair(a,b) => a | _  => raise …
fun pair? v = case v of Pair _ => true | _  => false
```

More on The One Type

- Built-in constructors for “theType”: numbers, strings, booleans, pairs, symbols, procedures, etc.

- Each struct-definition creates a *new constructor*, dynamically adding to “theType”

Static checking

- *Static checking* is anything done to reject a program *after* it (successfully) parses but *before* it runs

- ***Part of a PL's definition: what static checking is performed***
  - A "helpful tool" could do more checking

- Common way to define a PL's static checking is via a *type system*
  - *Approach* is to give each variable, expression, etc. a type
  - *Purposes* include preventing misuse of primitives (e.g., **`4/"hi"`**), enforcing abstraction, and avoiding dynamic checking
    - Dynamic means at run-time

- Dynamically-typed languages do (almost) no static checking
  - Line is not absolute

Example: ML, what types prevent

In ML, type-checking ensures a program (when run) will **never** have:

- A primitive operation used on a value of the wrong type
  - Arithmetic on a non-number
  - **`e1 e2`** where **`e1`** does not evaluate to a function
  - A non-boolean between **`if`** and **`then`**
- A variable not defined in the environment
- A pattern-match with a redundant pattern
- Code outside a module call a function not in the module's signature
- …

(First two are "standard" for type systems, but different languages' type systems ensure different things)

Example: ML, what types allow

In ML, type-checking does ***not*** prevent any of these errors

- Instead, detected at run-time

- Calling functions such that exceptions occur, e.g., **hd []**
- An array-bounds error
- Division-by-zero

In general, no type system prevents logic / algorithmic errors:

- Reversing the branches of a conditional
- Calling **f** instead of **g**

(Without a program specification, type-checker can't "read minds")

Purpose is to prevent something

Have discussed facts about *what* the ML type system does and does not prevent

- Separate from *how* (e.g., one type for each variable) though previously studied many of ML's typing rules

Language design includes deciding *what* is checked and *how*

Hard part is making sure the type system "achieves its purpose"

- That "the how" accomplishes "the what"
- More precise definition next

A question of eagerness

"Catching a bug before it matters"

is in inherent tension with

"Don't report a bug that might not matter"

Static checking / dynamic checking are two points on a continuum

Silly example: Suppose we just want to prevent evaluating **`3 / 0`**

- Keystroke time: disallow it in the editor
- Compile time: disallow it if seen in code
- Link time: disallow it if seen in code that may be called to evaluate **`main`**
- Run time: disallow it right when we get to the division
- Later: Instead of doing the division, return **`+inf.0`** instead
  - Just like **`3.0 / 0.0`** does in every (?) PL (it's useful!)

Correctness

Suppose a type system is supposed to prevent X for some X

- A type system is *sound* if it never accepts a program that, when run with some input, does X
  - No *false negatives*

- A type system is *complete* if it never rejects a program that, no matter what input it is run with, will not do X
  - No *false positives*

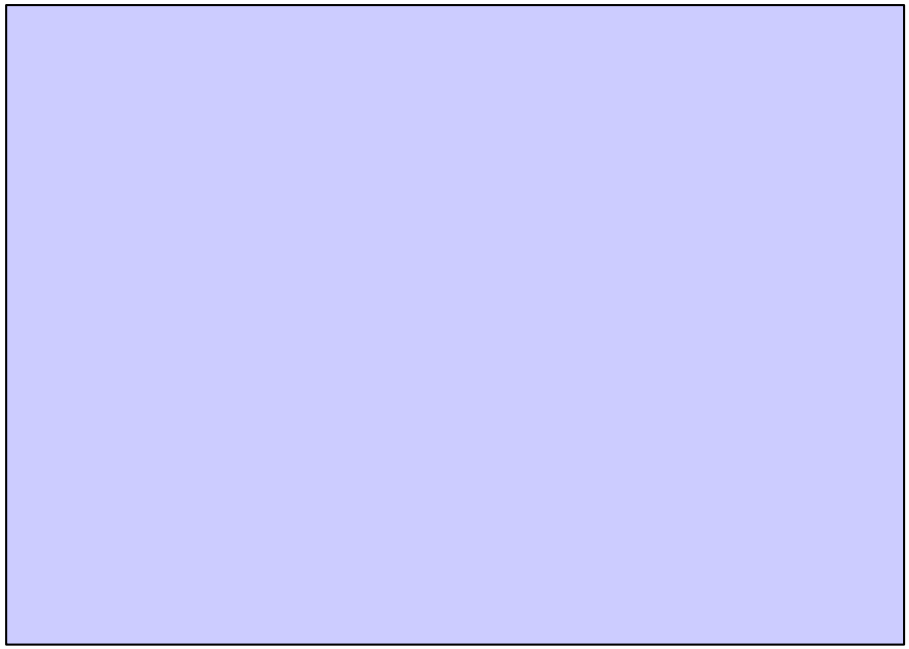
The goal is usually for a PL type system to be sound (so you can rely on it) but not complete

  - "Fancy features" like generics aimed at "fewer false positives"

Notice soundness/completeness is with respect to X

Venn Diagrams

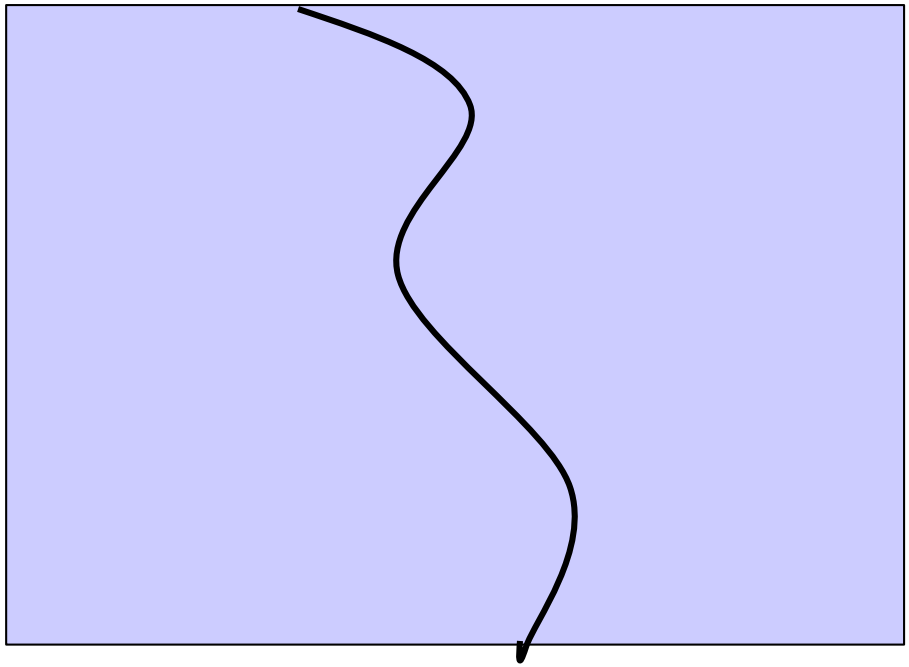
All possible programs
(in syntax of some language)



Venn Diagrams

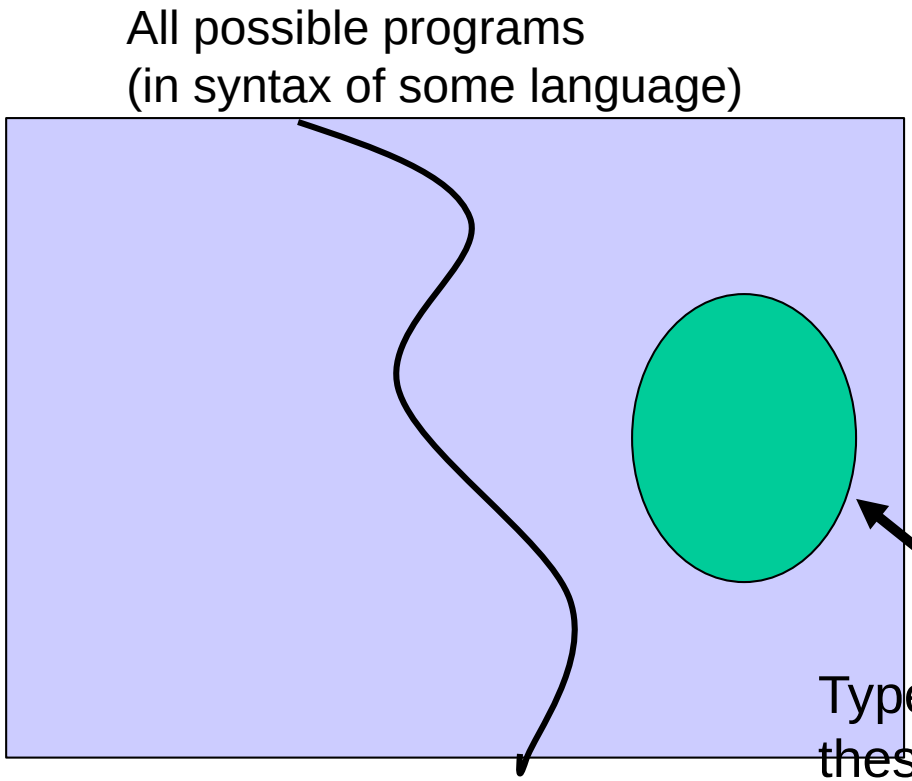
All possible programs
(in syntax of some language)

Programs for which there is >= 1 input that makes the program do bad thing X

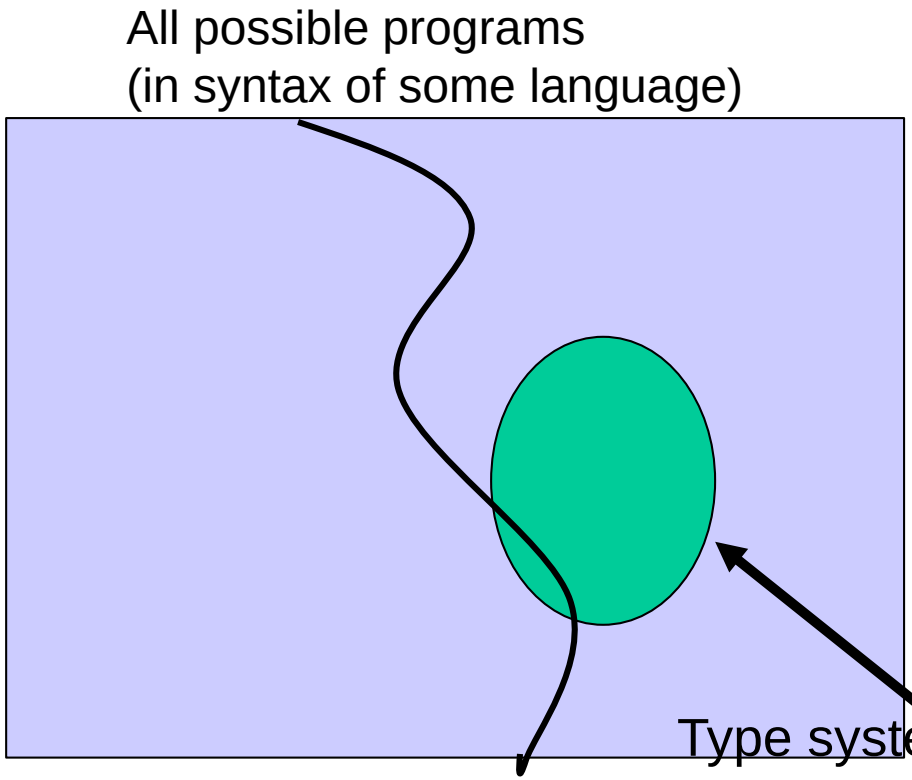


Programs for which there is no input that makes the program do bad thing X

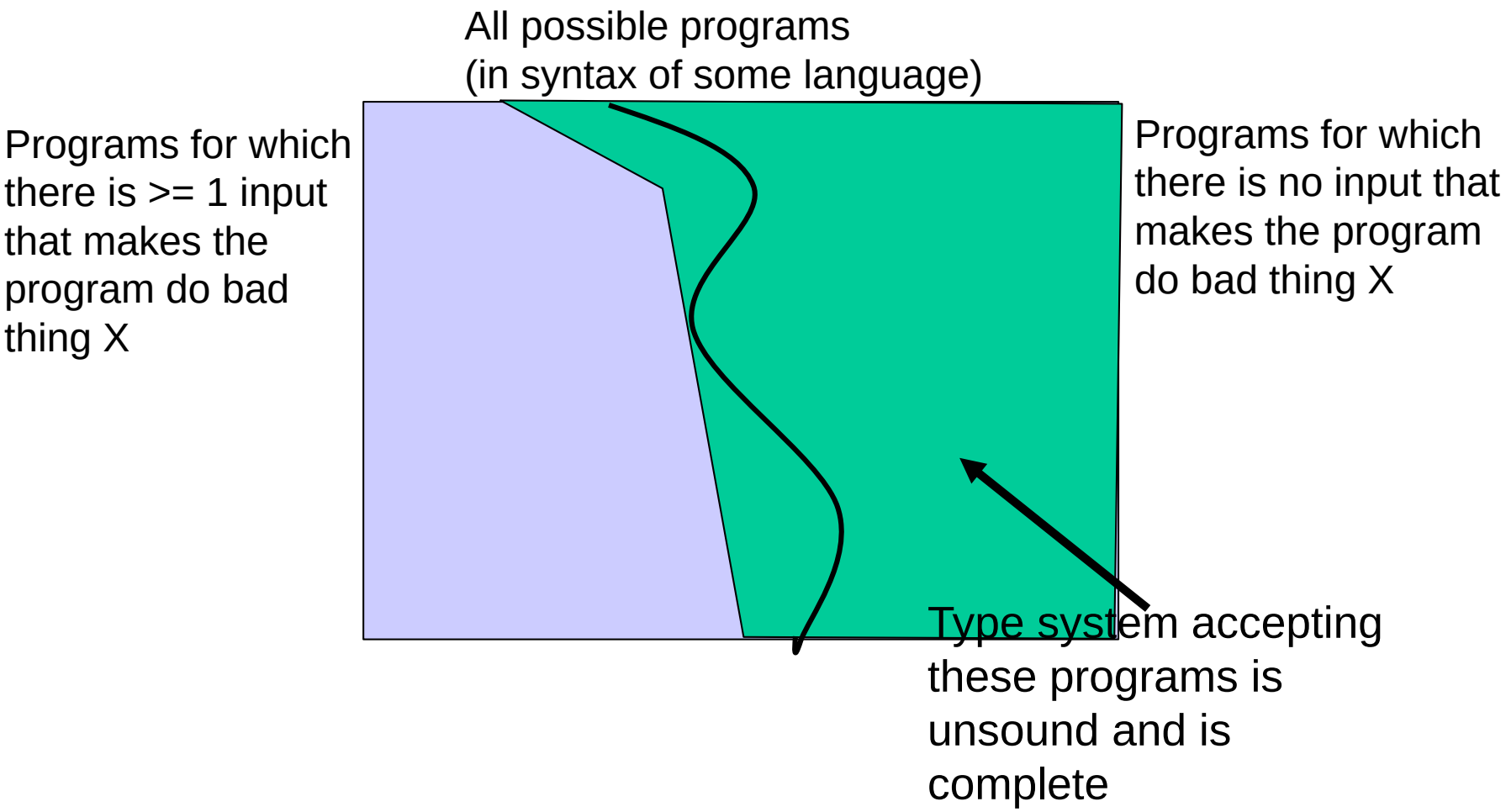
Venn Diagrams



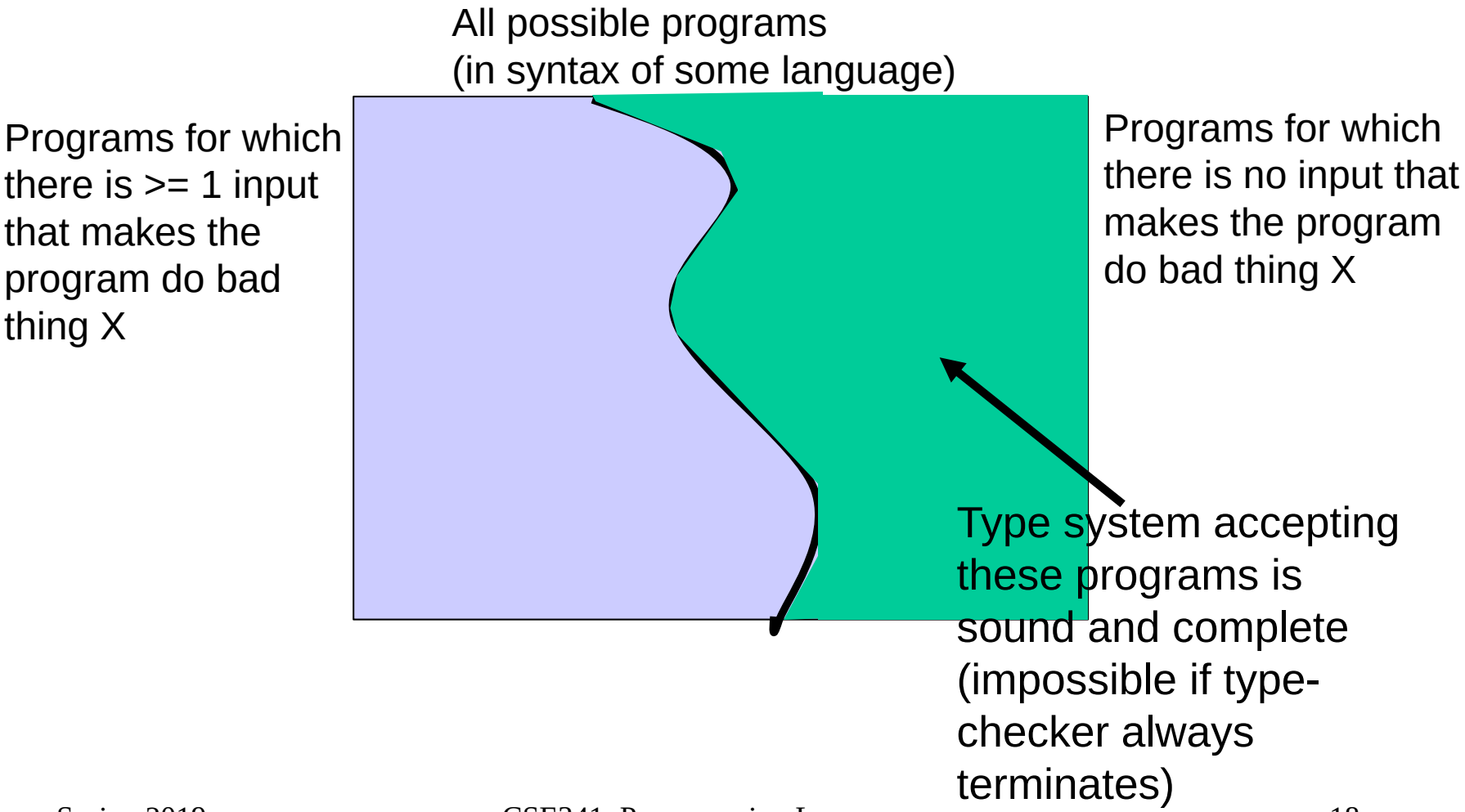
Venn Diagrams



Venn Diagrams



Venn Diagrams



Incompleteness

A few functions ML rejects even though they do not divide by a string

```
fun f1 x = 4 div "hi" (* but f1 never called *)

fun f2 x = if true then 0 else 4 div "hi"

fun f3 x = if x then 0 else 4 div "hi"
val x = f3 true

fun f4 x = if x <= abs x then 0 else 4 div "hi"

fun f5 x = 4 div x
val y = f5 (if true then 1 else "hi")
```

Why incompleteness

- Almost anything you might like to check statically is undecidable:
  - Any static checker *cannot* do all of: (1) always terminate, (2) be sound, (3) be complete
  - This is a mathematical theorem!
- Examples:
  - Will this function terminate on some 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 an essential concept at the core of computing
  - The inherent approximation of static checking is probably its most important ramification

What about unsoundness?

Suppose a type system were unsound. What could the PL do?

- Fix it with an updated language definition?
- Insert dynamic checks as needed to prevent X from happening?
- Just allow X to happen even if “tried to stop it”?
- Worse: Allow not just X, but *anything* to happen if “programmer gets something wrong”
  - Will discuss C and C++ next…

Why weak typing (C/C++)

Weak typing: There exist programs that, by definition, *must* pass static checking but then when run can "set the computer on fire"?

- Dynamic checking is optional and in practice not done
- Why might anything happen?

- Ease of language implementation: Checks left to the programmer
- Performance: Dynamic checks take time
- Lower level: Compiler does not insert information like array sizes, so it cannot do the checks

Weak typing is a poor name: Really about doing *neither* static nor dynamic checks

- A big problem is array bounds, which most PLs check dynamically

What weak typing has caused

- Old now-much-rarer saying: “strong types for weak minds”
  - Idea was humans will always be smarter than a type system (cf. undecidability), so need to let them say “trust me”
- Reality: humans are really bad at avoiding bugs
  - We need all the help we can get!
  - And type systems have gotten much more expressive (fewer false positives)
- 1 bug in a 30-million line operating system written in C can make an entire computer vulnerable
  - An important bug like this was probably announced this week (because there is one almost every week)

Example: Racket

- Racket is ***not*** weakly typed
  - It just checks most things dynamically*
  - Dynamic checking is the *definition* – if the *implementation* can analyze the code to ensure some checks are not needed, then it can *optimize them away*

- Not having ML or Java's rules can be convenient
  - Cons cells can build anything
  - Anything except `#f` is true
  - …

  This is nothing like the "catch-fire semantics" of weak typing

*Checks macro usage and undefined-variables in modules statically

Another misconception

What operations are primitives defined on and when an error?

- Example: Is **`"foo" + "bar"`** allowed?
- Example: Is **`"foo" + 3`** allowed?
- Example: Is **`arr[10]`** allowed if **`arr`** has only 5 elements?
- Example: Can you call a function with too few or too many arguments?

This is not static vs. dynamic checking (sometimes confused with it)

- It is “what is the run-time semantics of the primitive”
- It is related because it also involves trade-offs between catching bugs sooner versus maybe being more convenient

Racket generally less lenient on these things than, e.g., Ruby

Now can argue…

Having carefully stated facts about static checking,

can *now* consider arguments about which is *better*:

static checking or dynamic checking

Remember most languages do some of each

- For example, perhaps types for primitives are checked statically, but array bounds are not

Claim 1a: Dynamic is more convenient

Dynamic typing lets you build a heterogeneous list or return a "number or a string" without workarounds

```
(define (f y)
  (if (> y 0) (+ y y) "hi"))

(let ([ans (f x)])
  (if (number? ans) (number->string ans) ans))
```

```
datatype t = Int of int | String of string
fun f y = if y > 0 then Int(y+y) else String "hi"

case f x of
   Int i => Int.toString i
 | String s => s
```

Claim 1b: Static is more convenient

Can assume data has the expected type without cluttering code with dynamic checks or having errors far from the logical mistake

```
(define (cube x)
  (if (not (number? x))
      (error "bad arguments")
      (* x x x)))

(cube 7)
```

```
fun cube x = x * x * x

cube 7
```

Claim 2a: Static prevents useful programs

Any sound static type system forbids programs that do nothing wrong, forcing programmers to code around limitations

```
(define (f g)
  (cons (g 7) (g #t)))

(define pair_of_pairs
  (f (lambda (x) (cons x x))))
```

```
fun f g = (g 7, g true) (* does not type-check *)

val pair_of_pairs = f (fn x => (x,x))
```

Claim 2b: Static lets you tag as needed

Rather than suffer time, space, and late-errors costs of tagging everything, statically typed languages let programmers “tag as needed” (e.g., with datatypes)

In the extreme, can use "TheOneRacketType" in ML

- Extreme rarely needed in practice

```
datatype tort = Int of int
              | String of string
              | Cons of tort * tort
              | Fun of tort -> tort
              | …

if e1
then Fun (fn x => case x of Int i => Int (i*i*i))
else Cons (Int 7, String "hi")
```

Claim 3a: Static catches bugs earlier

Static typing catches many simple bugs as soon as “compiled”

- Since such bugs are always caught, no need to test for them
- In fact, can code less carefully and “lean on” type-checker

```
(define (pow x) ; curried
  (lambda (y)
    (if (= y 0)
        1
        (* x (pow x (- y 1))))))) ; oops
```

```
fun pow x y =  (* does not type-check *)
  if y = 0
  then 1
  else x * pow (x,y-1)
```

Claim 3b: Static catches only easy bugs

But static often catches only “easy” bugs, so you still have to test your functions, which should find the “easy” bugs too

```
(define (pow x) ; curried
  (lambda (y)
    (if (= y 0)
        1
        (+ x ((pow x) (- y 1)))))) ; oops
```

```
fun pow x y =  (* curried *)
  if y = 0
  then 1
  else x + pow x (y-1) (* oops *)
```

Claim 4a: Static typing is faster

Language implementation:

- Does not need to store tags (space, time)
- Does not need to check tags (time)

Your code:

- Does not need to check arguments and results

Claim 4b: Dynamic typing is faster

Language implementation:

- Can use optimization to remove some unnecessary tags and tests
  - Example: `(let ([x (+ y y)]) (* x 4))`
- While that is hard (impossible) in general, it is often easier for the performance-critical parts of a program

Your code:

- Do not need to "code around" type-system limitations with extra tags, functions etc.

Claim 5a: Code reuse easier with dynamic

Without a restrictive type system, more code can just be reused with data of different types

- If you use cons cells for everything, libraries that work on cons cells are useful
- Collections libraries are amazingly useful but often have very complicated static types
- Etc.

Claim 5b: Code reuse easier with static

- Modern type systems should support reasonable code reuse with features like generics and subtyping

- If you use cons cells for everything, you will confuse what represents what and get hard-to-debug errors
  - Use separate static types to keep ideas separate
  - Static types help avoid library *misuse*

So far

Considered 5 things important when writing code:

1. Convenience
2. Not preventing useful programs
3. Catching bugs early
4. Performance
5. Code reuse

But took the naive view that software is developed by taking an existing spec, coding it up, testing it, and declaring victory.

Reality:

- Often a lot of prototyping *before* a spec is stable
- Often a lot of maintenance / evolution *after* version 1.0

Claim 6a: Dynamic better for prototyping

Early on, you may not know what cases you need in datatypes and functions

- But static typing disallows code without having all cases; dynamic lets incomplete programs run
- So you make premature commitments to data structures
- And end up writing code to appease the type-checker that you later throw away
  - Particularly frustrating while prototyping

Claim 6b: Static better for prototyping

What better way to document your evolving decisions on data structures and code-cases than with the type system?

- New, evolving code most likely to make inconsistent assumptions

Easy to put in temporary stubs as necessary, such as

```
| _ => raise Unimplemented
```

Claim 7a: Dynamic better for evolution

Can change code to be more permissive without affecting old callers

- Example: Take an **`int`** or a **`string`** instead of an **`int`**
- *All* ML callers must now use a constructor on arguments and pattern-match on results
- Existing Racket callers can be *oblivious*

```
(define (f x) (* 2 x))
```

```
(define (f x)
  (if (number? x)
      (* 2 x)
      (string-append x x)))
```

```
fun f x = 2 * x
```

```
fun f x =
  case f x of
     Int i    => Int (2 * i)
   | String s => String(s ^ s)
```

Claim 7b: Static better for evolution

When we change type of data or code, the type-checker gives us a "to do" list of everything that must change

- Avoids introducing bugs
- The more of your spec that is in your types, the more the type-checker lists what to change when your spec changes

Example: Changing the return type of a function

Example: Adding a new constructor to a datatype

- Good reason not to use wildcard patterns

Counter-argument: The to-do list is mandatory, which makes evolution in pieces a pain: cannot test part-way through

Coda

- Static vs. dynamic typing is too coarse a question
  - Better question: *What* should we enforce statically?

- Legitimate trade-offs you should know
  - Rational discussion informed by facts!

- Ideal (?): Flexible languages allowing best-of-both-worlds?
  - Would programmers use such flexibility well? Who decides?
  - "Gradual typing": a great idea still under active research

PAUL G. ALLEN SCHOOL
OF COMPUTER SCIENCE & ENGINEERING

CSE341: Programming Languages

Lecture 19
Introduction to Ruby and OOP

Dan Grossman

Spring 2019

Ruby logistics

- Next two sections use the Ruby language
  - http://www.ruby-lang.org/
  - Installation / basic usage instructions on course website
    - Version 2.X.Y required, but differences not so relevant

- Excellent documentation available, much of it free
  - So may not cover every language detail in course materials
  - http://ruby-doc.org/
  - http://www.ruby-lang.org/en/documentation/
  - Particularly recommend “Programming Ruby 1.9 & 2.0, The Pragmatic Programmers’ Guide”
    - Not free

Ruby: Our focus

- *Pure object-oriented*: *all* values are *objects* (even numbers)
- *Class-based*: Every object has a class that determines behavior
  - Like Java, unlike Javascript
  - *Mixins* (not [old] Java interfaces nor C++ multiple inheritance)
- *Dynamically typed*
- Convenient *reflection*: Run-time inspection of objects
- Very *dynamic*: Can change classes during execution
- *Blocks* and libraries encourage lots of closure idioms
- Syntax, scoping rules, semantics of a "*scripting language*"
  - Variables "spring to life" on use
  - Very flexible arrays

Ruby: Not our focus

- Lots of support for string manipulation and regular expressions
- Popular for server-side web applications
  - Ruby on Rails
- Often many ways to do the same thing
  - More of a “why not add that too?” approach

Where Ruby fits

	dynamically typed	statically typed
functional	Racket	SML
object-oriented (OOP)	Ruby	Java

Note: Racket also has classes and objects when you want them

- In Ruby everything uses them (at least implicitly)

Historical note: *Smalltalk* also a dynamically typed, class-based, pure OOP language with blocks and convenient reflection

- Smaller just-as-powerful language
- Ruby less simple, more “modern and useful”

Dynamically typed OOP helps identify OOP's essence by not having to discuss types

A note on the homework

Next homework is about understanding and extending an *existing* program in an *unfamiliar* language

- Good practice
- Quite different feel than previous homeworks
- *Read* code: determine what you do and do not (!) need to understand

Homework requires the Tk graphics library to be installed such that the provided Ruby code can use it

Getting started

- See **`lec19_silly.rb`** file for our getting-started program
- Can run file **`foo.rb`** at the command-line with **`ruby foo.rb`**
- Or can use **`irb`**, which is a REPL
  - Run file **`foo.rb`** with **`load "foo.rb"`**

The rules of class-based OOP

In Ruby:

1. All values are references to *objects*
2. Objects communicate via *method calls*, also known as *messages*
3. Each object has its own (private) *state*
4. Every object is an instance of a *class*
5. An object's class determines the object's *behavior*
   - How it handles method calls
   - Class contains method definitions

Java/C#/etc. similar but do not follow (1) (e.g., numbers, **`null`**) and allow objects to have non-private state

Defining classes and methods

```
class Name
  def method_name1 method_args1
    expression1
  end
  def method_name2 method_args2
    expression2
  end
  …
end
```

- Define a class with methods as defined
- Method returns its last expression
  - Ruby also has explicit **`return`** statement
- Syntax note: Line breaks often required (else need more syntax), but indentation always only style

Creating and using an object

- **`ClassName.new`** creates a new object whose class is **`ClassName`**

- **`e.m`** evaluates **`e`** to an object and then calls its **`m`** method
  - Also known as "sends the **`m`** message"
  - Can also write **`e.m()`** with no space

- Methods can take arguments, called like **`e.m(e1,…,en)`**
  - Parentheses optional in some places, but recommended

Variables

- Methods can use local variables
  - Syntax: starts with letter
  - Scope is method body
- No declaring them, just assign to them anywhere in method body (!)
- Variables are mutable, **x=e**
- Variables also allowed at “top-level” or in REPL
- Contents of variables are always references to objects because all values are objects

Self

- **`self`** is a special keyword/variable in Ruby
  - (Same as **`this`** in Java/C#/C++)

- Refers to "the current object"
  - The object whose method is executing

- So call another method on "same object" with **`self.m(…)`**
  - Syntactic sugar: can just write **`m(…)`**

- Also can pass/return/store "the whole object" with just **`self`**

Objects have state

- An object's state persists
  - Can grow and change from time object is created

- State only directly accessible from object's methods
  - Can read, write, extend the state
  - Effects persist for next method call

- State consists of *instance variables* (also known as fields)
  - Syntax: starts with an @, e.g., **`@foo`**
  - "Spring into being" with assignment
    - So mis-spellings silently add new state (!)
  - Using one not in state not an error; produces **`nil`** object

Aliasing

- Creating an object returns a reference to a new object
  - Different state from every other object

- Variable assignment (e.g., **x=y**) creates an alias
  - Aliasing means same object means same state

Initialization

- A method named **`initialize`** is special
  - Is called on a new object before **`new`** returns
  - Arguments to **`new`** are passed on to **`initialize`**
  - Excellent for creating object invariants
  - (Like constructors in Java/C#/etc.)

- Usually good *style* to create instance variables in **`initialize`**
  - Just a convention
  - Unlike OOP languages that make "what fields an object has" a (fixed) part of the class definition
    - In Ruby, different instances of same class can have different instance variables

Class variables

- There is also state shared by the entire class
- Shared by (and only accessible to) all instances of the class
  - (Like Java static fields)
- Called *class variables*
  - Syntax: starts with an @@, e.g., `@@foo`
- Less common, but sometimes useful
  - And helps explain via contrast that each object has its own instance variables

Class constants and methods

- *Class constants*
  - Syntax: start with capital letter, e.g., **`Foo`**
  - Should not be mutated
  - Visible outside class **`C`** as **`C::Foo`** (unlike class variables)

- *Class methods* (cf. Java/C# static methods)
  - Syntax (in some class **`C`**):

```
def self.method_name (args)
  …
end
```

  - Use (of class method in class **`C`**):

```
C.method_name(args)
```

  - Part of the class, not a particular instance of it

Who can access what

- We know “hiding things” is essential for modularity and abstraction
- OOP languages generally have various ways to hide (or not) instance variables, methods, classes, etc.
  - Ruby is no exception
- Some basic Ruby rules here as an example…

Object state is private

- In Ruby, object state is always private
  - Only an object's methods can access its instance variables
  - Not even another instance of the same class
  - So can write **`@foo`**, but not **`e.@foo`**

- To make object-state publicly visible, define "getters" / "setters"
  - Better/shorter style coming next

```
def get_foo
  @foo
end
def set_foo x
  @foo = x
end
```

Conventions and sugar

- Actually, for field **`@foo`** the convention is to name the methods

```
def foo
  @foo
end
```

```
def foo= x
  @foo = x
end
```

- Cute sugar: When *using* a method ending in **=**, can have space before the **=**

```
e.foo = 42
```

- Because defining getters/setters is so common, there is shorthand for it in class definitions
  - Define just getters: **`attr_reader :foo, :bar, …`**
  - Define getters and setters: **`attr_accessor :foo, :bar, …`**

- Despite sugar: getters/setters are just methods

Why private object state

- This is “more OOP” than public instance variables
- Can later change class implementation without changing clients
  - Like we did with ML modules that hid representation
  - And like we will soon do with subclasses
- Can have methods that “seem like” setters even if they are not

```
def celsius_temp= x
  @kelvin_temp = x + 273.15
end
```

- Can have an unrelated class that implements the same methods and use it with same clients
  - See later discussion of “duck typing”

Method visibility

- Three *visibilities* for methods in Ruby:
  - **`private`**: only available to object itself
  - **`protected`**: available only to code in the class or subclasses
  - **`public`**: available to all code

- Methods are **`public`** by default
  - Multiple ways to change a method's visibility
  - Here is one way…

Method visibilities

```
class Foo =
# by default methods public
   …
 protected
# now methods will be protected until
# next visibility keyword
   …
 public
   …

 private
   …
end
```

One detail

If **`m`** is private, then you can only call it via **`m`** or **`m(args)`**

- As usual, this is shorthand for **`self.m …`**
- But for private methods, only the shorthand is allowed

Now (see the code)

- Put together much of what we have learned to define and use a small class for rational numbers
  - Called **`MyRational`** because Ruby 1.9 has great built-in support for fractions using a class **`Rational`**

- Will also use several new and useful expression forms
  - Ruby is too big to show everything; see the documentation

- Way our class works: Keeps fractions in reduced form with a positive denominator
  - Like an ML-module example earlier in course

Pure OOP

- Ruby is fully committed to OOP:

  Every value is a reference to an object

- Simpler, smaller semantics

- Can call methods on anything
  - May just get a dynamic “undefined method” error

- Almost everything is a method call
  - Example: `3 + 4`

Some examples

- Numbers have methods like **`+`**, **`abs`**, **`nonzero?`**, etc.

- **`nil`** is an object used as a “nothing” object
  - Like **`null`** in Java/C#/C++ except it is an object
  - Every object has a **`nil?`** method, where **`nil`** returns **`true`** for it
  - Note: **`nil`** and **`false`** are “false”, everything else is “true”

- Strings also have a **`+`** method
  - String concatenation
  - Example: **`"hello" + 3.to_s`**

All code is methods

- All methods you define are part of a class
- Top-level methods just added to **`Object`** class
  - Private in file, public in REPL, more or less (details are weird and not so important to us)
- Subclassing discussion coming later, but:
  - Since all classes you define are *subclasses* of **`Object`**, all *inherit* the top-level methods
  - So you can call these methods anywhere in the program
  - Unless a class overrides (*roughly-not-exactly*, shadows) it by defining a method with the same name

Reflection and exploratory programming

- All objects also have methods like:
  - **`methods`**
  - **`class`**

- Can use at run-time to query "what an object can do" and respond accordingly
  - Called *reflection*

- Also useful in the REPL to explore what methods are available
  - May be quicker than consulting full documentation

- Another example of "just objects and method calls"

Changing classes

- Ruby programs (or the REPL) can add/change/replace methods while a program is running

- Breaks abstractions and makes programs very difficult to analyze, but it does have plausible uses
  - Simple example: Add a useful helper method to a class you did not define
    - Controversial in large programs, but may be useful

- For us: Helps re-enforce “the rules of OOP”
  - Every object has a class
  - A class determines its instances’ behavior

Examples

- Add a **`double`** method to our **`MyRational`** class
- Add a **`double`** method to the built-in **`FixNum`** class
- Defining top-level methods adds to the built-in **`Object`** class
  - Or replaces methods
- Replace the **`+`** method in the built-in **`FixNum`** class
  - Oops: watch **`irb`** crash

The moral

- Dynamic features cause interesting semantic questions

- Example:
  - First create an instance of class **`C`**, e.g., **`x = C.new`**
  - Now replace method method **`m`** in **`C`**
  - Now call **`x.m`**

  Old method or new method? In Ruby, new method

  The point is Java/C#/C++ do not have to ask the question
  - May allow more optimized method-call implementations as a result

Duck Typing

"If it walks like a duck and quacks like a duck, it's a duck"

- Or don't worry that it may not be a duck

When writing a method you might think, "I need a **Foo** argument" but really you need an object with enough methods similar to **Foo**'s methods that your method works

- Embracing duck typing is always making method calls rather than assuming/testing the class of arguments

Plus: More code reuse; very OOP approach

- What messages an object receive is "all that matters"

Minus: Almost nothing is equivalent

- **`x+x`** versus **`x*2`** versus **`2*x`**
- Callers may assume a lot about how callees are implemented

Duck Typing Example

```
def mirror_update pt
  pt.x = pt.x * (-1)
end
```

- Natural thought: "Takes a **`Point`** object (definition not shown here), negates the **`x`** value"
  - Makes sense, though a **`Point`** instance method more OOP
- Closer: "Takes anything with getter and setter methods for **`@x`** instance variable and multiplies the **`x`** field by **`-1`**"
- Closer: "Takes anything with methods **`x=`** and **`x`** and calls **`x=`** with the result of multiplying result of **`x`** and **`-1`**"
- Duck typing: "Takes anything with method **`x=`** and **`x`** where result of **`x`** has a **`*`** method that can take **`-1`**. Sends result of calling **`x`** the **`*`** message with **`-1`** and sends that result to **`x=`**"

With our example

```
def mirror_update pt
  pt.x = pt.x * (-1)
end
```

- Plus: Maybe **`mirror_update`** is useful for classes we did not anticipate
- Minus: If someone does use (abuse?) duck typing here, then we cannot change the implementation of **`mirror_update`**
  - For example, to **`- pt.x`**
- Better (?) example: Can pass this method a number, a string, or a **`MyRational`**

```
def double x
  x + x
end
```

PAUL G. ALLEN SCHOOL
OF COMPUTER SCIENCE & ENGINEERING

CSE341: Programming Languages

Lecture 20
Arrays and Such, Blocks and Procs, Inheritance and Overriding

Dan Grossman

Spring 2019

This lecture

Three mostly separate topics

- Flexible arrays, ranges, and hashes [actually covered in section]
- Ruby's approach to almost-closures (blocks) and closures (Procs)
  - [partially discussed in section as well]
  - Convenient to use; unusual approach
  - Used throughout large standard library
    - Explicit loops rare
    - Instead of a loop, go find a useful iterator
- Subclasses, inheritance, and overriding
  - The essence of OOP, now in a more dynamic language

Ruby Arrays

- Lots of special syntax and many provided methods for the **`Array`** class

- Can hold any number of other objects, *indexed* by number
  - Get via **`a[i]`**
  - Set via **`a[i] = e`**

- Compared to arrays in many other languages
  - More flexible and dynamic
  - Fewer operations are errors
  - Less efficient

- "The standard collection" (like lists were in ML and Racket)

Using Arrays

- See many examples, some demonstrated here
- Consult the documentation/tutorials
  - If seems sensible and general, probably a method for it
- Arrays make good tuples, lists, stacks, queues, sets, …
- Iterating over arrays typically done with methods taking blocks
  - Next topic…

Blocks

Blocks are probably Ruby's strangest feature compared to other PLs

But *almost* just closures

- Normal: easy way to pass anonymous functions to methods for all the usual reasons
- Normal: Blocks can take 0 or more arguments
- Normal: Blocks use lexical scope: block body uses environment where block was defined

Examples:

```
3.times { puts "hi" }
[4,6,8].each { puts "hi" }
i = 7
[4,6,8].each {|x| if i > x then puts (x+1) end }
```

Some strange things

- Can pass 0 or 1 block with *any* message
  - Callee might ignore it
  - Callee might give an error if you do not send one
  - Callee might do different things if you do/don't send one
    - Also number-of-block-arguments can matter

- Just put the block "next to" the "other" arguments (if any)
  - Syntax: **`{e}`**, **`{|x| e}`**, **`{|x,y| e}`**, etc. (plus variations)
    - Can also replace **`{`** and **`}`** with **do** and **end**
      - Often preferred for blocks > 1 line

Blocks everywhere

- Rampant use of great block-taking methods in standard libraray
- Ruby has loops but very rarely used
  - Can write **`(0..i).each {|j| e}`**, but often better options
- Examples (consult documentation for many more)

```
a = Array.new(5) {|i| 4*(i+1)}
a.each { puts "hi" }
a.each {|x| puts (x * 2) }
a.map  {|x| x * 2 } #synonym: collect
a.any? {|x| x > 7 }
a.all? {|x| x > 7 }
a.inject(0) {|acc,elt| acc+elt }
a.select {|x| x > 7 } #non-synonym: filter
```

More strangeness

- Callee does not give a name to the (potential) block argument

- Instead, just calls it with **`yield`** or **`yield(args)`**
  - Silly example:

```
def silly a
  (yield a) + (yield 42)
end
```

```
x.silly 5 { |b| b*2 }
```

  - See code for slightly less silly example

- Can ask **`block_given?`** but often just assume a block is given or that a block's presence is implied by other arguments

Blocks are "second-class"

All a method can do with a block is **`yield`** to it

- Cannot return it, store it in an object (e.g., for a callback), …
- But can also turn blocks into real closures
- Closures are instances of class **`Proc`**
  - Called with method **`call`**

This is Ruby, so there are several ways to make **`Proc`** objects ☺

- One way: method **`lambda`** of **`Object`** takes a block and returns the corresponding **`Proc`**

Example

```
a = [3,5,7,9]
```

- Blocks are fine for applying to array elements

```
b = a.map {|x| x+1 }
i = b.count {|x| x>=6 }
```

- But for an array of closures, need **`Proc`** objects

```
c = a.map {|x| lambda {|y| x>=y}}
c[2].call 17
j = c.count {|x| x.call(5) }
```

  - More common use is callbacks

Moral

- First-class ("can be passed/stored anywhere") makes closures more powerful than blocks
- But blocks are (a little) more convenient and cover most uses
- This helps us understand what first-class means
- Language design question: When is convenience worth making something less general and powerful?

More collections

- *Hashes* like arrays but:
  - *Keys* can be *anything*; strings and symbols common
  - No natural ordering like numeric indices
  - Different syntax to make them

  Like a dynamic record with anything for field names
  - Often pass a hash rather than many arguments

- *Ranges* like arrays of contiguous numbers but:
  - More efficiently represented, so large ranges fine

Good style to:

  - Use ranges when you can
  - Use hashes when non-numeric keys better represent data

Similar methods

- Arrays, hashes, and ranges all have some methods other don't
  - E.g., **`keys`** and **`values`**

- But also have many of the same methods, particularly iterators
  - Great for duck typing
  - Example

```
def foo a
  a.count {|x| x*x < 50}
end

foo [3,5,7,9]
foo (3..9)
```

Once again separating "how to iterate" from "what to do"

Next major topic

- Subclasses, inheritance, and overriding
  - The essence of OOP
  - Not unlike you have seen in Java, but worth studying from PL perspective and in a more dynamic language

Subclassing

- A class definition has a *superclass* (**`Object`** if not specified)

```
class ColorPoint < Point  …
```

- The superclass affects the class definition:
  - Class *inherits* all method definitions from superclass
  - But class can *override* method definitions as desired

- Unlike Java/C#/C++:
  - No such thing as “inheriting fields” since all objects create instance variables by assigning to them
  - Subclassing has nothing to do with a (non-existent) type system: can still (try to) call any method on any object

Example (to be continued)

```
class Point
  attr_accessor  :x, :y
  def initialize(x,y)
    @x = x
    @y = y
  end
  def distFromOrigin
    # direct field access
    Math.sqrt(@x*@x
              + @y*@y)
  end
  def distFromOrigin2
    # use getters
    Math.sqrt(x*x
              + y*y)
  end
end
```

```
class ColorPoint < Point
  attr_accessor  :color
  def initialize(x,y,c)
    super(x,y)
    @color = c
  end
end
```

An object has a class

```
p  = Point.new(0,0)
cp = ColorPoint.new(0,0,"red")
p.class                          # Point
p.class.superclass               # Object
cp.class                         # ColorPoint
cp.class.superclass              # Point
cp.class.superclass.superclass   # Object
cp.is_a? Point                   # true
cp.instance_of? Point            # false
cp.is_a? ColorPoint              # true
cp.instance_of? ColorPoint       # true
```

- Using these methods is usually non-OOP style
  - Disallows other things that “act like a duck”
  - Nonetheless semantics is that an instance of **`ColorPoint`** “is a” **`Point`** but is not an “instance of” **`Point`**
  - [Java note: **`instanceof`** is like Ruby's **`is_a?`**]

Example continued

- Consider alternatives to:

```
class ColorPoint < Point
  attr_accessor  :color
  def initialize(x,y,c)
    super(x,y)
    @color = c
  end
end
```

- Here subclassing is a good choice, but programmers often overuse subclassing in OOP languages

Why subclass

- Instead of creating **`ColorPoint`**, could add methods to **`Point`**
  - That could mess up other users and subclassers of **`Point`**

```
class Point
  attr_accessor :color
  def initialize(x,y,c="clear")
    @x = x
    @y = y
    @color = c
  end
end
```

Why subclass

- Instead of subclassing **`Point`**, could copy/paste the methods
  - Means the same thing *if* you don't use methods like **`is_a?`** and **`superclass`**, but of course code reuse is nice

```
class ColorPoint
  attr_accessor :x, :y, :color
  def initialize(x,y,c="clear")
    …
  end
  def distFromOrigin
    Math.sqrt(@x*@x + @y*@y)
  end
  def distFromOrigin2
    Math.sqrt(x*x + y*y)
  end
end
```

Why subclass

- Instead of subclassing **`Point`**, could use a **`Point`** instance variable
  - Define methods to send same message to the **`Point`**
  - Often OOP programmers overuse subclassing
  - But for **`ColorPoint`**, subclassing makes sense: less work and can use a **`ColorPoint`** wherever code expects a **`Point`**

```
class ColorPoint
  attr_accessor :color
  def initialize(x,y,c="clear")
    @pt = Point.new(x,y)
    @color = c
  end
  def x
    @pt.x
  end
  … # similar “forwarding” methods
    # for y, x=, y=
end
```

Overriding

- **`ThreeDPoint`** is more interesting than **`ColorPoint`** because it overrides **`distFromOrigin`** and **`distFromOrigin2`**
  - Gets code reuse, but *highly disputable* if it is appropriate to say a **`ThreeDPoint`** "is a" **`Point`**
  - Still just avoiding copy/paste

```
class ThreeDPoint < Point
  …
  def initialize(x,y,z)
    super(x,y)
    @z = z
  end
  def distFromOrigin # distFromOrigin2 similar
    d = super
    Math.sqrt(d*d + @z*@z)
  end
  …
end
```

So far…

- With examples so far, objects are not so different from closures
  - Multiple methods rather than just "call me"
  - Explicit instance variables rather than environment where function is defined
  - Inheritance avoids helper functions or code copying
  - "Simple" overriding just replaces methods

- But there is one big difference:

  Overriding can make a method defined in the superclass call a method in the subclass

  - *The* essential difference of OOP, studied carefully next lecture

Example: Equivalent except constructor

```
class PolarPoint < Point
  def initialize(r,theta)
    @r = r
    @theta = theta
  end
  def x
    @r * Math.cos(@theta)
  end
  def y
    @r * Math.sin(@theta)
  end
  def distFromOrigin
    @r
  end
  …
end
```

- Also need to define **x=** and **y=** (see code file)
- Key punchline: **distFromOrigin2**, defined in **Point**, "already works"

```
def distFromOrigin2
  Math.sqrt(x*x+y*y)
end
```

  - Why: calls to **self** are resolved in terms of the object's class

PAUL G. ALLEN SCHOOL
OF COMPUTER SCIENCE & ENGINEERING

CSE341: Programming Languages

Lecture 21
Dynamic Dispatch Precisely, and Manually in Racket

Dan Grossman

Spring 2019

Dynamic dispatch

Dynamic dispatch

- Also known as *late binding* or *virtual methods*
- Call **`self.m2()`** in method **`m1`** defined in class **`C`** can *resolve to* a method **`m2`** defined in a subclass of **`C`**
- Most unique characteristic of OOP

Need to define the semantics of *method lookup* as carefully as we defined *variable lookup* for our PLs

Review: variable lookup

Rules for “looking things up” is a key part of PL semantics

- ML: Look up *variables* in the appropriate environment
  - Lexical scope for closures
  - *Field names* (for records) are different: not variables
- Racket: Like ML plus **`let`**, **`letrec`**
- Ruby:
  - Local variables and blocks mostly like ML and Racket
  - But also have instance variables, class variables, methods (all more like record fields)
    - Look up in terms of **`self`**, which is special

Using **`self`**

- **`self`** maps to some “current” object
- Look up instance variable **`@x`** using object bound to **`self`**
- Look up class variables **`@@x`** using object bound to **`self.class`**
- Look up methods…

Ruby method lookup

The semantics for method calls also known as message sends

`e0.m(e1,…,en)`

1. Evaluate **`e0`**, **`e1`**, …, **`en`** to objects **`obj0`**, **`obj1`**, …, **`objn`**
   - As usual, may involve looking up **`self`**, variables, fields, etc.
2. Let **`C`** be the class of **`obj0`** (every object has a class)
3. If **`m`** is defined in **`C`**, pick that method, else recur with the superclass of **`C`** unless **`C`** is already **`Object`**
   - If no **`m`** is found, call **`method_missing`** instead
     - Definition of **`method_missing`** in **`Object`** raises an error
4. Evaluate body of method picked:
   - With formal arguments bound to **`obj1`**, …, **`objn`**
   - With **`self`** bound to **`obj0`** -- this implements dynamic dispatch!

Note: Step (3) complicated by *mixins*: will revise definition later

Punch-line again

`e0.m(e1,…,en)`

To implement dynamic dispatch, evaluate the method body with **`self`** mapping to the *receiver* (result of **`e0`**)

- That way, any **`self`** calls in body of **`m`** use the receiver's class,
  - Not necessarily the class that defined **`m`**

- This much is the same in Ruby, Java, C#, Smalltalk, etc.

Comments on dynamic dispatch

- This is why **`distFromOrigin2`** worked in **`PolarPoint`**

- More complicated than the rules for closures
  - Have to treat **`self`** specially
  - May seem simpler only if you learned it first
  - Complicated does not necessarily mean inferior or superior

Static overloading

In Java/C#/C++, method-lookup rules are similar, but more complicated because > 1 methods in a class can have same name

- Java/C/C++: Overriding only when number/types of arguments the same
- Ruby: same-method-name always overriding

Pick the “best one” using the *static* (!) types of the arguments

- Complicated rules for “best”
- Type-checking error if there is no “best”

Relies fundamentally on type-checking rules

- Ruby has none

A simple example, part 1

In ML (and other languages), closures are closed

```
fun even x = if x=0 then true  else odd  (x-1)
and odd  x = if x=0 then false else even (x-1)
```

So we can shadow **odd**, but any call to the closure bound to **odd** above will "do what we expect"

- Does not matter if we shadow **even** or not

```
(* does not change odd – too bad; this would
   improve it *)
fun even x = (x mod 2)=0
```

```
(* does not change odd – good thing; this would
   break it *)
fun even x = false
```

A simple example, part 2

In Ruby (and other OOP languages), subclasses can change the behavior of methods they do not override

```
class A
  def even x
    if x==0 then true  else odd  (x-1) end
  end
  def odd x
    if x==0 then false else even (x-1) end
  end
end
class B < A  # improves odd in B objects
  def even x ; x % 2 == 0 end
end
class C < A  # breaks odd in C objects
  def even x ; false end
end
```

The OOP trade-off

Any method that makes calls to overridable methods can have its behavior changed in subclasses 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 overriding
    - "private" or "final" methods

- So *easier* for subclasses to affect behavior without copying code
  - Provided method in superclass is not modified later

Manual dynamic dispatch

Now: Write Racket code with little more than pairs and functions that *acts like* objects with dynamic dispatch

Why do this?

- (Racket actually has classes and objects available)

- Demonstrates how one language's *semantics* is an idiom in another language
- Understand dynamic dispatch better by coding it up
  - Roughly how an interpreter/compiler might

Analogy: Earlier optional material encoding higher-order functions using objects and explicit environments

Our approach

Many ways to do it; our code does this:

- An "object" has a list of field pairs and a list of method pairs

```
(struct obj (fields methods))
```

- Field-list element example:

```
(mcons 'x 17)
```

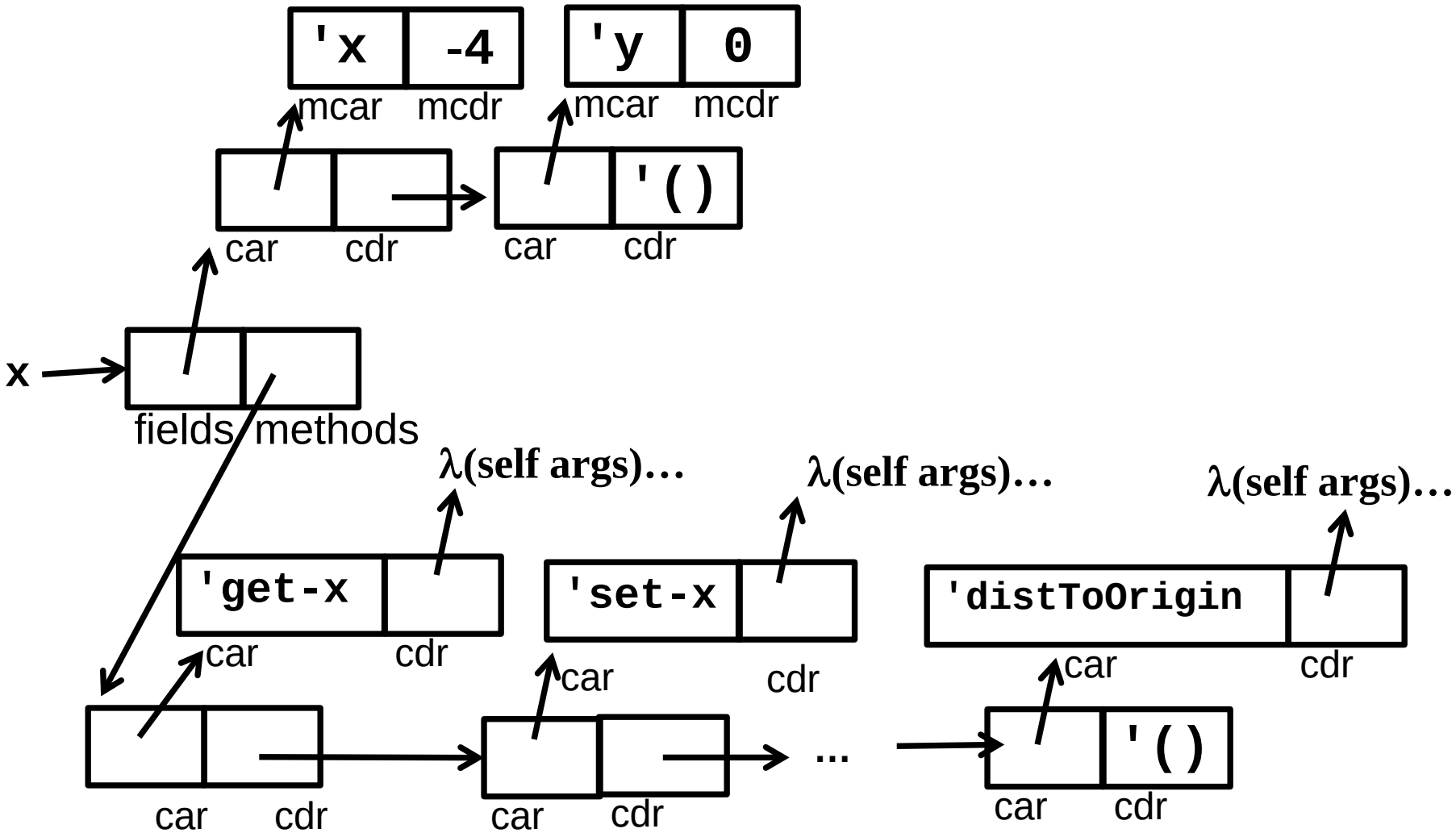
- Method-list element example:

```
(cons 'get-x (lambda (self args) …))
```

Notes:

- Lists sufficient but not efficient
- Not class-based: object has a list of methods, not a class that has a list of methods [could do it that way instead]
- Key trick is lambdas taking an extra **`self`** argument
  - All "regular" arguments put in a list **`args`** for simplicity

A point object bound to **x**

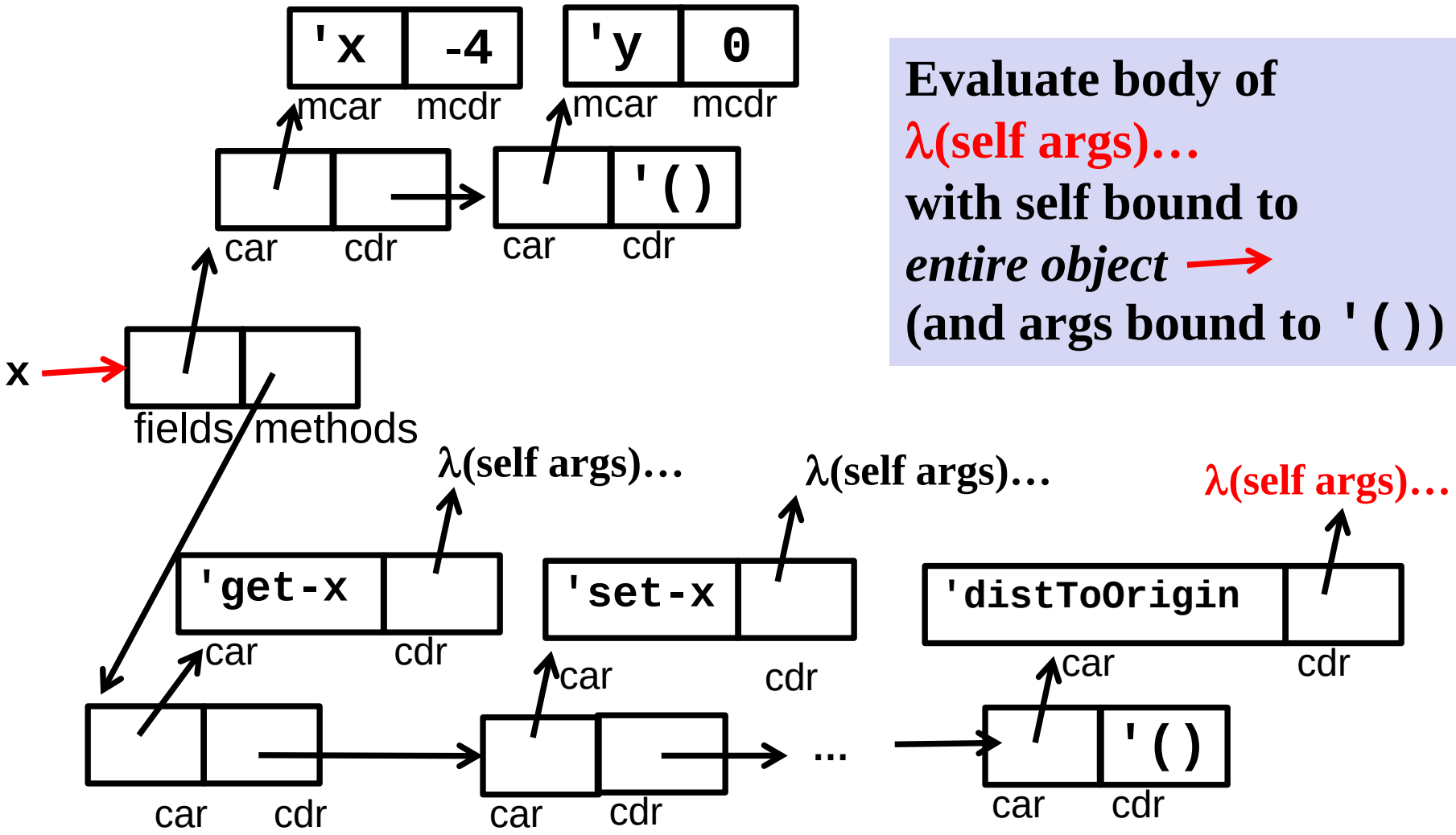


Key helper functions

Now define plain Racket functions to get field, set field, call method

```
(define (assoc-m v xs)
   …) ; assoc for list of mutable pairs

(define (get obj fld)
   (let ([pr (assoc-m fld (obj-fields obj))]))
      (if pr (mcdr pr) (error …))))

(define (set obj fld v)
   (let ([pr (assoc-m fld (obj-fields obj))]))
      (if pr (set-mcdr! pr v) (error …))))

(define (send obj msg . args)
   (let ([pr (assoc msg (obj-methods obj))]))
      (if pr ((cdr pr) obj args) (error …))))
```

(send x 'distToOrigin)



Evaluate body of λ(self args)… with self bound to *entire object* → (and args bound to '())

Constructing points

- Plain-old Racket function can take initial field values and build a point object
  - Use functions **`get`**, **`set`**, and **`send`** on result and in "methods"
  - Call to self: **`(send self 'm …)`**
  - Method arguments in **`args`** list

```
(define (make-point _x _y)
  (obj
    (list (mcons 'x _x)
          (mcons 'y _y))
    (list (cons 'get-x (λ(self args)(get self 'x)))
          (cons 'get-y (λ(self args)(get self 'y)))
          (cons 'set-x (λ(self args)(…)))
          (cons 'set-y (λ(self args)(…)))
          (cons 'distToOrigin (λ(self args)(…))))))
```

"Subclassing"

- Can use **`make-point`** to write **`make-color-point`** or **`make-polar-point`** functions (see code)

- Build a new object using fields and methods from "super" "constructor"
  - Add new or overriding methods to the *beginning* of the list
    - **`send`** will find the first matching method
  - Since **`send`** passes the entire receiver for **`self`**, dynamic dispatch works as desired

Why not ML?

- We were wise not to try this in ML!
- ML's type system does not have subtyping for declaring a polar-point type that “is also a” point type
  - Workarounds possible (e.g., one type for all objects)
  - Still no good type for those **`self`** arguments to functions
    - Need quite sophisticated type systems to support dynamic dispatch if it is not *built into the language*
- In fairness, languages with subtyping but not generics make it analogously awkward to write generic code

PAUL G. ALLEN SCHOOL
OF COMPUTER SCIENCE & ENGINEERING

CSE341: Programming Languages

Lecture 22
OOP vs. Functional Decomposition; Adding Operators & Variants; Double-Dispatch

Dan Grossman

Spring 2019

Breaking things down

- In functional (and procedural) programming, break programs down into functions that perform some operation
- In object-oriented programming, break programs down into classes that give behavior to some kind of data

This lecture:

  - These two forms of *decomposition* are so exactly opposite that they are two ways of looking at the same "matrix"
  - Which form is "better" is somewhat personal taste, but also depends on how you expect to *change/extend software*
  - For some operations over two (multiple) arguments, functions and pattern-matching are straightforward, but with OOP we can do it with *double dispatch* (multiple dispatch)

The expression example

Well-known and compelling example of a common *pattern*:

- Expressions for a small language
- Different variants of expressions: ints, additions, negations, …
- Different operations to perform: **`eval`**, **`toString`**, **`hasZero`**, …

Leads to a matrix (2D-grid) of variants and operations

- Implementation will involve deciding what "should happen" for each entry in the grid *regardless of the PL*

	`eval`	**`toString`**	**`hasZero`**	**`…`**
`Int`				
`Add`				
`Negate`				
`…`				

Standard approach in ML

	`eval`	`toString`	`hasZero`	`…`
`Int`				
`Add`				
`Negate`				
`…`				

- Define a *datatype*, with one *constructor* for each variant
  - (No need to indicate datatypes if dynamically typed)
- "Fill out the grid" via one function per column
  - Each function has one branch for each column entry
  - Can combine cases (e.g., with wildcard patterns) if multiple entries in column are the same

[See the ML code]

Standard approach in OOP

	`eval`	`toString`	`hasZero`	`…`
`Int`				
`Add`				
`Negate`				
`…`				

- Define a *class*, with one *abstract method* for each operation
  - (No need to indicate abstract methods if dynamically typed)
- Define a *subclass* for each variant
- So "fill out the grid" via one class per row with one method implementation for each grid position
  - Can use a method in the superclass if there is a default for multiple entries in a column

[See the Ruby and Java code]

A big course punchline

	`eval`	`toString`	`hasZero`	`…`
`Int`				
`Add`				
`Negate`				
`…`				

- FP and OOP often 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 is no perfect way since software has many dimensions of structure
  - Tools, IDEs can help with multiple "views" (e.g., rows / columns)

Extensibility

	`eval`	`toString`	`hasZero`	`noNegConstants`
`Int`				
`Add`				
`Negate`				
`Mult`				

- For implementing our grid so far, SML / Racket style usually by column and Ruby / Java style usually by row
- But beyond just style, this decision affects what (unexpected?) software *extensions* need not change old code
- Functions [see ML code]:
  - Easy to add a new operation, e.g., **`noNegConstants`**
  - Adding a new variant, e.g., **`Mult`** requires modifying old functions, but ML type-checker gives a to-do list if original code avoided wildcard patterns

Extensibility

	`eval`	`toString`	`hasZero`	`noNegConstants`
`Int`				
`Add`				
`Negate`				
`Mult`				

- For implementing our grid so far, SML / Racket style usually by column and Ruby / Java style usually by row
- But beyond just style, this decision affects what (unexpected?) software *extensions* are easy and/or do not change old code
- Objects [see Ruby code]:
  - Easy to add a new variant, e.g., **`Mult`**
  - Adding a new operation, e.g., **`noNegConstants`** requires modifying old classes, but Java type-checker gives a to-do list if original code avoided default methods

The other way is possible

- Functions allow new operations and objects allow new variants without modifying existing code *even if they didn't plan for it*
  - Natural result of the decomposition

Optional:

- Functions can support new variants somewhat awkwardly "if they plan ahead"
  - *Not explained here: Can use type constructors to make datatypes extensible and have operations take function arguments to give results for the extensions*

- Objects can support new operations somewhat awkwardly "if they plan ahead"
  - *Not explained here: The popular Visitor Pattern uses the double-dispatch pattern to allow new operations "on the side"*

Thoughts on Extensibility

- Making software extensible is valuable and hard
  - If you know you want new operations, use FP
  - If you know you want new variants, use OOP
  - If both? Languages like Scala try; it's a hard problem
  - Reality: The future is often hard to predict!

- Extensibility is a double-edged sword
  - Code more reusable without being changed later
  - But makes original code more difficult to reason about locally or change later (could break extensions)
  - Often language mechanisms to make code *less* extensible (ML modules hide datatypes; Java's **`final`** prevents subclassing/overriding)

Binary operations

	`eval`	`toString`	`hasZero`	`…`
`Int`				
`Add`				
`Negate`				
`…`				

- Situation is more complicated if an operation is defined over multiple arguments that can have different variants
  - Can arise in original program or after extension

- Function decomposition deals with this much more simply…

Example

To show the issue:

- Include variants **`String`** and **`Rational`**
- (Re)define **`Add`** to work on any pair of **`Int`**, **`String`**, **`Rational`**
  - Concatenation if either argument a **`String`**, else math

Now just defining the addition operation is a *different* 2D grid:

	`Int`	**`String`**	**`Rational`**
`Int`			
`String`			
`Rational`			

ML Approach

Addition is different for most **`Int`**, **`String`**, **`Rational`** combinations

- Run-time error for non-value expressions

Natural approach: pattern-match on the pair of values

- For *commutative* possibilities, can re-call with **`(v2,v1)`**

```
fun add_values (v1,v2) =
  case (v1,v2) of
     (Int i, Int j) => Int (i+j)
   | (Int i, String s) => String (Int.toString i ^ s)
   | (Int i, Rational(j,k)) => Rational (i*k+j,k)
   | (Rational _, Int _) => add_values (v2,v1)
   | … (* 5 more cases (3*3 total): see the code *)

fun eval e =
  case e of
     …
   | Add(e1,e2) => add_values (eval e1, eval e2)
```

Example

To show the issue:

- Include variants **`String`** and **`Rational`**
- (Re)define **`Add`** to work on any pair of **`Int`**, **`String`**, **`Rational`**
  - Concatenation if either argument a **`String`**, else math

Now just defining the addition operation is a *different* 2D grid:

	`Int`	**`String`**	**`Rational`**
`Int`			
`String`			
`Rational`			

Worked just fine with functional decomposition — what about OOP…

What about OOP?

Starts promising:

- Use OOP to call method **`add_values`** to one value with other value as result

```
class Add
  …
  def eval
    e1.eval.add_values e2.eval
  end
end
```

Classes **`Int`**, **`MyString`**, **`MyRational`** then all implement

- Each handling 3 of the 9 cases: "add **`self`** to argument"

```
class Int
  …
  def add_values v
    … # what goes here?
  end
end
```

First try

- This approach is common, but is "not as OOP"
  - *So do not do it on your homework*

```
class Int
  def add_values v
    if v.is_a? Int
       Int.new(v.i + i)
    elsif v.is_a? MyRational
       MyRational.new(v.i+v.j*i,v.j)
    else
       MyString.new(v.s + i.to_s)
  end
end
```

- A "hybrid" style where we used dynamic dispatch on 1 argument and then switched to Racket-style type tests for other argument
  - Definitely not "full OOP"

Another way…

- **`add_values`** method in **`Int`** needs "what kind of thing" **`v`** has
  - Same problem in **`MyRational`** and **`MyString`**

- In OOP, "always" solve this by calling a method on **`v`** instead!

- But now we need to "tell" **`v`** "what kind of thing" **`self`** is
  - We know that!
  - "Tell" **`v`** by calling different methods on **`v`**, passing **`self`**

- Use a "programming trick" (?) called *double-dispatch*…

Double-dispatch "trick"

- **`Int`**, **`MyString`**, and **`MyRational`** each define all of **`addInt`**, **`addString`**, and **`addRational`**
  - For example, **`String`**'s **`addInt`** is for concatenating an integer argument to the string in **`self`**
  - 9 total methods, one for each case of addition

- **`Add`**'s **`eval`** method calls **`e1.eval.add_values e2.eval`**, which dispatches to **`add_values`** in **`Int`**, **`String`**, or **`Rational`**
  - **`Int`**'s **`add_values`**: **`v.addInt self`**
  - **`MyString`**'s **`add_values`**: **`v.addString self`**
  - **`MyRational`**'s **`add_values`**: **`v.addRational self`**

  So **`add_values`** performs "2nd dispatch" to the correct case of 9!

[Definitely see the code]

Why showing you this

- Honestly, partly to belittle full commitment to OOP
- To understand dynamic dispatch via a sophisticated idiom
- Because required for the homework
- To contrast with *multimethods* (optional)

Works in Java too

- In a statically typed language, double-dispatch works fine
  - Just need all the dispatch methods in the type

```
abstract class Value extends Exp {
  abstract Value add_values(Value other);
  abstract Value addInt(Int other);
  abstract Value addString(Strng other);
  abstract Value addRational(Rational other);
}
class Int extends Value { … }
class Strng extends Value { … }
class Rational extends Value { … }
```

[See Java code]

Being Fair

Belittling OOP style for requiring the manual trick of double dispatch is somewhat unfair…

What would work better:

- **`Int`**, **`MyString`**, and **`MyRational`** each define three methods all named **`add_values`**
  - One **`add_values`** takes an **`Int`**, one a **`MyString`**, one a **`MyRational`**
  - So 9 total methods named **`add_values`**
  - **`e1.eval.add_values e2.eval`** picks the right one of the 9 at run-time using the classes of the two arguments
- Such a semantics is called *multimethods* or *multiple dispatch*

Multimethods

General idea:

- Allow multiple methods with same name
- Indicate which ones take instances of which classes
- Use dynamic dispatch on arguments in addition to receiver to pick which method is called

If dynamic dispatch is essence of OOP, this is more OOP

- No need for awkward manual multiple-dispatch

Downside: Interaction with subclassing can produce situations where there is “no clear winner” for which method to call

Ruby: Why not?

Multimethods a bad fit (?) for Ruby because:

- Ruby places no restrictions on what is passed to a method
- Ruby never allows methods with the same name
  - Same name means overriding/replacing

Java/C#/C++: Why not?

- Yes, Java/C#/C++ allow multiple methods with the same name
- No, these language do *not* have multimethods
  - They have *static overloading*
  - Uses static types of arguments to choose the method
    - But of course run-time class of receiver [odd hybrid?]
  - No help in our example, so still code up double-dispatch manually
- Actually, C# 4.0 has a way to get effect of multimethods
- Many other language have multimethods (e.g., Clojure)
  - They are not a new idea

PAUL G. ALLEN SCHOOL
OF COMPUTER SCIENCE & ENGINEERING

CSE341: Programming Languages

Lecture 23
Multiple Inheritance, Mixins, Interfaces, Abstract Methods

Dan Grossman

Spring 2019

What next?

Have used classes for OOP's essence: inheritance, overriding, dynamic dispatch

Now, what if we want to have more than *just 1 superclass*

- *Multiple inheritance*: allow > 1 superclasses
  - Useful but has some problems (see C++)
- Ruby-style *mixins*: 1 superclass; > 1 method providers
  - Often a fine substitute for multiple inheritance and has fewer problems (see also Scala *traits*)
- Java/C#-style *interfaces*: allow > 1 types
  - Mostly irrelevant in a dynamically typed language, but fewer problems

Multiple Inheritance

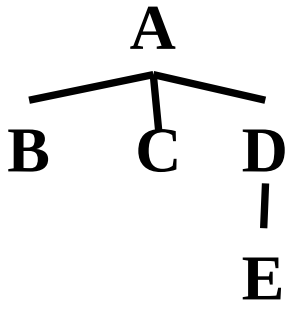
- If inheritance and overriding are so useful, why limit ourselves to one superclass?
  - Because the semantics is often awkward (this topic)
  - Because it makes static type-checking harder (not discussed)
  - Because it makes efficient implementation harder (not discussed)

- Is it useful? Sure!
  - Example: Make a **`ColorPt3D`** by inheriting from **`Pt3D`** and **`ColorPt`** (or maybe just from **`Color`**)
  - Example: Make a **`StudentAthlete`** by inheriting from **`Student`** and **`Athlete`**
  - With single inheritance, end up copying code or using non-OOP-style helper methods

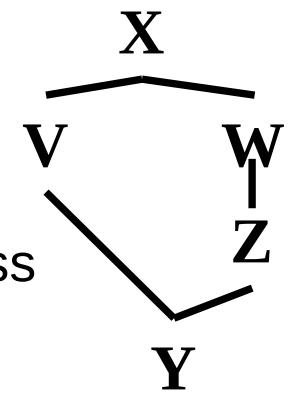
Trees, dags, and diamonds

- Note: The phrases *subclass*, *superclass* can be ambiguous
  - There are *immediate* subclasses, superclasses
  - And there are *transitive* subclasses, superclasses

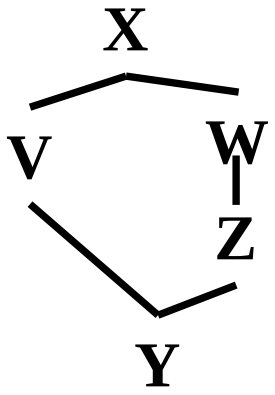
- Single inheritance: the *class hierarchy* is a tree
  - Nodes are classes
  - Parent is immediate superclass
  - Any number of children allowed



- Multiple inheritance: the class hierarchy no longer a tree
  - Cycles still disallowed (a directed-acyclic graph)
  - If multiple paths show that *X* is a (transitive) superclass of *Y*, then we have *diamonds*



What could go wrong?



- If *V* and *Z* both define a method **`m`**,
  what does *Y* inherit? What does **`super`** mean?
  - *Directed resends* useful (e.g., **`Z::super`**)

- What if *X* defines a method **`m`** that *Z* but not *V* overrides?
  - Can handle like previous case, but sometimes undesirable (e.g., **`ColorPt3D`** wants **`Pt3D`**'s overrides to "win")

- If *X* defines fields, should *Y* have one copy of them (**`f`**) or two (**`V::f`** and **`Z::f`**)?
  - Turns out each behavior can be desirable (next slides)
  - So C++ has (at least) two forms of inheritance

3DColorPoints

If Ruby had multiple inheritance, we would want **`ColorPt3D`** to inherit methods that share one **`@x`** and one **`@y`**

```
class Pt
  attr_accessor :x, :y
  …
end
class ColorPt < Pt
 attr_accessor :color
 …
end
class Pt3D < Pt
 attr_accessor :z
 … # override some methods
end
class ColorPt3D < Pt3D, ColorPt # not Ruby!
end
```

ArtistCowboys

This code has **`Person`** define a pocket for subclasses to use, but an **`ArtistCowboy`** wants *two* pockets, one for each **`draw`** method

```
class Person
  attr_accessor :pocket
  …
end
class Artist < Person # pocket for brush objects
 def draw # access pocket
 …
end
class Cowboy < Person # pocket for gun objects
 def draw # access pocket
 …
end
class ArtistCowboy < Artist, Cowboy # not Ruby!
end
```

Mixins

- A *mixin* is (just) a collection of methods
  - Less than a class: no instances of it

- Languages with mixins (e.g., Ruby modules) typically let a class have one superclass but *include* any number of mixins

- Semantics: *Including a mixin makes its methods part of the class*
  - Extending or overriding in the order mixins are included in the class definition
  - More powerful than helper methods because mixin methods can access methods (and instance variables) on **`self`** not defined in the mixin

Example

```
module Doubler
  def double
    self + self # assume included in classes w/ +
  end
end
class String
  include Doubler
end
class AnotherPt
  attr_accessor :x, :y
  include Doubler
  def + other
    ans = AnotherPt.new
    ans.x = self.x + other.x
    ans.y = self.y + other.y
    ans
end
```

Lookup rules

Mixins change our lookup rules slightly:

- When looking for receiver **`obj`**'s method **`m`**, look in **`obj`**'s class, then mixins that class includes (later includes shadow), then **`obj`**'s superclass, then the superclass' mixins, etc.

- As for instance variables, the mixin methods are included in the same object
  - So usually bad style for mixin methods to use instance variables since a name clash would be like our **`CowboyArtist`** pocket problem (but sometimes unavoidable?)

The two big ones

The two most popular/useful mixins in Ruby:

- Comparable: Defines **<**, **>**, **==**, **!=**, **>=**, **<=** in terms of **<=>**
- Enumerable: Defines many iterators (e.g., **`map`**, **`find`**) in terms of **`each`**

Great examples of using mixins:

  - Classes including them get a bunch of methods for just a little work
  - Classes do not "spend" their "one superclass" for this
  - Do not need the complexity of multiple inheritance

- See the code for some examples

Replacement for multiple inheritance?

- A mixin works pretty well for **`ColorPt3D`**:
  - Color a reasonable mixin except for using an instance variable

```
module Color
 attr_accessor :color
end
```

- A mixin works awkwardly-at-best for **`ArtistCowboy`**:
  - Natural for **`Artist`** and **`Cowboy`** to be **`Person`** subclasses
  - Could move methods of one to a mixin, but it is odd style and still does not get you two pockets

```
module ArtistM …
class Artist < Person
  include ArtistM
class ArtistCowboy < Cowboy
  include ArtistM
```

Statically-Typed OOP

- Now contrast multiple inheritance and mixins with Java/C#-style interfaces

- Important distinction, but interfaces are about static typing, which Ruby does not have

- So will use Java code after quick introduction to static typing for class-based OOP…
  - Sound typing for OOP prevents "method missing" errors

Classes as Types

- In Java/C#/etc. each class is also a type

- Methods have types for arguments and result

```
class A {
  Object m1(Example e, String s) {…}
  Integer m2(A foo, Boolean b, Integer i) {…}
}
```

- If **C** is a (transitive) subclass of **D**, then **C** is a *subtype* of **D**
  - Type-checking allows subtype anywhere supertype allowed
  - So can pass instance of **C** to a method expecting instance of **D**

Interfaces are (or were) JustTypes

```
interface Example {
  void   m1(int x, int y);
  Object m2(Example x, String y);
}
```

- An interface is not a class; it is [er, used to be] only a type
  - Does not contain method *definitions*, only their *signatures* (types)
    - Unlike mixins
    - (Changed in Java 8, makes them more like mixins!)
  - Cannot use **new** on an interface
    - Like mixins

Implementing Interfaces

- A class can explicitly implement any number of interfaces
  - For class to type-check, it must implement every method in the interface with the right type
    - More on allowing subtypes later!
  - Multiple interfaces no problem; just implement everything
- If class type-checks, it is a subtype of the interface

```
class A implements Example {
  public void m1(int x, int y) {…}
  public Object m2(Example e, String s) {…}
}
class B implements Example {
  public void m1(int pizza, int beer) {…}
  public Object m2(Example e, String s) {…}
}
```

Multiple interfaces

- Interfaces provide no methods or fields
  - So no questions of method/field duplication when implementing multiple interfaces, unlike multiple inheritance

- What interfaces are for:
  - "Caller can give any instance of any class implementing **`I`**"
    - So callee can call methods in **`I`** regardless of class
  - So much more flexible type system

- Interfaces have little use in a dynamically typed language
  - Dynamic typing *already* much more flexible, with trade-offs we studied

Connections

Let's now answer these questions:

- What does a statically typed OOP language need to support "required overriding"?
- How is this similar to higher-order functions?
- Why does a language with multiple inheritance (e.g., C++) not need Java/C#-style interfaces?

[Explaining Java's abstract methods / C++'s pure virtual methods]

Required overriding

Often a class expects all subclasses to override some method(s)

- The purpose of the superclass is to abstract common functionality, but some non-common parts have no default

A Ruby approach:

- Do not define must-override methods in superclass
- Subclasses can add it
- Creating instance of superclass can cause method-missing errors

```
# do not use A.new
# all subclasses should define m2
class A
  def m1 v
    … self.m2 e …
  end
end
```

Static typing

- In Java/C#/C++, prior approach fails type-checking
  - No method **`m2`** defined in superclass
  - One solution: provide error-causing implementation

```
class A
  def m1 v
    … self.m2 e …
  end
  def m2 v
    raise "must be overridden"
  end
end
```

  - Better: Use static checking to prevent this error…

Abstract methods

- Java/C#/C++ let superclass give signature (type) of method subclasses should provide
  - Called *abstract methods* or *pure virtual methods*
  - Cannot creates instances of classes with such methods
    - Catches error at compile-time
    - Indicates intent to code-reader
    - Does *not* make language more powerful

```
abstract class A {
  T1 m1(T2 x) { … m2(e); … }
  abstract T3 m2(T4 x);
}
class B extends A {
  T3 m2(T4 x) { … }
}
```

Passing code to other code

- Abstract methods and dynamic dispatch: An OOP way to have subclass "pass code" to other code in superclass

```
abstract class A {
  T1 m1(T2 x) { … m2(e); … }
  abstract T3 m2(T4 x);
}
class B extends A {
  T3 m2(T4 x) { … }
}
```

- Higher-order functions: An FP way to have caller "pass code" to callee

```
fun f (g,x) = … g e …

fun h x = … f((fn y => …),…)
```

No interfaces in C++

- If you have multiple inheritance and abstract methods, you do not also need interfaces
- Replace each interface with a class with all abstract methods
- Replace each “implements interface” with another superclass

So: Expect to see interfaces only in statically typed OOP without multiple inheritance

  - Not Ruby
  - Not C++

PAUL G. ALLEN SCHOOL
OF COMPUTER SCIENCE & ENGINEERING

CSE341: Programming Languages

Lecture 24
Subtyping

Dan Grossman

Spring 2019

Last major topic: Subtyping

Build up key ideas from first principles

- In pseudocode because:
  - No time for another language
  - Simpler to first show subtyping without objects

Then:

- How does subtyping relate to types for OOP?
  - Brief sketch only
- What are the relative strengths of subtyping and generics?
- How can subtyping and generics combine synergistically?

A tiny language

- Can cover most core subtyping ideas by just considering *records with mutable fields*

- Will make up our own syntax
  - ML has records, but no subtyping or field-mutation
  - Racket and Ruby have no type system
  - Java uses class/interface names and rarely fits on a slide

Records (half like ML, half like Java)

Record creation (field names and contents):

```
{f1=e1, f2=e2, …, fn=en}
```

Evaluate **`ei`**, make a record

Record field access:

```
e.f
```

Evaluate **`e`** to record **`v`** with an **`f`** field, get contents of **`f`** field

Record field update

```
e1.f = e2
```

Evaluate **`e1`** to a record **`v1`** and **`e2`** to a value **`v2`**;
Change **`v1`**'s **`f`** field (which must exist) to **`v2`**;
Return **`v2`**

A Basic Type System

Record types: What fields a record has and type for each field

```
{f1:t1, f2:t2, …, fn:tn}
```

Type-checking expressions:

- If **`e1`** has type **`t1`**, **…**, **`en`** has type **`tn`**,
  then **`{f1=e1, …, fn=en}`** has type **`{f1:t1, …, fn:tn}`**
- If **`e`** has a record type containing **`f : t`**,
  then **`e.f`** has type **`t`**
- If **`e1`** has a record type containing **`f : t`** and **`e2`** has type **`t`**,
  then **`e1.f = e2`** has type **`t`**

This is sound

These evaluation rules and typing rules prevent ever trying to access a field of a record that does not exist

Example program that type-checks (in a made-up language):

```
fun distToOrigin (p:{x:real,y:real}) =
  Math.sqrt(p.x*p.x + p.y*p.y)

val pythag : {x:real,y:real} = {x=3.0, y=4.0}
val five : real = distToOrigin(pythag)
```

Motivating subtyping

But according to our typing rules, this program does not type-check

- It does nothing wrong and seems worth supporting

```
fun distToOrigin (p:{x:real,y:real}) =
  Math.sqrt(p.x*p.x + p.y*p.y)

val c : {x:real,y:real,color:string} =
   {x=3.0, y=4.0, color="green"}

val five : real = distToOrigin(c)
```

A good idea: allow extra fields

Natural idea: If an expression has type

`{f1:t1, f2:t2, …, fn:tn}`

Then it can *also* have a type with some fields removed

This is what we need to type-check these function calls:

```
fun distToOrigin (p:{x:real,y:real}) = …
fun makePurple (p:{color:string}) =
    p.color = "purple"

val c :{x:real,y:real,color:string} =
   {x=3.0, y=4.0, color="green"}

val _ = distToOrigin(c)
val _ = makePurple(c)
```

Keeping subtyping separate

A programming language already has a lot of typing rules and we do not want to change them

- Example: The type of an actual function argument must ***equal*** the type of the function parameter

We can do this by adding "just two things to our language"

- *Subtyping*: Write **`t1 <: t2`** for **`t1`** is a subtype of **`t2`**
- One new typing rule that uses subtyping:

  If **`e`** has type **`t1`** and **`t1 <: t2`**,

  then **`e`** (also) has type **`t2`**

Now all we need to do is define **`t1 <: t2`**

Subtyping is not a matter of opinion

- Misconception: If we are making a new language, we can have whatever typing and subtyping rules we want

- Not if you want to prevent what you claim to prevent [soundness]
  - Here: No accessing record fields that do not exist

- Our typing rules were *sound* before we added subtyping
  - We should keep it that way

- Principle of *substitutability*: If **`t1 <: t2`**, then any value of type **`t1`** must be usable in every way a **`t2`** is
  - Here: Any value of subtype needs all fields any value of supertype has

Four good rules

For our record types, these rules all meet the substitutability test:

1. "Width" subtyping: A supertype can have a subset of fields with the same types
2. "Permutation" subtyping: A supertype can have the same set of fields with the same types in a different order
3. Transitivity: If **`t1 <: t2`** and **`t2 <: t3`**, then **`t1 <: t3`**
4. Reflexivity: Every type is a subtype of itself

(4) may seem unnecessary, but it composes well with other rules in a full language and "does no harm"

More record subtyping?

[Warning: I am misleading you ☺]

Subtyping rules so far let us drop fields but not change their types

Example: A circle has a center field holding another record

```
fun circleY (c:{center:{x:real,y:real}, r:real}) =
   c.center.y

val sphere:{center:{x:real,y:real,z:real}, r:real} =
  {center={x=3.0,y=4.0,z=0.0}, r=1.0}

val _ = circleY(sphere)
```

For this to type-check, we need:

```
{center:{x:real,y:real,z:real}, r:real}
                    <:
{center:{x:real,y:real}, r:real}
```

Do not have this subtyping – could we?

```
{center:{x:real,y:real,z:real}, r:real}
                    <:
{center:{x:real,y:real}, r:real}
```

- No way to get this yet: we can drop **`center`**, drop **`r`**, or permute order, but cannot "reach into a field type" to do subtyping

- So why not add another subtyping rule… "Depth" subtyping:
  If **`ta <: tb`**, then **`{f1:t1, …, f:ta, …, fn:tn} <: {f1:t1, …, f:tb, …, fn:tn}`**

- Depth subtyping (along with width on the field's type) lets our example type-check

Stop!

- It is nice and all that our new subtyping rule lets our example type-check
- But it is not worth it if it breaks soundness
  - Also allows programs that can access missing record fields
- Unfortunately, it breaks soundness ☹

Mutation strikes again

If **`ta <: tb`**,
then **`{f1:t1, …, f:ta, …, fn:tn} <:`**
`{f1:t1, …, f:tb, …, fn:tn}`

```
fun setToOrigin(c:{center:{x:real,y:real}, r:real})=
   c.center = {x=0.0, y=0.0}

val sphere:{center:{x:real,y:real,z:real}, r:real} =
  {center={x=3.0, y=4.0, z=0.0}, r=1.0}

val _ = setToOrigin(sphere)
val _ = sphere.center.z (* kaboom! (no z field) *)
```

Moral of the story

- In a language with records/objects with getters and setters, depth subtyping is unsound
  - Subtyping cannot change the type of fields

- If fields are immutable, then depth subtyping is sound!
  - Yet another benefit of outlawing mutation!
  - Choose two of three: setters, depth subtyping, soundness

- Remember: subtyping is not a matter of opinion

Picking on Java (and C#)

Arrays should work just like records in terms of depth subtyping

- But in Java, if **`t1 <: t2`**, then **`t1[] <: t2[]`**
- So this code type-checks, surprisingly

```
class Point { … }
class ColorPoint extends Point { … }
…
void m1(Point[] pt_arr) {
  pt_arr[0] = new Point(3,4);
}
String m2(int x) {
  ColorPoint[] cpt_arr = new ColorPoint[x];
  for(int i=0; i < x; i++)
     cpt_arr[i] = new ColorPoint(0,0,"green");
  m1(cpt_arr); // !
  return cpt_arr[0].color; // !
}
```

Why did they do this?

- More flexible type system allows more programs but prevents fewer errors
  - Seemed especially important before Java/C# had generics
- Good news: despite this "inappropriate" depth subtyping
  - **`e.color`** will never fail due to there being no **`color`** field
  - Array *reads* **`e1[e2]`** always return a (subtype of) **`t`** if **`e1`** is a **`t[]`**
- Bad news: to get the good news
  - **`e1[e2]=e3`** can fail even if **`e1`** has type **`t[]`** and **`e3`** has type **`t`**
  - Array *stores* check the *run-time class* of **`e1`**'s elements and do not allow storing a supertype
  - No type-system help to avoid such bugs / performance cost

So what happens

```
void m1(Point[] pt_arr) {
  pt_arr[0] = new Point(3,4); // can throw
}
String m2(int x) {
  ColorPoint[] cpt_arr = new ColorPoint[x];
  …
  m1(cpt_arr); // "inappropriate" depth subtyping
  ColorPoint c = cpt_arr[0]; // fine, cpt_arr
    // will always hold (subtypes of) ColorPoints
  return c.color; // fine, a ColorPoint has a color
}
```

- Causes code in **`m1`** to throw an **`ArrayStoreException`**
  - Even though logical error is in **`m2`**
  - At least run-time checks occur only on array stores, not on field accesses like **`c.color`**

null

- Array stores probably the most *surprising* choice for flexibility over static checking
- But **`null`** is the most *common* one in practice
  - **`null`** is not an object; it has *no* fields or methods
  - But Java and C# let it have *any* object type (backwards, huh?!)
  - So, in fact, we do *not* have the static guarantee that evaluating **`e`** in **`e.f`** or **`e.m(…)`** produces an object that has an **`f`** or **`m`**
  - The "or **`null`**" caveat leads to run-time checks and errors, as you have surely noticed
- Sometimes **`null`** is convenient (like ML's option types)
  - But also having "cannot be **`null`**" types would be nice

Now functions

- Already know a caller can use subtyping for arguments passed
  - Or on the result
- More interesting: When is one function type a subtype of another?
  - Important for higher-order functions: If a function expects an argument of type **`t1 -> t2`**, can you pass a **`t3 -> t4`** instead?
  - Coming next: Important for understanding methods
    - (An object type is a lot like a record type where “method positions” are immutable and have function types)

Example

```
fun distMoved (f : {x:real,y:real}->{x:real,y:real},
               p : {x:real,y:real}) =
   let val p2 : {x:real,y:real} = f p
       val dx : real = p2.x - p.x
       val dy : real = p2.y - p.y
   in Math.sqrt(dx*dx + dy*dy) end

fun flip p = {x = ~p.x, y=~p.y}
val d = distMoved(flip, {x=3.0, y=4.0})
```

No subtyping here yet:

- **`flip`** has exactly the type **`distMoved`** expects for **`f`**
- Can pass **`distMoved`** a record with extra fields for **`p`**, but that's old news

Return-type subtyping

```
fun distMoved (f : {x:real,y:real}->{x:real,y:real},
               p : {x:real,y:real}) =
   let val p2 : {x:real,y:real} = f p
       val dx : real = p2.x - p.x
       val dy : real = p2.y - p.y
   in Math.sqrt(dx*dx + dy*dy) end

fun flipGreen p = {x = ~p.x, y=~p.y, color="green"}
val d = distMoved(flipGreen, {x=3.0, y=4.0})
```

- Return type of **`flipGreen`** is **`{x:real,y:real,color:string}`**, but **`distMoved`** expects a return type of **`{x:real,y:real}`**
- Nothing goes wrong: If **`ta <: tb`**, then **`t->ta <: t->tb`**
  - A function can return "*more* than it needs to"
  - Jargon: "Return types are *covariant*"

This is wrong

```
fun distMoved (f : {x:real,y:real}->{x:real,y:real},
               p : {x:real,y:real}) =
   let val p2 : {x:real,y:real} = f p
       val dx : real = p2.x - p.x
       val dy : real = p2.y - p.y
   in Math.sqrt(dx*dx + dy*dy) end

fun flipIfGreen p = if p.color = "green" (*kaboom!*)
                    then {x = ~p.x, y=~p.y}
                    else {x = p.x, y=p.y}
val d = distMoved(flipIfGreen, {x=3.0, y=4.0})
```

- Argument type of **`flipIfGreen`** is **`{x:real,y:real,color:string}`**, but it is called with a **`{x:real,y:real}`**
- Unsound! **`ta <: tb`** does **NOT** allow **`ta -> t <: tb -> t`**

The other way works!

```
fun distMoved (f : {x:real,y:real}->{x:real,y:real},
               p : {x:real,y:real}) =
   let val p2 : {x:real,y:real} = f p
       val dx : real = p2.x - p.x
       val dy : real = p2.y - p.y
   in Math.sqrt(dx*dx + dy*dy) end

fun flipX_Y0 p = {x = ~p.x, y=0.0}
val d = distMoved(flipX_Y0, {x=3.0, y=4.0})
```

- Argument type of **`flipX_Y0`** is **`{x:real}`**, but it is called with a **`{x:real,y:real}`**, which is fine
- If **`tb <: ta`**, then **`ta -> t <: tb -> t`**
  - A function can assume "*less* than it needs to" about arguments
  - Jargon: "Argument types are *contravariant*"

Can do both

```
fun distMoved (f : {x:real,y:real}->{x:real,y:real},
               p : {x:real,y:real}) =
   let val p2 : {x:real,y:real} = f p
       val dx : real = p2.x - p.x
       val dy : real = p2.y - p.y
   in Math.sqrt(dx*dx + dy*dy) end

fun flipXMakeGreen p = {x = ~p.x, y=0.0, color="green"}
val d = distMoved(flipXMakeGreen, {x=3.0, y=4.0})
```

- **`flipXMakeGreen`** has type

  `{x:real} -> {x:real,y:real,color:string}`
- Fine to pass a function of such a type as function of type

  `{x:real,y:real} -> {x:real,y:real}`
- If **`t3 <: t1`** and **`t2 <: t4`**, then **`t1 -> t2 <: t3 -> t4`**

Conclusion

- If `t3 <: t1` and `t2 <: t4`, then `t1 -> t2 <: t3 -> t4`
  - Function subtyping contravariant in argument(s) and covariant in results
- Also essential for understanding subtyping and methods in OOP
- Most unintuitive concept in the course
  - Smart people often forget and convince themselves covariant arguments are okay
  - These people are always mistaken
  - At times, you or your boss or your friend may do this
  - Remember: A guy with a PhD in PL ***jumped up and down*** insisting that function/method subtyping is always contravariant in its argument -- covariant is unsound

PAUL G. ALLEN SCHOOL
OF COMPUTER SCIENCE & ENGINEERING

CSE341: Programming Languages

Lecture 25
Subtyping for OOP; Comparing/Combining Generics and Subtyping

Dan Grossman

Spring 2019

Now…

Use what we learned about subtyping for records and functions to understand subtyping for class-based OOP

- Like in Java/C#

Recall:

- Class names are also types
- Subclasses are also subtypes
- Substitution principle: Instance of subclass should usable in place of instance of superclass

An object is…

- Objects: mostly records holding fields and methods
  - Fields are mutable
  - Methods are immutable functions that also have access to **`self`**

- So *could* design a type system using types very much like record types
  - Subtypes could have extra fields and methods
  - Overriding methods could have contravariant arguments and covariant results compared to method overridden
    - Sound only because method "slots" are immutable!

Actual Java/C#…

Compare/contrast to what our "theory" allows:

1. Types are class names and subtyping are explicit subclasses
2. A subclass can add fields and methods
3. A subclass can override a method with a covariant return type
   - (No contravariant arguments; instead makes it a non-overriding method of the same name)

(1) Is a subset of what is sound (so also sound)

(3) Is a subset of what is sound and a different choice (adding method instead of overriding)

Classes vs. Types

- A class defines an object's behavior
  - Subclassing inherits behavior and changes it via extension and overriding

- A type describes an object's methods' argument/result types
  - A subtype is substitutable in terms of its field/method types

- These are separate concepts: try to use the terms correctly
  - Java/C# confuse them by requiring subclasses to be subtypes
  - A class name is both a class and a type
  - Confusion is convenient in practice

Optional: More details

Java and C# are sound: They do not allow subtypes to do things that would lead to “method missing” or accessing a field at the wrong type

Confusing (?) Java example:

- Subclass can declare field name already declared by superclass
- Two classes can use any two types for the field name
- Instances of subclass have two fields with same name
- “Which field is in scope” depends on which class defined the method

`self/this` *is special*

- Recall our Racket encoding of OOP-style
  - "Objects" have a list of fields and a list of functions that take **`self`** as an explicit extra argument

- So if **`self/this`** is a function argument, is it contravariant?
  - No, it is *covariant*: a method in a subclass can use fields and methods only available in the subclass: essential for OOP

```
class A {
  int m(){ return 0; }
}
class B extends A {
  int x;
  int m(){ return x; }
}
```

  - Sound because calls always use the "whole object" for **`self`**
  - This is why coding up your own objects manually works much less well in a statically typed languages

What are generics good for?

Some good uses for parametric polymorphism:

- Types for functions that combine other functions:

```
fun compose (g,h) = fn x => g (h x)
(* compose : ('b -> 'c) * ('a -> 'b) -> ('a -> 'c) *)
```

- Types for functions that operate over generic collections

```
val length : 'a list -> int
val map : ('a -> 'b) -> 'a list -> 'b list
val swap : ('a * 'b) -> ('b * 'a)
```

- Many other idioms
- General point: When types can “be anything” but multiple things need to be “the same type”

Generics in Java

- Java generics a bit clumsier syntactically and semantically, but can express the same ideas
  - Without closures, often need to use (one-method) objects
  - See also earlier optional lecture on closures in Java/C
- Simple example without higher-order functions (optional):

```
class Pair<T1,T2> {
  T1 x;
  T2 y;
  Pair(T1 _x, T2 _y){ x = _x; y = _y; }
  Pair<T2,T1> swap() {
     return new Pair<T2,T1>(y,x);
  }
  …
}
```

Subtyping is not good for this

- Using subtyping for containers is much more painful for clients
  - Have to downcast items retrieved from containers
  - Downcasting has run-time cost
  - Downcasting can fail: no static check that container holds the type of data you expect
  - (Only gets more painful with higher-order functions like **`map`**)

```
class LamePair {
  Object x;
  Object y;
  LamePair(Object _x, Object _y){ x=_x; y=_y; }
  LamePair swap() { return new LamePair(y,x); }
}

// error caught only at run-time:
String s = (String)(new LamePair("hi",4).y);
```

What is subtyping good for?

Some good uses for subtype polymorphism:

- Code that “needs a Foo” but fine to have “more than a Foo”
- Geometry on points works fine for colored points
- GUI widgets specialize the basic idea of “being on the screen” and “responding to user actions”

Awkward in ML

ML does not have subtyping, so this simply does not type-check:

```
(* {x:real, y:real} -> real *)
fun distToOrigin ({x=x,y=y}) =
   Math.sqrt(x*x + y*y)

val five = distToOrigin {x=3.0,y=4.0,color="red"}
```

Cumbersome workaround: have caller pass in getter functions:

```
(* ('a -> real) * ('a -> real) * 'a -> real *)
fun distToOrigin (getx, gety, v) =
   Math.sqrt((getx v)*(getx v)
             + (gety v)*(gety v))
```

- And clients still need different getters for points, color-points

Wanting both

- Could a language have generics and subtyping?
  - Sure!

- More interestingly, want to combine them
  - "Any type **T1** that is a subtype of **T2**"
  - Called bounded polymorphism
  - Lets you do things naturally you cannot do with generics or subtyping separately

Example

Method that takes a list of points and a circle (center point, radius)

- Return new list of points in argument list that lie within circle

Basic method signature:

```
List<Point> inCircle(List<Point> pts,
                     Point center,
                     double r) { … }
```

Java implementation straightforward assuming **`Point`** has a **`distance`** method:

```
List<Point> result = new ArrayList<Point>();
for(Point pt : pts)
  if(pt.distance(center) < r)
    result.add(pt);
return result;
```

Subtyping?

```
List<Point> inCircle(List<Point> pts,
                     Point center,
                     double r) { … }
```

- Would like to use **`inCircle`** by passing a **`List<ColorPoint>`** and getting back a **`List<ColorPoint>`**

- Java rightly disallows this: While **`inCircle`** would "do nothing wrong" its type does not prevent:
  - Returning a list that has a non-color-point in it
  - Modifying **`pts`** by adding non-color-points to it

Generics?

```
List<Point> inCircle(List<Point> pts,
                     Point center,
                     double r) { … }
```

- We could change the method to be

```
<T> List<T> inCircle(List<T> pts,
                     Point center,
                     double r) { … }
```

  - Now the type system allows passing in a **`List<Point>`** to get a **`List<Point>`** returned or a **`List<ColorPoint>`** to get a **`List<ColorPoint>`** returned
  - But cannot implement **`inCircle`** properly: method body should have *no* knowledge of type **`T`**

Bounds

- What we want:

```
<T> List<T> inCircle(List<T> pts,
                     Point center,
                     double r) where T <: Point
 { … }
```

- Caller uses it generically, but must instantiate **`T`** with some subtype of **`Point`** (including **`Point`**)
- Callee can assume **`T <: Point`** so it can do its job
- Callee must return a **`List<T>`** so output will contain only elements from **`pts`**

Real Java

- The actual Java syntax:

```
<T extends Pt> List<T> inCircle(List<T> pts,
                                Pt center,
                                double r) {
   List<T> result = new ArrayList<T>();
   for(T pt : pts)
     if(pt.distance(center) < r)
       result.add(pt);
   return result;
}
```

- Note: For backward-compatibility and implementation reasons, in Java there is actually always a way to use casts to get around the static checking with generics ☹
  - With or without bounded polymorphism