

```
fun append (xs,ys) =  
  if xs=[]  
  then ys  
  else (hd xs)::append(tl xs,ys)  
  
fun map (f,xs) =  
  case xs of  
    [] => []  
  | x::xs' => (f x)::(map(f,xs'))  
  
val a = map (increment, [4,8,12,16])  
val b = map (hd, [[8,6],[7,5],[3,0,9]])
```

Programming Languages

Dan Grossman

Ways to Build New Types

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`

Coming soon

- 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

```
fun append (xs,ys) =  
  if xs=[]  
  then ys  
  else (hd xs)::append(tl xs,ys)  
  
fun map (f,xs) =  
  case xs of  
    [] => []  
  | x::xs' => (f x)::(map(f,xs'))  
  
val a = map (increment, [4,8,12,16])  
val b = map (hd, [[8,6],[7,5],[3,0,9]])
```

Programming Languages

Dan Grossman

Records

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 pieces:

```
#myfieldname e
```

(Evaluation rules and type-checking as expected)

Example

```
{name = "Amelia", id = 41123 - 12}
```

Evaluates to

```
{id = 41111, name = "Amelia"}
```

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 do it differently; some languages have


```
fun append (xs,ys) =  
  if xs=[]  
  then ys  
  else (hd xs)::append(tl xs,ys)  
  
fun map (f,xs) =  
  case xs of  
    [] => []  
  | x::xs' => (f x)::(map(f,xs'))  
  
val a = map (increment, [4,8,12,16])  
val b = map (hd, [[8,6],[7,5],[3,0,9]])
```

Programming Languages

Dan Grossman

Tuples as Syntactic Sugar

The truth about tuples

Previously, we 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
- (e_1, \dots, e_n) is another way of writing $\{1=e_1, \dots, n=e_n\}$
- $t_1 * \dots * t_n$ is another way of writing $\{1:t_1, \dots, n:t_n\}$
- 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**

```
fun append (xs,ys) =  
  if xs=[]  
  then ys  
  else (hd xs)::append(tl xs,ys)  
  
fun map (f,xs) =  
  case xs of  
    [] => []  
  | x::xs' => (f x)::(map(f,xs'))  
  
val a = map (increment, [4,8,12,16])  
val b = map (hd, [[8,6],[7,5],[3,0,9]])
```

Programming Languages

Dan Grossman

Datatype Bindings

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

```
fun append (xs,ys) =  
  if xs=[]  
  then ys  
  else (hd xs)::append(tl xs,ys)  
  
fun map (f,xs) =  
  case xs of  
    [] => []  
  | x::xs' => (f x)::(map(f,xs'))  
  
val a = map (increment, [4,8,12,16])  
val b = map (hd, [[8,6],[7,5],[3,0,9]])
```

Programming Languages

Dan Grossman

Case Expressions

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 (soon!)

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

```
fun append (xs,ys) =  
  if xs=[]  
  then ys  
  else (hd xs)::append(tl xs,ys)  
  
fun map (f,xs) =  
  case xs of  
    [] => []  
  | x::xs' => (f x)::(map(f,xs'))  
  
val a = map (increment, [4,8,12,16])  
val b = map (hd, [[8,6],[7,5],[3,0,9]])
```

Programming Languages

Dan Grossman

Useful Datatypes

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 rank = 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 student_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 it 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

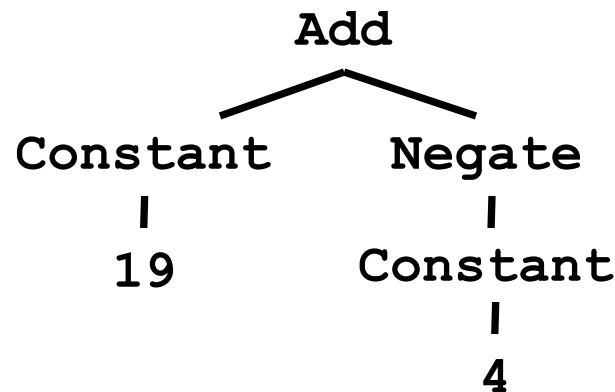
A more exciting (?) example of a datatype, using self-reference

```
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)
```

```
fun append (xs,ys) =  
  if xs=[]  
  then ys  
  else (hd xs)::append(tl xs,ys)  
  
fun map (f,xs) =  
  case xs of  
    [] => []  
  | x::xs' => (f x)::(map(f,xs'))  
  
val a = map (increment, [4,8,12,16])  
val b = map (hd, [[8,6],[7,5],[3,0,9]])
```

Programming Languages

Dan Grossman

Pattern-Matching So Far: Precisely

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 C_i of type $t_i \rightarrow t$

- $C_i \ v$ is a value, i.e., the result “includes the tag”

Omit “of t ” for constructors that are just tags, no underlying data

- Such a C_i 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 **p_i** is the first *pattern* to *match* **v**, then result is evaluation of **e_i** in environment “extended by the match”
- Pattern **C_i (x₁ , ... , x_n)** matches value **C_i (v₁ , ... , v_n)** and extends the environment with **x₁** to **v₁** ... **x_n** to **v_n**
 - For “no data” constructors, pattern **C_i** matches value **C_i**

```
fun append (xs,ys) =  
  if xs=[]  
  then ys  
  else (hd xs)::append(tl xs,ys)  
  
fun map (f,xs) =  
  case xs of  
    [] => []  
  | x::xs' => (f x)::(map(f,xs'))  
  
val a = map (increment, [4,8,12,16])  
val b = map (hd, [[8,6],[7,5],[3,0,9]])
```

Programming Languages

Dan Grossman

Another Expression Example

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 `.sm1` file...

```
fun append (xs,ys) =  
  if xs=[]  
  then ys  
  else (hd xs)::append(tl xs,ys)  
  
fun map (f,xs) =  
  case xs of  
    [] => []  
  | x::xs' => (f x)::(map(f,xs'))  
  
val a = map (increment, [4,8,12,16])  
val b = map (hd, [[8,6],[7,5],[3,0,9]])
```

Programming Languages

Dan Grossman

Type Synonyms

Creating new types

- A *datatype binding* introduces a new type name
 - Distinct from all existing types
 - Only way to create values of the new type is the constructors
- A *type synonym* is a new kind of binding

```
type aname = t
```

- Just creates another name for a type
- The type and the name are *interchangeable in every way*
- Do not worry about what REPL prints: picks what it wants just like it picks the order of record field names

Why have this?

For now, type synonyms just a convenience for talking about types

- Example (where **suit** and **rank** already defined):

type card = suit * rank

- Write a function of type

card -> bool

- Okay if REPL says your function has type

suit * rank -> bool

Convenient, but does not let us “do” anything new

Later in course will see another use related to modularity

```
fun append (xs,ys) =  
  if xs=[]  
  then ys  
  else (hd xs)::append(tl xs,ys)  
  
fun map (f,xs) =  
  case xs of  
    [] => []  
  | x::xs' => (f x)::(map(f,xs'))  
  
val a = map (increment, [4,8,12,16])  
val b = map (hd, [[8,6],[7,5],[3,0,9]])
```

Programming Languages

Dan Grossman

Lists and Options are Datatypes

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 **valueOf**

```
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

```
fun append (xs,ys) =  
  if xs=[]  
  then ys  
  else (hd xs)::append(tl xs,ys)  
  
fun map (f,xs) =  
  case xs of  
    [] => []  
  | x::xs' => (f x)::(map(f,xs'))  
  
val a = map (increment, [4,8,12,16])  
val b = map (hd, [[8,6],[7,5],[3,0,9]])
```

Programming Languages

Dan Grossman

Polymorphic Datatypes

Finish the story

- Claimed built-in options and lists are not needed/special
 - Other than special syntax for list constructors
- But these datatype bindings are polymorphic type constructors
 - `int list` and `string list` and `int list list` are all types, not `list`
 - Functions might or might not be polymorphic
 - `val sum_list : int list -> int`
 - `val append : 'a list * 'a list -> 'a list`
- Good language design: Can define new polymorphic datatypes
- Semi-optional: Do *not* need to understand this for homework 2

Defining polymorphic datatypes

- Syntax: put one or more type variables before datatype name

```
datatype 'a option = NONE | SOME of 'a
```

```
datatype 'a mylist = Empty | Cons of 'a * 'a mylist
```

```
datatype ('a,'b) tree =  
    Node of 'a * ('a,'b) tree * ('a,'b) tree  
    | Leaf of 'b
```

- Can use these type variables in constructor definitions
- Binding then introduces a type constructor, not a type
 - Must say `int mylist` or `string mylist` or `'a mylist`
 - Not “plain” `mylist`

Nothing else changes

Use constructors and case expressions as usual

- No change to evaluation rules
- Type-checking will make sure types are used consistently
 - Example: cannot mix element types of list
- Functions will be polymorphic or not based on how data is used

```
fun append (xs,ys) =  
  if xs=[]  
  then ys  
  else (hd xs)::append(tl xs,ys)  
  
fun map (f,xs) =  
  case xs of  
    [] => []  
  | x::xs' => (f x)::(map(f,xs'))  
  
val a = map (increment, [4,8,12,16])  
val b = map (hd, [[8,6],[7,5],[3,0,9]])
```

Programming Languages

Dan Grossman

Pattern-Matching for Each-Of Types: The Truth
About Function Arguments

An exciting segment

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 **(*x*₁, ..., *x*_n)**
matches the tuple value **(*v*₁, ..., *v*_n)**
- The pattern **{*f*₁=*x*₁, ..., *f*_n=*x*_n}**
matches the record value **{*f*₁=*v*₁, ..., *f*_n=*v*_n}**
(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 `()`

```
fun append (xs,ys) =  
  if xs=[]  
  then ys  
  else (hd xs)::append(tl xs,ys)  
  
fun map (f,xs) =  
  case xs of  
    [] => []  
  | x::xs' => (f x)::(map(f,xs'))  
  
val a = map (increment, [4,8,12,16])  
val b = map (hd, [[8,6],[7,5],[3,0,9]])
```

Programming Languages

Dan Grossman

A Little Type Inference

A new way to go

- For homework 2:
 - Do not use the `#` character
 - Do not need to write down any explicit types
- These are related
 - Type-checker can use patterns to figure out the types
 - With just `#foo` or `#1` it cannot determine “what other fields”

Why no problem

Easy for type-checker to determine function types:

```
fun sum_triple (x, y, z) =  
  x + y + z  
  
fun full_name {first=x, middle=y, last=z} =  
  x ^ " " ^ y ^ " " ^ z
```

Get error message without explicit type annotation:

```
fun sum_triple (triple : int*int*int) =  
  #1 triple + #2 triple + #3 triple  
  
fun full_name (r : {first:string, middle:string,  
                    last:string}) =  
  #first r ^ " " ^ #middle r ^ " " ^ #last r
```


Unexpected polymorphism

- Sometimes type-checker is “smarter than you expect”
 - Types of some parts might be less constrained than you think
 - Example: If you do not use something it can have any type

```
(* int * 'a * int -> int *)  
fun partial_sum (x, y, z) =  
    x + z  
  
(*{first:string, last:string, middle:'a} -> string*)  
fun partial_name {first=x, middle=y, last=z} =  
    x ^ " " ^ z
```

- This is okay!
 - A more general type than you need is always acceptable
 - Assuming your function is correct, of course
 - More precise definition of “more general type” next segment

```
fun append (xs,ys) =  
  if xs=[]  
  then ys  
  else (hd xs)::append(tl xs,ys)  
  
fun map (f,xs) =  
  case xs of  
    [] => []  
  | x::xs' => (f x)::(map(f,xs'))  
  
val a = map (increment, [4,8,12,16])  
val b = map (hd, [[8,6],[7,5],[3,0,9]])
```

Programming Languages

Dan Grossman

Polymorphic Types and Equality Types

An example

- “Write a function that appends two string lists”

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

- You expect `string list * string list -> string list`
- Implementation says `'a list * 'a list -> 'a list`
- This is okay [such as on your homework]: why?

More general

The type

```
'a list * 'a list -> 'a list
```

is **more general** than the type

```
string list * string list -> string list
```

- It “can be used” as **any less general type**, such as

```
int list * int list -> int list
```

- But it is **not** more general than the type

```
int list * string list -> int list
```

The “more general” rule

Easy rule you (and the type-checker) can apply without thinking:

A type $t1$ is **more general** than the type $t2$ if you can take $t1$,
replace its type variables consistently, and get $t2$

- Example: Replace each '**a**' with **int * int**
- Example: Replace each '**a**' with **bool** and each '**b**' with **bool**
- Example: Replace each '**a**' with **bool** and each '**b**' with **int**
- Example: Replace each '**b**' with '**a**' and each '**a**' with '**a**'

Other rules

- Can combine the “more general” rule with rules for equivalence
 - Use of type synonyms does not matter
 - Order of field names does not matter

Example, given

```
type foo = int * int
```

the type

```
{quux : 'b, bar : int * 'a, baz : 'b}
```

is more general than

```
{quux : string, bar : foo, baz : string}
```

which is equivalent to

```
{bar : int*int, baz : string, quux : string}
```

Equality types

- You might also see type variables with a second “quote”
 - Example: `' 'a list * ' 'a -> bool`
- These are “equality types” that arise from using the `=` operator
 - The `=` operator works on lots of types: `int`, `string`, tuples containing all equality types, ...
 - But not all types: function types, `real`, ...
- The rules for more general are exactly the same except you have to replace an equality-type variable with a type that can be used with `=`
 - A “strange” feature of ML because `=` is special

Example

```
(* ''a * ''a -> string *)  
fun same_thing(x, y) =  
    if x=y then "yes" else "no"  
  
(* int -> string *)  
fun is_three x =  
    if x=3 then "yes" else "no"
```

(You can ignore the warning about “calling polyEqual”)


```
fun append (xs,ys) =  
  if xs=[]  
  then ys  
  else (hd xs)::append(tl xs,ys)  
  
fun map (f,xs) =  
  case xs of  
    [] => []  
  | x::xs' => (f x)::(map(f,xs'))  
  
val a = map (increment, [4,8,12,16])  
val b = map (hd, [[8,6],[7,5],[3,0,9]])
```

Programming Languages

Dan Grossman

Nested Patterns

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::t11, hd2::t12, hd3::t13) =>  
    (hd1, hd2, hd3) :: zip3(t11, t12, t13)  
  | _ => raise ListLengthMismatch  
  
fun unzip3 triples =  
  case triples of  
    [] => ([], [], [])  
  | (a, b, c) :: t1 =>  
    let val (l1, l2, l3) = unzip3 t1  
    in  
      (a :: l1, b :: l2, c :: l3)  
    end  
end
```

More examples to come (see code files)

```
fun append (xs,ys) =  
  if xs=[]  
  then ys  
  else (hd xs)::append(tl xs,ys)  
  
fun map (f,xs) =  
  case xs of  
    [] => []  
  | x::xs' => (f x)::(map(f,xs'))  
  
val a = map (increment, [4,8,12,16])  
val b = map (hd, [[8,6],[7,5],[3,0,9]])
```

Programming Languages

Dan Grossman

More Nested Patterns

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**

```
fun append (xs,ys) =  
  if xs=[]  
  then ys  
  else (hd xs)::append(tl xs,ys)  
  
fun map (f,xs) =  
  case xs of  
    [] => []  
  | x::xs' => (f x)::(map(f,xs'))  
  
val a = map (increment, [4,8,12,16])  
val b = map (hd, [[8,6],[7,5],[3,0,9]])
```

Programming Languages

Dan Grossman

Nested Patterns Precisely

(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 $(p1, \dots, pn)$ and v is $(v1, \dots, vn)$, 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


```
fun append (xs,ys) =  
  if xs=[]  
  then ys  
  else (hd xs)::append(tl xs,ys)  
  
fun map (f,xs) =  
  case xs of  
    [] => []  
  | x::xs' => (f x)::(map(f,xs'))  
  
val a = map (increment, [4,8,12,16])  
val b = map (hd, [[8,6],[7,5],[3,0,9]])
```

Programming Languages

Dan Grossman

Optional: Function Patterns

Yet more pattern-matching

[Your instructor has never preferred this style, but others like it and you are welcome to use it]

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

fun eval (Constant i) = i
  | eval (Add(e1,e2)) = (eval e1) + (eval e2)
  | eval (Negate e1)  = ~ (eval e1)
  | eval (Multiply(e1,e2)) = (eval e1) + (eval e2)
```

Nothing more powerful

In general

```
fun f x =  
  case x of  
    p1 => e1  
  | p2 => e2  
  ...
```

Can be written as

```
fun f p1 = e1  
  | f p2 = e2  
  ...  
  | f pn = en
```

If you prefer (assuming **x** is not used in any branch)

```
fun append (xs,ys) =  
  if xs=[]  
  then ys  
  else (hd xs)::append(tl xs,ys)  
  
fun map (f,xs) =  
  case xs of  
    [] => []  
  | x::xs' => (f x)::(map(f,xs'))  
  
val a = map (increment, [4,8,12,16])  
val b = map (hd, [[8,6],[7,5],[3,0,9]])
```

Programming Languages

Dan Grossman

Exceptions

Exceptions

An exception binding introduces a new kind of exception

```
exception MyFirstException  
exception MySecondException of int * int
```

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

```
raise MyFirstException  
raise (MySecondException(7,9))
```

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

- If doesn't match, exception continues to propagate

```
e1 handle MyFirstException => e2  
e1 handle MySecondException(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**

```
fun append (xs,ys) =  
  if xs=[]  
  then ys  
  else (hd xs)::append(tl xs,ys)  
  
fun map (f,xs) =  
  case xs of  
    [] => []  
  | x::xs' => (f x)::(map(f,xs'))  
  
val a = map (increment, [4,8,12,16])  
val b = map (hd, [[8,6],[7,5],[3,0,9]])
```

Programming Languages

Dan Grossman

Tail Recursion

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

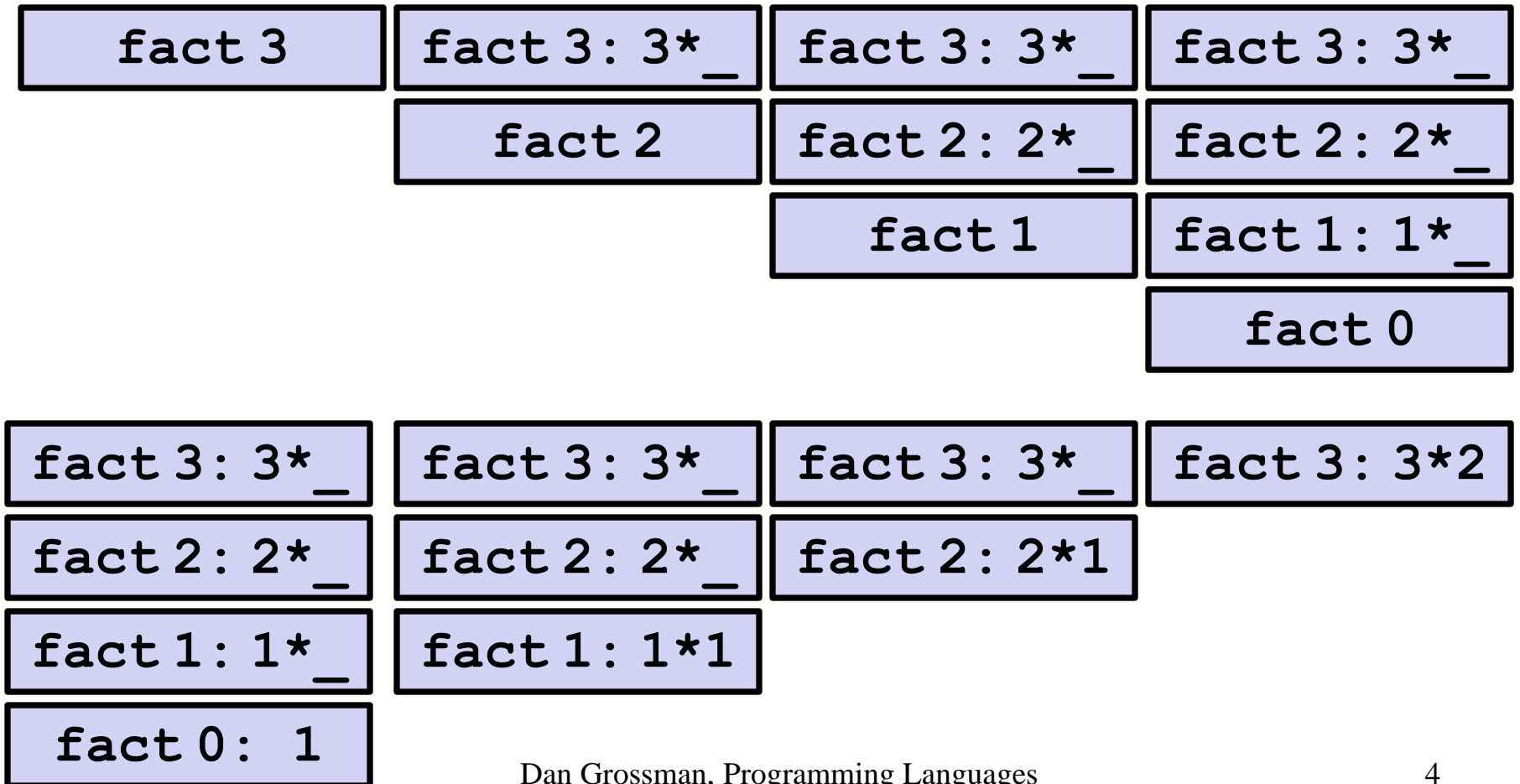
- Calling a function f pushes an instance of f on the stack
- When a call to f finishes, it is popped from the stack

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
```

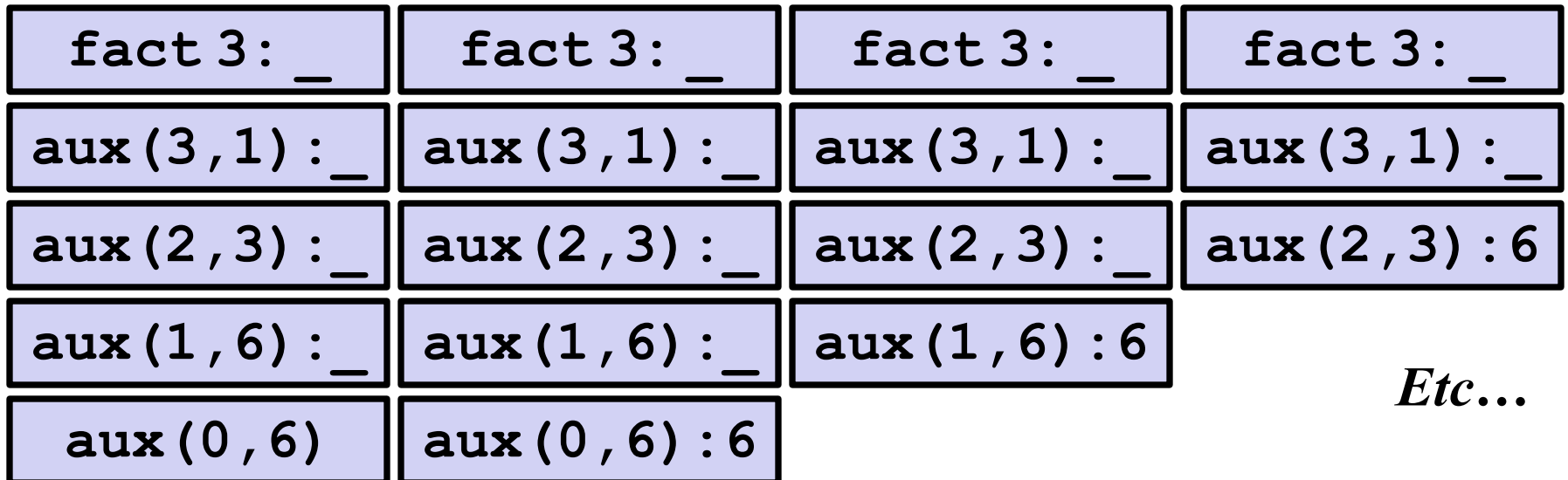
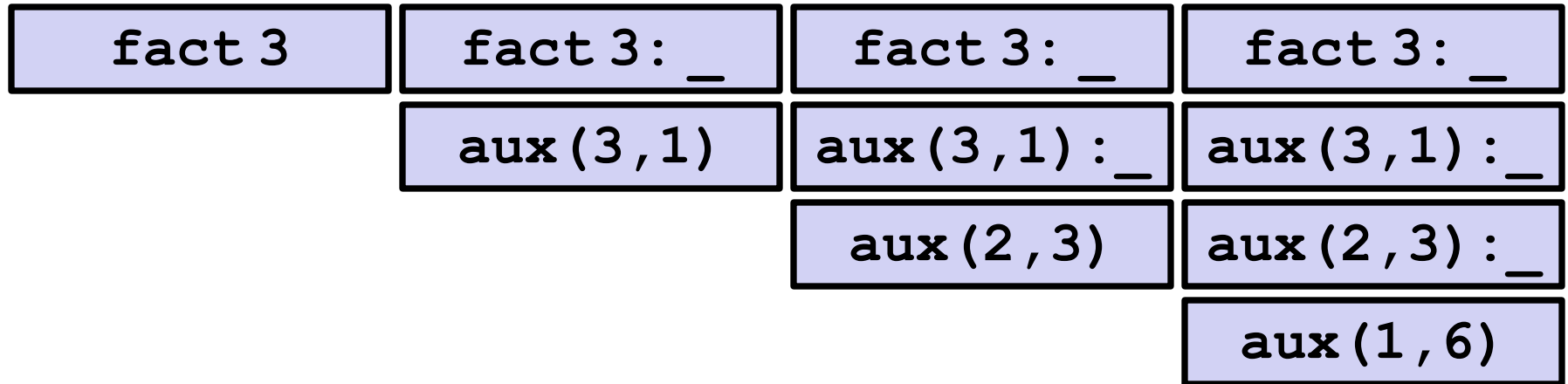


Example Revised

```
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



Etc...

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
```

fact 3

aux(3,1)

aux(2,3)

aux(1,6)

aux(0,6)

```
fun append (xs,ys) =  
  if xs=[]  
  then ys  
  else (hd xs)::append(tl xs,ys)  
  
fun map (f,xs) =  
  case xs of  
    [] => []  
  | x::xs' => (f x)::(map(f,xs'))  
  
val a = map (increment, [4,8,12,16])  
val b = map (hd, [[8,6],[7,5],[3,0,9]])
```

Programming Languages

Dan Grossman

Accumulators

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 $1+2+\dots+(\text{length}-1)$ is almost $\text{length} \times \text{length} / 2$
 - Moral: beware list-append, especially within outer recursion
- Cons constant-time (and fast), so accumulator version much better

```
fun append (xs,ys) =  
  if xs=[]  
  then ys  
  else (hd xs)::append(tl xs,ys)  
  
fun map (f,xs) =  
  case xs of  
    [] => []  
  | x::xs' => (f x)::(map(f,xs'))  
  
val a = map (increment, [4,8,12,16])  
val b = map (hd, [[8,6],[7,5],[3,0,9]])
```

Programming Languages

Dan Grossman

Tail Recursion: Perspective and Definition

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 $\mathbf{f\ x}$ is the “immediate result” for the enclosing function body, then $\mathbf{f\ x}$ 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
- ...