<section-header> CSE341: Programming Languages Lecture 5 More Datatypes and Pattern-Matching Dan Grossman Autumn 2017</section-header>	<text><text><text><text><code-block><text><code-block></code-block></text></code-block></text></text></text></text>
<pre>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</pre>	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 } 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 Add (Constant (10+9), Negate (Constant 4)) How to picture the resulting value in your head:	<pre>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)</pre>

Autumn 2017

5

Constant

19

Autumn 2017

1

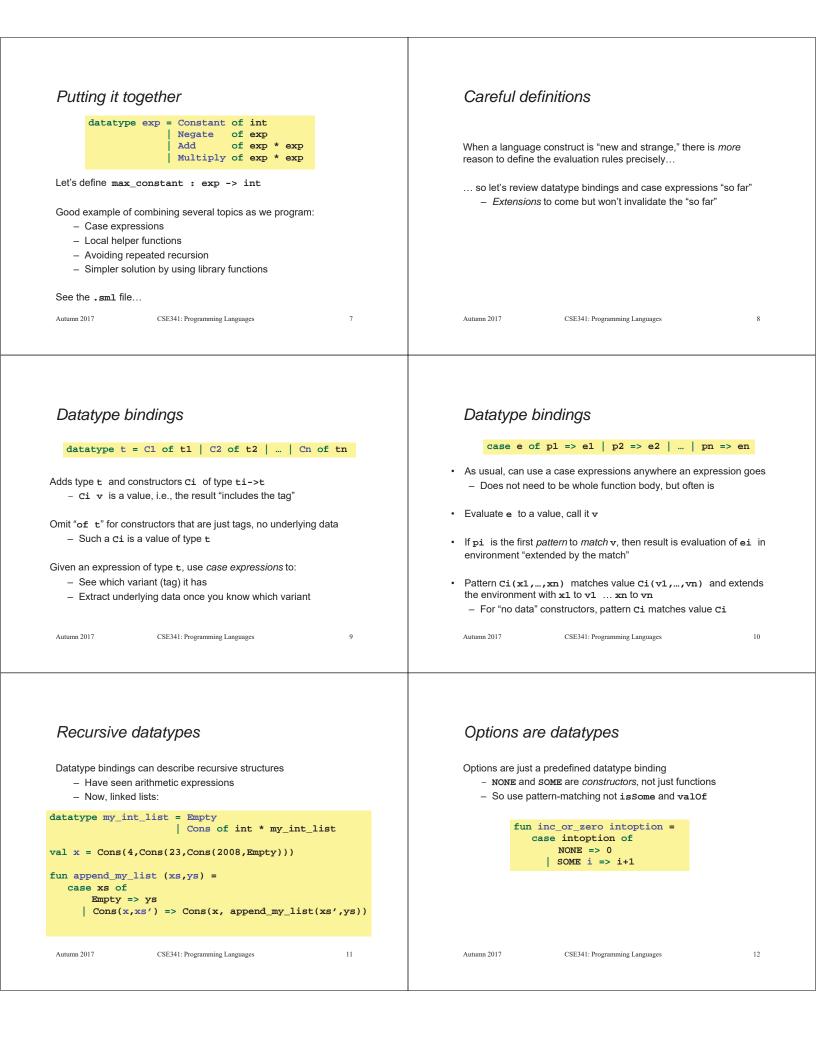
Negate

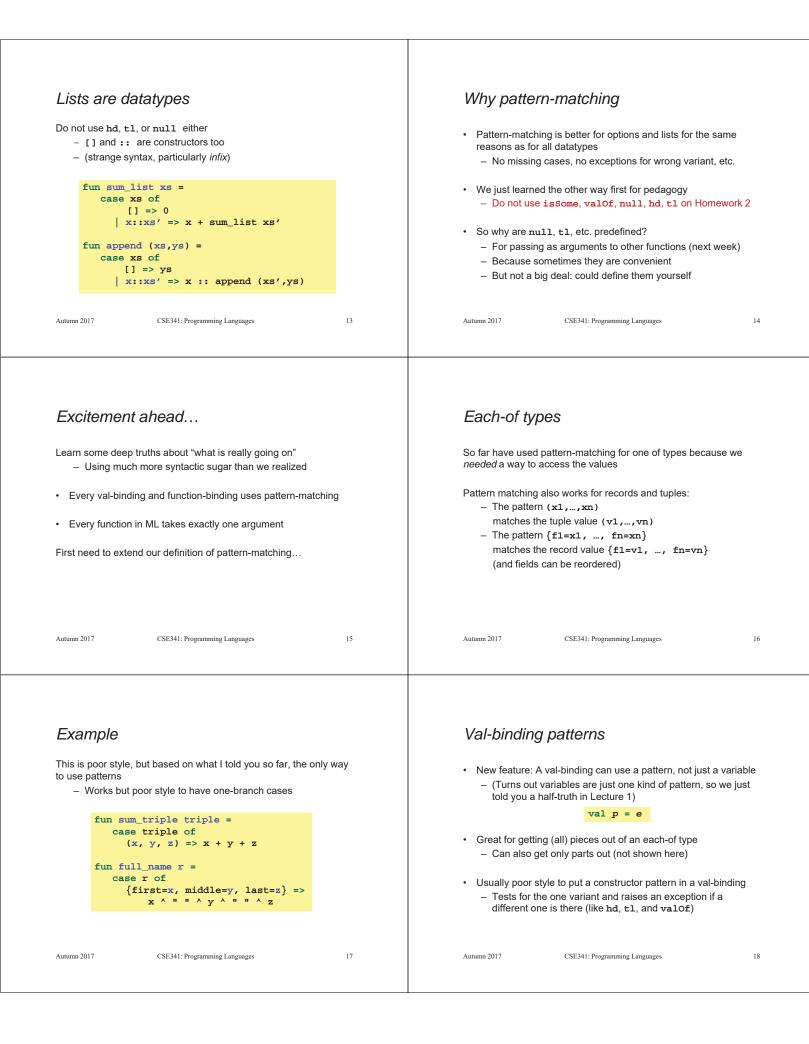
l Constant

I 4 CSE341: Programming Languages

CSE341: Programming Languages

6





Function-argument patterns Better example This is okay style A function argument can also be a pattern - Though we will improve it again next - Match against the argument in a function call - Semantically identical to one-branch case expressions fun f p = efun sum_triple triple = let val (x, y, z) = triple Examples (great style!): in x + y + zfun sum_triple (x, y, z) = end x + y + zfun full_name r = fun full_name {first=x, middle=y, last=z} = let val {first=x, middle=y, last=z} = r x ^ = = ^ y ^ = = ^ z in ^ = = ^ y ^ = = ^ z х end CSE341: Programming Languages Autumn 2017 19 Autumn 2017 CSE341: Programming Languages 20 Hmm A new way to go · For Homework 2: A function that takes one triple of type int*int*int and returns an int that is their sum: - Do not use the # character - Do not need to write down any explicit types fun sum_triple (x, y, z) = x + y + zA function that takes three int arguments and returns an int that is their sum fun sum_triple (x, y, z) = x + y + zSee the difference? (Me neither.) © Autumn 2017 CSE341: Programming Languages 21 Autumn 2017 CSE341: Programming Languages 22

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 ()

```
Autumn 2017 CSE341: Programming Languages 23
```