

## 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) Autumn 2018 CSE341: Programming Languages						

## 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

7

 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

Autumn 2018

CSE341: Programming Languages

#### 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

Autumn 2018	CSE341: Programming Languages	11

#### Example

{name	e = "Matai	", id = 4 -	3}	
Evaluates to				
{i	d = 1, nam	ne = "Matai	"}	
And has type				
	{id : int,	, name : st	ring}	
If some expression fields with:	on such as a v #id x		this type, the	n get
	program coul	e any record ty d also make a se} of type {i		ro:bool}
Autumn 2018	CSE341: Pr	ogramming Languages		8

## The truth about tuples

 $\ensuremath{\mathsf{Previous}}$  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

Autumn 2018 CSE341: Programming Languages

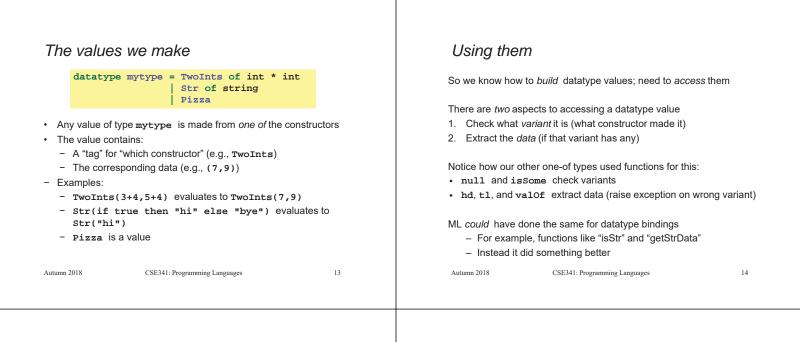
10

# Datatype bindings

A "strange" (?) and totally awesome (!) way to make one-of types: - A datatype binding

datatype	mytype	=	TwoInts	of	int	*	int
			Str of a Pizza	stri	ing		
		1	11110				

- 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



#### 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

Autumn 2018

CSE341: Programming Languages

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  $C \times or C(x,y)$  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

Autumn 2018

CSE341: Programming Languages

16

## Why this way is better

0. You can use pattern-matching to write your own testing and data-extractions functions if you must  $% \left( {{{\rm{A}}_{\rm{B}}}} \right)$ 

- 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

Autumn 2018

CSE341: Programming Languages

17

15