Unions and Products

Concepts of Programming Languages
Lecture 3

CAS CS 320

Practice Problem

Implement a function ***first_digit*** *which takes an integer* ***n*** *as an input and returns the first digit of* ***n*** *(<u>without</u> converting to a string)*

Outline

- Discuss Formal Typing/Semantic Rules
- Demonstrate how to organize data in OCaml in terms of products and unions types

Learning Objectives

- Read inference rules, i.e., translate mathematical notation to English and English to mathematical notation
- Work with basic structured data in OCaml

Recap

Recall: Local Variables (Informal)

body

```
let x = 2 in x + x
```

Recall: Local Variables (Informal)

body

```
let x = 2 in x + x
```

syntax: `let VARIABLE = EXPRESSION in BODY`

Recall: Local Variables (Informal)

body

```
let x = 2 in x + x
```

syntax: `let VARIABLE = EXPRESSION in BODY`

typing: the type is the same as that of BODY *given BODY is well-typed after substituting the VARIABLE in BODY*

Recall: Local Variables (Informal)

body

```
let x = 2 in x + x
```

syntax: `let VARIABLE = EXPRESSION in BODY`

typing: the type is the same as that of BODY *given BODY is well-typed after substituting the VARIABLE in BODY*

semantics: the is the same as the value of BODY *after substituting the VARIABLE in BODY*

Recall: A Note on Substitution

```
let x = 2 in x + x
```

$\longrightarrow$

```
2 + 2
```

Recall: A Note on Substitution

`let x = 2 in x + x` ⟶ `2 + 2`

Formally, we write $[v/x]e$ to mean "substitute v for x in e", e.g., $[3/x](x+x)$ is the same as $3+3$

Recall: A Note on Substitution

$\boxed{\texttt{let x = 2 in x + x}} \longrightarrow \boxed{\texttt{2 + 2}}$

Formally, we write $[v/x]e$ to mean "substitute v for x in e", e.g., $[3/x](x+x)$ is the same as $3+3$

Intuitively, substitution is simple: **replace the variable**

Recall: A Note on Substitution

$\boxed{\texttt{let x = 2 in x + x}} \longrightarrow \boxed{\texttt{2 + 2}}$

Formally, we write $[v/x]e$ to mean "substitute v for x in e", e.g., $[3/x](x + x)$ is the same as $3 + 3$

Intuitively, substitution is simple: **replace the variable**

Turns out, this is **very hard** to do correctly, *it's subtle* and a source of a lot of mistakes in PL implementations

Recall: If-Expressions (Informal)

```
let abs x = if x > 0 then x else -x
```

Recall: If-Expressions (Informal)

```
let abs x = if x > 0 then x else -x
```

Syntax: `if CONDITION then TRUE-CASE else FALSE-CASE`

Recall: If-Expressions (Informal)

```
let abs x = if x > 0 then x else -x
```

Syntax: if CONDITION then TRUE-CASE else FALSE-CASE

Typing: CONDITION must be a Boolean and TRUE-CASE and FALSE-CASE must be the same type. The type is then the same as that of TRUE-CASE and FALSE-CASE

Recall: If-Expressions (Informal)

```
let abs x = if x > 0 then x else -x
```

Syntax: if CONDITION then TRUE-CASE else FALSE-CASE

Typing: CONDITION must be a Boolean and TRUE-CASE and FALSE-CASE must be the same type. The type is then the same as that of TRUE-CASE and FALSE-CASE

Semantics: If CONDITION holds, then we get the TRUE-CASE, otherwise we get the FALSE-CASE

Recall: Functions (Informal)

body

```
fun x -> x + 1
```

Recall: Functions (Informal)

body

```
fun x -> x + 1
```

Syntax: `fun VAR-NAME -> EXPR`

Recall: Functions (Informal)

body

```
fun x -> x + 1
```

Syntax: `fun VAR-NAME -> EXPR`

Typing: the type of a function is **T1 -> T2** where T1 is the type of the input and T2 is the type of the output

Recall: Functions (Informal)

body

```
fun x -> x + 1
```

Syntax: `fun VAR-NAME -> EXPR`

Typing: the type of a function is **T1 -> T2** where T1 is the type of the input and T2 is the type of the output

Semantics: A function will evaluate to a special ***function value*** (printed as `<fun>` by UTop)

Recall: Curried Functions

```
let f = fun x -> fun y -> fun z -> x + y + z
```

We should think of the above function as something which takes an input and returns **another function**

In other words, we *partially apply* the function

Recall: Application (Informally)

```
(fun x -> fun y -> x + y + 1) 3 2
```

Recall: Application (Informally)

```
(fun x -> fun y -> x + y + 1) 3 2
```

Syntax: `FUNCTION-EXPR ARG-EXPR`

Recall: Application (Informally)

```
(fun x -> fun y -> x + y + 1) 3 2
```

Syntax: `FUNCTION-EXPR ARG-EXPR`

Typing: If FUNCTION-EXPR is of type T1 -> T2, and ARG-EXPR is of type T1, then the type is T2

Recall: Application (Informally)

```
(fun x -> fun y -> x + y + 1) 3 2
```

Syntax: FUNCTION-EXPR ARG-EXPR

Typing: If FUNCTION-EXPR is of type T1 -> T2, and ARG-EXPR is of type T1, then the type is T2

Semantics: Substitute the value of ARG-EXPR into the body of FUNCTION-EXPR and evaluate that

Inference Rules

Note: Production Rules and Syntax

```
<expr> ::= <expr> + <expr>
```

Note: Production Rules and Syntax

```
<expr> ::= <expr> + <expr>
```

Last week, we saw the above notation. This is called a ***production rule*** and is part of a ***BNF grammar***

Note: Production Rules and Syntax

```
<expr> ::= <expr> + <expr>
```

Last week, we saw the above notation. This is called a ***production rule*** and is part of a ***BNF grammar***

Reminder, this reads as: if e_1 is a well-formed expression and e_2 is a well-formed expression, then $e_1 + e_2$ is a well-formed expression

Note: Production Rules and Syntax

```
<expr> ::= <expr> + <expr>
```

Last week, we saw the above notation. This is called a ***production rule*** and is part of a ***BNF grammar***

Reminder, this reads as: if e_1 is a well-formed expression and e_2 is a well-formed expression, then $e_1 + e_2$ is a well-formed expression

We won't focus on this until the second half of the course but you should start to get comfortable with the syntax

Inference Rules

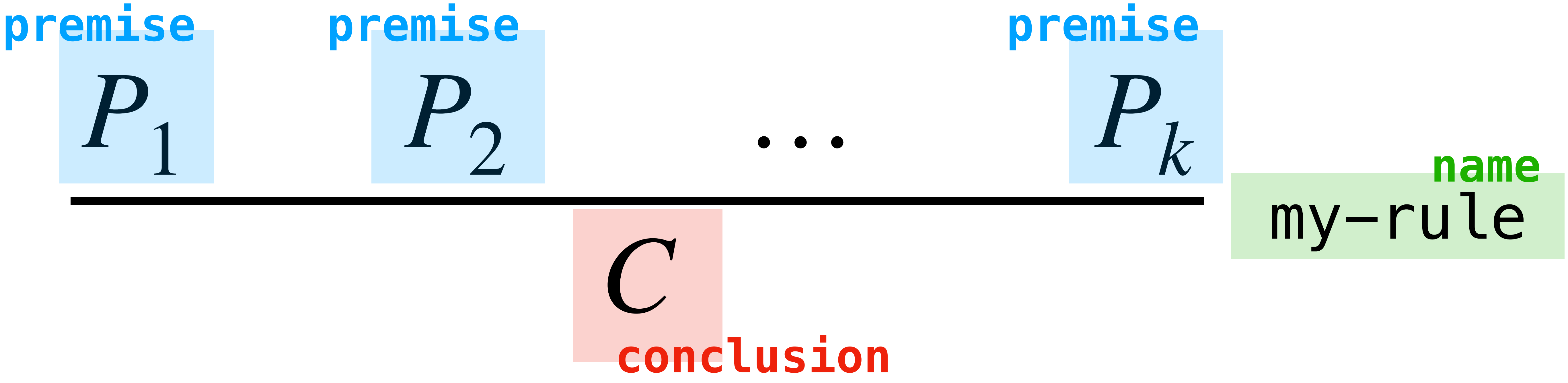
$$\frac{\overset{\texttt{premise}}{P_1} \qquad \overset{\texttt{premise}}{P_2} \qquad \cdots \qquad \overset{\texttt{premise}}{P_k}}{\underset{\texttt{conclusion}}{C}}\ \overset{\texttt{name}}{\texttt{my-rule}}$$

Inference Rules

$$\frac{\overset{\texttt{premise}}{P_1} \qquad \overset{\texttt{premise}}{P_2} \qquad \cdots \qquad \overset{\texttt{premise}}{P_k}}{\underset{\texttt{conclusion}}{C}}\ \overset{\texttt{name}}{\texttt{my-rule}}$$

Then general form of an inference rule has a collection of **premises** and a **conclusion**

Inference Rules



Then general form of an inference rule has a collection of **premises** and a **conclusion**

There may be no premises, this is called an **axiom**

Inference Rules

$$\dfrac{\overset{\text{premise}}{P_1} \qquad \overset{\text{premise}}{P_2} \qquad \cdots \qquad \overset{\text{premise}}{P_k}}{\underset{\text{conclusion}}{C}} \; \overset{\text{name}}{\texttt{my-rule}}$$

We can read this as:

*If P_1 through P_k hold, then C holds (by **my-rule**)*

Typing Judgments

$$\overset{\text{context}}{\boxed{\Gamma}} \vdash \overset{\text{expression}}{\boxed{e}} : \overset{\text{type}}{\boxed{\tau}}$$

A <u>typing judgment</u> a compact way of representing the statement:

e **is of type** τ **in the context** Γ

A **typing rule** is an inference rule whose premises and conclusion are typing judgments

Recall: Integer Addition Typing Rule

$$\frac{\Gamma \vdash e_1 : \texttt{int} \qquad \Gamma \vdash e_2 : \texttt{int}}{\Gamma \vdash e_1 \texttt{ + } e_2 : \texttt{int}} \text{ (addInt)}$$

If e_1 *is an* `int` *(in any context* Γ*) and* e_2 *is an* `int` *then (in any context* Γ*)* e_1 `+` e_2 *is an* `int` *(in any context* Γ*)*

Contexts

$$\Gamma = \{\ \texttt{x : int, y : string, z : int -> string}\ \}$$

Contexts

$$\Gamma = \{\ \texttt{x : int, y : string, z : int -> string}\ \}$$

A **context** is a set of **variable declarations**

Contexts

$\Gamma = \{$ `x : int, y : string, z : int -> string` $\}$

A **context** is a set of **variable declarations**

A variable declaration $(x : \tau)$ says: "I declare that the variable x is of type τ"

Contexts

$$\Gamma = \{\ \texttt{x : int, y : string, z : int -> string}\ \}$$

A **context** is a set of **variable declarations**

A variable declaration $(x : \tau)$ says: "I declare that the variable x is of type τ"

A context keeps track of all the types of variables in the "environment"

Example: Reading Typing Judgements

$$\{\texttt{b} : \texttt{bool}\} \vdash \texttt{if b then 2 else 3} : \texttt{int}$$

Example: Reading Typing Judgements

$$\{\texttt{b} : \texttt{bool}\} \vdash \texttt{if b then 2 else 3} : \texttt{int}$$

In English: *Given I declare that* *b* *is a* *bool*, *the expression* *if b then 2 else 3* *is an* *int*

Example: Reading Typing Judgements

$$\{\texttt{b} : \texttt{bool}\} \vdash \texttt{if b then 2 else 3} : \texttt{int}$$

In English: *Given I declare that b is a bool, the expression if b then 2 else 3 is an int*

The context allows us to determine the type of an expression *relative to the types of variables*

Semantic Judgements

$$\overset{\text{expression}}{e} \Downarrow \overset{\text{value}}{v}$$

A <u>semantic judgment</u> is a compact way of representing the statement:

The expression e **evaluates to the value** v

A **semantic rule** is an inference rule with semantic judgments

Recall: Integer Addition Semantic Rule

$$\frac{e_1 \Downarrow v_1 \qquad e_2 \Downarrow v_2}{e_1 + e_2 \Downarrow v_1 + v_2} \text{ (evalInt)}$$

If e_1 evaluates to the (integer) v_1 and e_2 evaluates to the (integer) v_2, then $e_1 + e_2$ evaluates to the (integer) $v_1 + v_2$

Example: Reading Semantic Judgments

$$\texttt{if 2 > 3 then 2 + 2 else 3} \Downarrow 3$$

In English: The expression `if 2 > 3 then 2 + 2 else 3` evaluates to the value 3

Note: Judgements are Statements

$$\{\texttt{b} : \texttt{bool}\} \vdash \texttt{if b then 2 else 3} : \texttt{string}$$

Note: Judgements are Statements

$$\{\mathtt{b} : \mathtt{bool}\} \vdash \mathtt{if\ b\ then\ 2\ else\ 3} : \mathtt{string}$$

A judgement is a *statement* in the same way that "there are infinitely many twin primes" or "pigs fly" is a statement

Note: Judgements are Statements

$$\{\texttt{b} : \texttt{bool}\} \vdash \texttt{if b then 2 else 3} : \texttt{string}$$

A judgement is a *statement* in the same way that "there are infinitely many twin primes" or "pigs fly" is a statement

We haven't **proved** anything by writing down a typing judgment

Note: Judgements are Statements

$$\{\texttt{b} : \texttt{bool}\} \vdash \texttt{if b then 2 else 3} : \texttt{string}$$

A judgement is a *statement* in the same way that "there are infinitely many twin primes" or "pigs fly" is a statement

We haven't **proved** anything by writing down a typing judgment

On Thursday: We will talk about **typing derivations**, which are used to demonstrate that expressions *actually* have their desired types in our PL

Note: Values are not Expressions

$$\texttt{if 2 > 3 then 2 + 2 else 3} \Downarrow 3$$

In this course, we will draw a distinction between values and expressions (note the font)

Example. We'll use regular numbers to represented integer values, and we'll use $\top$ and $\bot$ for the true and false Boolean values

Questions?

Expressions, Formally

Up Next

`We'll formalize what we've seen so far:`

- `Let-expressions`
- `If-Expressions`
- `Functions`
- `Application`

For now, just think of these as formal descriptions of how our PL behaves

Let-Expressions (Syntax Rule)

```
<expr> ::= let <var> = <expr> in <expr>
```

If x is a valid variable name, and e_1 is a well-formed expression and e_2 is a well-formed expression then

$$\texttt{let}\ x = e_1\ \texttt{in}\ e_2$$

is a well-formed expression

Let-Expressions (Typing Rule)

$$\frac{\Gamma \vdash e_1 : \tau_1 \qquad \Gamma, x : \tau_1 \vdash e_2 : \tau}{\Gamma \vdash \texttt{let}\ x\ \texttt{=}\ e_1\ \texttt{in}\ e_2 : \tau}\ (\text{let})$$

If e_1 is of type τ_1 in the context Γ, and e_2 is of type τ in the context Γ *with the variable declaration* $(x : \tau_1)$ *added to it*, then

$$\texttt{let}\ x\ \texttt{=}\ e_1\ \texttt{in}\ e_2$$

is of type τ in the context Γ

Let-Expressions (Semantic Rule)

$$\frac{e_1 \Downarrow v_1 \qquad [v_1/x]e_2 \Downarrow v}{\texttt{let } x \texttt{ = } e_1 \texttt{ in } e_2 \Downarrow v} \text{(letEval)}$$

If e_1 evaluates to v_1 and e_2 *with v_2 substituted for x* evaluates to v, then

$$\texttt{let } x \texttt{ = } e_1 \texttt{ in } e_2$$

evaluates to v

If-Expressions (Syntax Rule)

```
<expr> ::= if <expr> then <expr> else <expr>
```

If e_1 is a well-formed expression and e_2 is a well-formed expression and e_3 is a well-formed expression, then

$$\texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3$$

is a well-formed expression

If-Expressions (Typing Rule)

$$\frac{\Gamma \vdash e_1 : \texttt{bool} \qquad \Gamma \vdash e_2 : \tau \qquad \Gamma \vdash e_3 : \tau}{\Gamma \vdash \texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3 : \tau}\text{(if)}$$

If e_1 is of type bool in the context Γ and e_2 and e_3 are of type τ in the context Γ, then

$$\texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3$$

is of type τ in the context Γ

If-Expressions (Semantic Rule 1)

$$\frac{e_1 \Downarrow \top \qquad e_2 \Downarrow v_2}{\texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3 \Downarrow v_2}\ (\text{ifEvalTrue})$$

If e_1 evaluates to $\top$ and e_2 evaluates to v_2, then

$$\texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3$$

evaluates to v_2

If-Expressions (Semantic Rule 2)

$$\frac{e_1 \Downarrow \bot \qquad e_3 \Downarrow v_3}{\texttt{if}\ \ e_1\ \ \texttt{then}\ \ e_2\ \ \texttt{else}\ \ e_3 \Downarrow v_3}\ \text{(ifEvalFalse)}$$

If e_1 evaluates to $\bot$ and e_2 evaluates to v_2, then

$$\texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3$$

evaluates to v_3

Functions (Syntax Rule)

```
<expr> ::= fun <var> -> <expr>
```

If x is a valid variable name and e is a well-formed expression, then

`fun` x `->` e

is a well-formed expression

Functions (Typing Rule)

$$\frac{\Gamma, x : \tau_1 \vdash e : \tau_2}{\Gamma \vdash \texttt{fun}\ x\ \texttt{->}\ e : \tau_1\ \texttt{->}\ \tau_2}\ (\text{fun})$$

If e has type τ_2 in the context Γ with the variable declaration $(x : \tau_1)$ added, then

$$\texttt{fun}\ x\ \texttt{->}\ e$$

is of type τ_1 `->` τ_2 in the context Γ

Functions (Semantic Rule)

$$\frac{}{\texttt{fun}\ x\ \texttt{->}\ e \Downarrow \lambda x . e}\ \text{(funEval)}$$

Under no premises, the expression

$$\texttt{fun}\ x\ \texttt{->}\ e$$

evaluates to the function value $\lambda x . e$

Application (Syntax Rule)

`<expr> ::= <expr> <expr>`

If e_1 is a well-formed expression and e_2 is a well-formed expression, then $e_1\ e_2$ is a well-formed expression

Application (Typing Rule)

$$\frac{\Gamma \vdash e_1 : \tau_2 \texttt{ -> } \tau \qquad \Gamma \vdash e_2 : \tau_2}{\Gamma \vdash e_1\ e_2 : \tau} \text{(app)}$$

If e_1 has type τ_2 `->` τ under the context Γ and e_2 is of type τ_2 under the context Γ, then $e_1\ e_2$ is of type τ under the context Γ

Application (Semantic Rule)

$$\frac{e_1 \Downarrow \lambda\, x\, .\, e \qquad e_2 \Downarrow v_2 \qquad [v_2/x]e \Downarrow v}{e_1\ e_2 \Downarrow v}\text{(appEval)}$$

» e_1 evaluates to a function value $\lambda x . e$

» e_2 evaluates to v_2

» e with v_2 substituted for x evaluates to v

It follows that $e_1\ e_2$ evaluates to v

Example

```
(let x = 2 in fun y -> x + y) (2 + 3)
```

Understanding Check

Offline, go back to the recap slides at the beginning and compare the formal and informal descriptions...

We'll see more typing rules and semantic rules

We'll also give a written reference for the rules we talk about in class

Practice Problem

```
let k = fun x -> fun y -> x in
let x = 3 + k k 2 3 in
k x (k x)
```

What does the above expression evaluate to?

Products

Tuples

```
let point : float * float = (2.0, 3.0)
let student : string * int = ("Franco", 244342)
```

Tuples

```
let point : float * float = (2.0, 3.0)
let student : string * int = ("Franco", 244342)
```

Tuples are ordered unlabeled fixed-length heterogeneous collections of data

Tuples

```
let point : float * float = (2.0, 3.0)
let student : string * int = ("Franco", 244342)
```

Tuples are ordered unlabeled fixed-length heterogeneous collections of data

(I expect that these are familiar)

Tuples

```
let point : float * float = (2.0, 3.0)
let student : string * int = ("Franco", 244342)
```

Tuples are ordered unlabeled fixed-length heterogeneous collections of data

(I expect that these are familiar)

These are useful for returning multiple arguments from a function

Pattern Matching on Tuples

```
let hypotenuse (p : float * float) : float =
  match p with
  | (x, y) -> sqrt (x *. x +. y *. y)
```

There are no accessors for tuples

Instead we can use **pattern matching**

Pattern Matching in General

```
match e with
| p -> o
| ...
```

Pattern Matching in General

```
match e with
| p -> o
| ...
```

A **pattern** is like a typed template for how a piece of data should look

Pattern Matching in General

```
match e with
| p -> o
| ...
```

A **pattern** is like a typed template for how a piece of data should look

A **match-expression** is a way of *destructing* <u>any</u> piece of data in OCaml

Pattern Matching in General

```
match e with
| p -> o
| ...
```

A **pattern** is like a typed template for how a piece of data should look

A **match-expression** is a way of *destructing* <u>any</u> piece of data in OCaml

We *match* on an expression e, and check if the value of e *matches* with the pattern p

Note: Patterns are not Expressions

```
<expr> ::= match <expr> with
           | <pattern> -> <expr>
           | <pattern> -> <expr>
           | ....
```

Patterns are similar to expressions, but with some key differences

They can be wildcards, they can be variables, there's a lot of options

We'll talk more about patterns on Thursday

Advanced Pattern Matching

```
let hypotenuse ((x, y) : float * float) : float =
  sqrt (x *. x +. y *. y)

let hypotenuse (p : float * float) : float =
  let (x, y) = p in
  sqrt (x *. x +. y *. y)
```

Pattern matching can also be done implicitly in let-expression and function arguments!

And we can do all this formally...

Tuples (Syntax Rule)

```
<expr> ::= ( <expr> , ... , <expr> )
```

If $e_1, \ldots, e_n$ are well-formed expressions, then

$$(\; e_1 \; , \; \ldots \; , \; e_n \;)$$

is a well-formed expression

Tuple (Typing Rule)

$$\frac{\Gamma \vdash e_1 : \tau_1 \qquad \Gamma \vdash e_2 : \tau_2 \qquad \ldots \qquad \Gamma \vdash e_n : \tau_n}{\Gamma \vdash \texttt{(}\ e_1\ \texttt{,}\ \ldots\ \texttt{,}\ e_n\ \texttt{)} : \tau_1\ \texttt{*}\ \ldots\ \texttt{*}\ \tau_n}\ \text{(tuple)}$$

If $e_1, \ldots, e_n$ are of type $\tau_1, \ldots, \tau_n$, respectively, in the context Γ then

$$\texttt{(}\ e_1\ \texttt{,}\ \texttt{...}\texttt{,}\ e_n\ \texttt{)}$$

is of type τ_1 * ... * τ_n in the context Γ

Tuple (Semantic Rule)

$$\frac{e_1 \Downarrow v_1 \qquad \dots \qquad e_n \Downarrow v_n}{\texttt{(}\ e_1\ \texttt{,}\ \dots\ \texttt{,}\ e_n\ \texttt{)} \Downarrow \texttt{(}\ v_1\ \texttt{,}\ \dots\ \texttt{,}\ v_n\ \texttt{)}}\ (\text{tupleEval})$$

If $e_1, \dots, e_n$ evaluate to $v_1, \dots, v_n$, respectively, then

$$\texttt{(}\ e_1\ \texttt{,}\ \texttt{...,}\ e_n\ \texttt{)}$$

evaluates to (v_1 , ..., v_n)

Records

```
type point = { x_cord : float ; y_cord : float }
let origin : point = { x_cord = 0. ; y_cord = 0. }

type user = {
  name : string ;
  email : string ;
  num_posts : int ;
}
```

Records are unordered labeled fixed-length heterogeneous collections of data

They are useful for organizing large collections of data (akin to database records)

Record Syntax

```
type record_ty =
  {
    field1 : ty1;
    field2 : ty2;
    ...
    fieldn : tyn'
  }
```

```
let record_expr : record_ty =
  {
    field1 = expr1;
    field2 = expr2;
    ...
    fieldn : exprn;
  }
```

For a record, we have to specify the type of each field

When we construct a record, every field must have a value

Accessors

```
type point = { x_cord : float ; y_cord : float }

let dist (p : point) (q : point) =
  let xd = p.x_cord -. q.x_cord in
  let yd = p.y_cord -. q.y_cord in
  sqrt (xd *. xd +. yd *. yd)
```

Records support **dot-notation**

(we can also access records by pattern matching)

Record Updates

```
let new_post u : user =
  { u with num_posts = u.num_posts + 1 }
```

Record Updates

```
let new_post u : user =
  { u with num_posts = u.num_posts + 1 }
```

We can use **with-syntax** to update a smaller number of fields in a large record

Record Updates

```
let new_post u : user =
  { u with num_posts = u.num_posts + 1 }
```

We can use **with-syntax** to update a smaller number of fields in a large record

"u with number of posts incremented, keep everything else the same"

Record Updates

```
let new_post u : user =
  { u with num_posts = u.num_posts + 1 }
```

We can use **with-syntax** to update a smaller number of fields in a large record

"u with number of posts incremented, keep everything else the same"

Data in functional languages are immutable. This returns a new record with the update

Unions

Simple Variants

```
type os = BSD | Linux | MacOS | Windows
```

A **simple variant** is a *user-defined* type for values of a fixed collection of possibilities

Type names are **lower_case** and Constructors names are **Upper_case**

Simple Variants

```
      constructor
type os = BSD | Linux | MacOS | Windows
```

A **simple variant** is a *user-defined* type for values of a fixed collection of possibilities

Type names are **lower_case** and Constructors names are **Upper_case**

Pattern Matching

```
let supported (sys : os) : bool =
  match sys with
  | BSD -> false
  | _ -> true
```

We work with variants by **pattern matching:**

» giving a <u>pattern</u> that a value can <u>match</u> with

» writing what to do for each pattern

Pattern Matching

```
                    let supported (sys : os) : bool =
                      match sys with
constant pattern      | BSD -> false
wildcard pattern      | _ -> true
```

We work with variants by **pattern matching:**

» giving a <u>pattern</u> that a value can <u>match</u> with

» writing what to do for each pattern

Data-Carrying Variants

```
type linux_distro = Arch | Fedora | NixOS | Ubuntu

type os
  = BSD of int * int
  | Linux of linux_distro * int
  | MacOS of int
  | Windows of int

let supported (sys : os) : bool =
  match sys with
  | BSD (major , minor) -> major > 2 && minor > 3
  | _ -> true
```

Variants can carry data, which allows us to represent more complex structures

Data-Carrying Variants

```
type linux_distro = Arch | Fedora | NixOS | Ubuntu

type os
  = BSD of int * int
  | Linux of linux_distro * int
  | MacOS of int
  | Windows of int

let supported (sys : os) : bool =
  match sys with
  | BSD (major , minor) -> major > 2 && minor > 3
  | _ -> true
```

Note the syntax

Variants can carry data, which allows us to represent more complex structures

Pro Tip: Named Data-Carrying Variants

```
type os
  = MacOS of {
        major : int ;
        minor : int ;
        patch : int
  }
  | ...

let support (sys : os) : bool =
  match sys with
  | MacOS info -> info.minor >= 14 && info.patch >= 1
    (* MacOS Sonoma 10.14.(1-3) *)
  |...
```

Since we can carry *any* kind of data in a constructor, we can carry records to name the parts of our carried data.

Understanding Check

```
let area (s : shape) =
  match s with
  | Rect r -> r.base *. r.height
  | Triangle { sides = (a, b) ; angle } -> Float.sin angle *. a *. b
  | Circle r -> r *. r *. Float.pi
```

*Define the variant **shape** which makes this function type-check.*

Summary

Inference rules formally describe how the typing and semantics of a programming language work

Tuples and **records** allow us to group data

Variants allow us to organize data by *possible outcomes*