

# CMSC 330: Organization of Programming Languages

---

## Rust Basics

# Organization

---

- It turns out that a lot of Rust has direct analogues in OCaml
  - So we will introduce its elements with comparisons

```
let rec fact n =  
  if n = 0 then 1  
  else  
    let x = fact (n-1) in  
    n * x
```

# Factorial in Rust (recursively)

rec by default

```
fn fact(n:i32) -> i32
{
  if n == 0 { 1 }
  else {
    let x = fact(n-1);
    n * x
  }
}
```

parm, return  
types explicit

physical eq  
(no built-in  
structural eq)

block  
(expression)

local var  
type inferred

Rust

```
let rec fact n =
  if n = 0 then 1
  else
    let x = fact (n-1) in
    n * x
```

OCaml

# Running factorial

---

- Rust programs start at **main**

- As in C and Java

on call: arg

parens required

```
fn main() {  
    let res = fact(6);  
    println!("fact(6) = {}", res);  
}
```

- Prints **fact(6) = 720**

format string

string conversion

- Aside: command-line args via **env::args**

# Block Expressions *block*

---

- Syntax

- { *stmt*\* *e*? }

- i.e., zero or more statements (separated by semi-colons) followed by an optional final expression

- Evaluation

- Evaluate each *stmt*; result is evaluation of *e*

- Or () if *e* is absent

- Type checking

- Must type check each *stmt*, extending environment of subsequent *stmts* with added `let`-bindings

- Final type is the type of *e*

- Or `unit` if *e* is absent

# If *Expressions* (not Statements)

---

- Syntax
  - **if** *e* *block1* *block2*
    - *e* is the guard
    - *block1* and *block2* are the true/false branches
- Evaluation
  - Evaluate *e* to *v*
  - Result is evaluation of *block1* if *v* is **true**; or *block2* if *v* is **false**
- Type checking
  - *e* : **bool**
  - *block1* : *t* and *block2* : *t* for some *t*

# Functions

---

- Syntax

- **fn** *f*(*parms*) [ $\rightarrow$  *t*] *block*
- *f* is the *function name*
- *parms* are *formal parameters*, including their types
  - Zero or more; have form *x1:t1*, ..., *xn:tn*
- *t* is the *return type*
  - May be omitted if function returns unit value ()
- *block* is the body, which is a block expression

# Let Statements

---

- Syntax

- `let [mut]? x[:t]? = e;`

- Keyword is `mut` optional
    - Type `t` is optional; often can be inferred if missing

- Evaluation

- Evaluate `e` to `v`; set `x` to `v` within the defining scope
  - `x` is *immutable* unless `mut` keyword is present

- Type checking

- If type `t` given, then `e : t` required
    - Else `e` should have *some* type `t`, which is inferred
  - `x:t` assumed in rest of scope; immutability enforced



# Let Statement Usage Examples

---

```
{  
  let x = 37;  
  let y = x + 5;  
  y  
} //42
```

```
{  
  let x = 37;  
  x = x + 5; //err  
  x  
}
```

```
{ //err:  
  let x:u32 = -1;  
  let y = x + 5;  
  y  
}
```

```
{  
  let x = 37;  
  let x = x + 5;  
  x  
} //42
```

```
{  
  let mut x = 37;  
  x = x + 5;  
  x  
} //42
```

```
{  
  let x:i16 = -1;  
  let y:i16 = x+5;  
  y  
} //4
```

Redefining a variable *shadows* it (like OCaml)

Assigning to a variable only allowed if **mut**

Type annotations must be consistent (may override defaults)

# Quiz 1: What does this evaluate to?

---

```
{ let x = 6;  
  let y = "hi";  
  if x == 5 { y } else { 5 };  
  7  
}
```

- A. 6
- B. 7
- C. 5
- D. Error

# Quiz 1: What does this evaluate to?

---

```
{ let x = 6;
  let y = "hi";
  if x == 5 { y } else { 5 };
  7
}
```

A. 6

B. 7

C. 5

**D. Error – if and else have incompatible types**

## Quiz 2: What does this evaluate to?

---

```
{ let x = 6;  
  let y = 4;  
  let x = 8;  
  x == 10-y  
}
```

- A. 6
- B. true
- C. false
- D. error

## Quiz 2: What does this evaluate to?

---

```
{ let x = 6;  
  let y = 4;  
  let x = 8;  
  x == 10-y  
}
```

- A. 6
- B. true
- C. false
- D. error

# Pattern: Conditional Initialization

---

- Initialization expressions in `let` statements are arbitrary expressions
  - Thus can be dynamically determined

```
fn foo(cond:bool) -> i32 {  
  let num = if cond { 5 } else { 6 };  
  num+1  
}
```

```
foo(true) == 6
```

```
foo(false) == 7
```

# Using Mutation

---

- Mutation is useful when performing iteration
  - As in C and Java

```
fn fact(n: u32) -> u32 {  
  let mut x = n;  
  let mut a = 1;  
  loop {  
    if x <= 1 { break; }  
    a = a * x;  
    x = x - 1;  
  }  
  a  
}
```

locals  
mutable

infinite loop  
(break out)

# Other Looping Constructs

---

- **While** loops
  - **while** *e block*
- **For** loops
  - **for** *pat in e block*
    - More later – e.g., for iterating through collections
- These (and **loop**) are *expressions*
  - They return the final computed value
    - unit, if none
  - **break** may take an expression argument, which is the final result of the loop



## Quiz 3: What does this evaluate to?

---

```
let mut x = 1;  
for i in 1..6 {  
    let x = x + 1;  
}  
x
```

- A. 1
- B. 6
- C. 0
- D. error

## Quiz 3: What does this evaluate to?

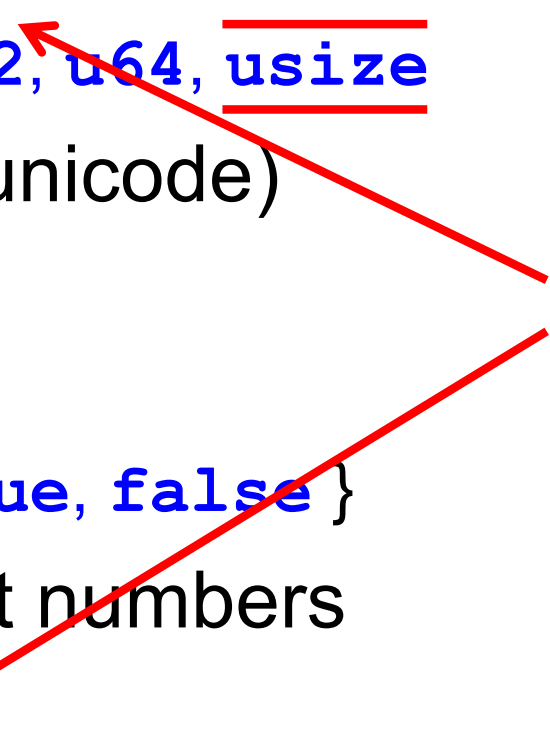
---

```
let mut x = 1;  
for i in 1..6 {  
    let x = x + 1;  
}  
x
```

- A. 1
- B. 6
- C. 0
- D. error

# Data: Scalar Types

---

- Integers
    - `i8`, `i16`, `i32`, `i64`, `isize`
    - `u8`, `u16`, `u32`, `u64`, `usize`
  - Characters (unicode)
    - `char`
  - Booleans
    - `bool` = { `true`, `false` }
  - Floating point numbers
    - `f32`, `f64`
  - Note: arithmetic operators (+, -, etc.) *overloaded*
- Machine word size
- Defaults (from inference)
- 

# Compound Data: Tuples and Arrays

---

- Tuples
  - n-tuple **type** (*t1*, ..., *tn*)
    - `unit ()` is just the 0-tuple
  - n-tuple **expression** (*e1*, ..., *en*)
  - Accessed by pattern matching or like a record field
- Arrays
  - constant length
    - Thus, not as useful as `Vec<t>` type, discussed later
  - array **type** [*t*]
    - And type [*t*; *n*] where *n* is the array's (constant) length
  - array **expression** has Ruby-like syntax [*e1*, ..., *en*]

# Compound Data: Tuples

---

```
fn dist(s: (f64, f64), e: (f64, f64)) -> f64 {  
  let (sx, sy) = s; pattern  
  let ex = e.0; accessor  
  let ey = e.1;  
  let dx = ex - sx;  
  let dy = ey - sy;  
  (dx*dx + dy*dy).sqrt() method invocation  
}
```

Rust

```
let dist s e =  
  let (sx, sy) = s in  
  let (ex, ey) = e in  
  let dx = ex -. sx in  
  let dy = ey -. sy in  
  sqrt (dx *. dx +. dy *. dy)
```

OCaml

# Compound Data: Tuples

---

Can include patterns in parameters directly, too

```
fn dist2( (sx, sy) : (f64, f64), (ex, ey) : (f64, f64) ) -> f64 {  
    let dx = ex - sx;  
    let dy = ey - sy;  
    (dx*dx + dy*dy).sqrt()  
}
```

Rust

```
let dist (sx, sy) (ex, ey) =  
    let dx = ex -. sx in  
    let dy = ey -. sy in  
    sqrt (dx *. dx +. dy *. dy)
```

OCaml

We'll see Rust structs later. They generalize tuples.

# Arrays

---

- Standard operations
  - **Creating** an array (can be mutable or not)
    - But must be of fixed length
  - **Indexing** an array
  - **Assigning** at an array index

```
let nums = [1,2,3];
let strs = ["Monday", "Tuesday", "Wednesday"];
let x = nums[0]; // 1
let s = strs[1]; // "Tuesday"
let mut xs = [1,2,3];
xs[0] = 1; // OK, since xs mutable
let i = 4;
let y = nums[i]; //fails (panics) at run-time
```

# Array Iteration

---

- Rust provides a way to **iterate over a collection**
  - Including arrays

```
let a = [10, 20, 30, 40, 50];
for element in a.iter() {
    println!("the value is: {}", element);
}
```

- `a.iter()` produces an **iterator**, like a Java iterator
  - This is a **method call**, *a la* Java. More about these later
- The special `for` syntax issues the `.next()` call until no elements are left
  - No possibility of running out of bounds



## Quiz 4: Will this function type check?

---

```
fn f(n: [u32]) -> u32 {  
    n[0]  
}
```

- A. Yes
- B. No

## Quiz 4: Will this function type check?

---

```
fn f(n: [u32]) -> u32 {  
    n[0]  
}
```

A. Yes

**B. No – because  
array length not  
known**

# Fun Fact

---

- The original Rust compiler was written in **OCaml**
  - Betrays the sentiments of the language's designers!
- Now the Rust compiler is written in ... **Rust**
  - How is this possible? Through a process called **bootstrapping**:
    - The first Rust compiler written in Rust is compiled by the Rust compiler written in OCaml
    - Now we can use the binary from the Rust compiler to compile itself
    - We discard the OCaml compiler and just keep updating the binary through self-compilation
    - So don't lose that binary! 😊