

CSE 231: Branches, Types and Tags

Ranjit Jhala April 7, 2026

Branches and Tags

Next, lets add

- **input** i.e. so we don't "know" result at compile-time, ✓
- **booleans** i.e. true and false,
- **equality** i.e. to *compare* values,
- **branches** i.e. if-then-else,
- **types** to distinguish *numbers* and *booleans*.

This will teach us about

- control flow **labels** and **conditional jumps** in assembly,
- **tagged data representation** using one machine word.

Concrete Syntax

```
<expr> := ...
  | input           -- user input
  | true            -- bool `true`
  | false           -- bool `false`
  | (= <expr> <expr>) -- equality test
  | (if <expr> <expr> <expr>) -- if-then-else
```

QUIZ: Abstract Syntax

First, lets fill in the cases for true, false, input, eq, and if in the abstract syntax.

```
enum Expr {
  // cases for Number, Add1, Sub1, Let, Var

}
```

QUIZ: Semantics of input

What should the semantics of input be?

Program

```
(add1 (add1 (add1 input)))
```

```
(+ input 10)
```

(+ input input)

Evaluator for input

How can we modify the eval function to handle input?

```
fn eval(e: &Expr, env: &Env) -> i64 {
  match expr {
    Expr::Input => _____,
  }
}
```

Runtime for input

How can we modify the runtime to handle input?

```
fn main() {
  let i: i64 = unsafe {
    our_code_starts_here(
      ^
    );
  }
  println!("{}", i);
}
```

Compiler for input

How can we modify the compiler to handle input?

```
fn compile_expr(e: &Expr, stack: &Stack) -> String {
  match expr {
    Expr::Input => mov rax, rdi,
  }
}
```

(let (input →) e)

true / false / 'if'

QUIZ: Semantics

Program $e_1 ? e_2 : e_3$ Result

(if true 22 33) \rightarrow 22

(if false 22 33) \rightarrow 33

(let (x (if false 22 99))
(if true (add1 x) 999)) \rightarrow 100

QUIZ: Evaluator

Lets fill in the cases for the eval function.

Challenge: How can we represent True and False as i64?

```
fn eval(expr: &Expr, env: &Env) -> i64 {
  match expr {
    // cases for Number, Add1, Sub1, Let, Var

    Expr::True => 1,
    Expr::False => 0,
    Expr::Eq(e1, e2) => {
      _____
      _____
    }
    Expr::If(cond, e1, e2) => {
      _____
      _____
      _____
    }
  }
}
```

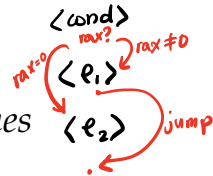
Naive Representation

Let's start old-school, C-style

- false is 0
- true is any non-zero value...

Why is this a bad idea?

(if true 22 33)
 (if cond e1 e2)



```
<cond>
  cmp rax, 0
  je ELSE
  <e1>
  jmp EXIT
ELSE:
  <e2>
EXIT:
```

Strategy: Branches

Compile if with labels, comparisons and jumps

Labels

```
our_code_label:
  ...
```

- Landmarks where execution can start or be diverted

Comparisons

```
cmp a1, a2
```

- Compare the values in a1 and a2
- Store the result in a special processor flag

Jump: Unconditional

Unconditional

```
jmp LABEL # jump to LABEL
```

- Just jump to LABEL, no questions asked!

Jump: Conditional

```
je LABEL # jump IF last comparison was EQUAL
jne LABEL # jump IF last comparison was NOT-EQUAL
```

- Jump depending on result of last comparison
- Else, carry on with next instruction...

QUIZ: Assembly

Suppose false is 0 and true is 1. Write the assembly code for

Program

```
(if true 22 33)
```

Assembly

```
mov rax, 1
```

`(if false 22 33)``mov rax, 0``(if cond e1 e2)``;; strategy for `if```<cond>``cmp rax, 0``je ELSE``<e1>``jmp EXIT``ELSE:``<e2>``EXIT:`

QUIZ: Compilation Code

Fill in the cases for `compile_expr` for true and false and if

How to get labels?

```
fn compile_expr(expr: &Expr, env: &Env) -> String {
  match expr {
    // other cases ...
```

`Expr::True => _____,`

`Expr::False => _____,`

`Expr::If(cond, e1, e2) => {`

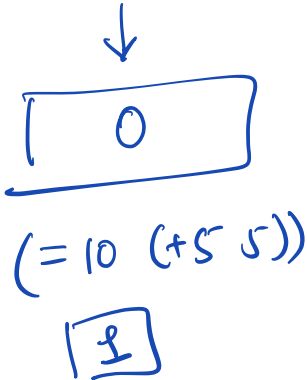
```
  }
}
```

QUIZ: Compiling Equality

Suppose false is 0 and true is 1. Write the assembly code for

Program

```
(= 10 20)
```



```
(= e1 e2)
```

Assembly

```
mov rax, 10
cmp rax, 20
jne lbl-ne
↓
lbl-eq:
  mov rax, 1
  jmp lbl-ent
lbl-ne:
  mov rax, 0
lbl-ent:
```

je cmovbe
jne cmovne

cmovbe dst, src

cmovne dst, src

```
;; strategy for `eq`
<e1>
mov [rbp-8+tmp], rax
<e2>
cmp [rbp-8+tmp], rax
```

jne lbl-ne
↓
lbl-eq:
 mov rax, 1
 jmp lbl-ent
lbl-ne:
 mov rax, 0

```
cmovbe rax, 1
cmovne rax, 0
```

EXERCISE Try to do it *without* jumps!

lbl-ent:

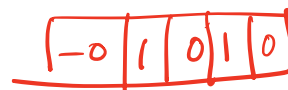
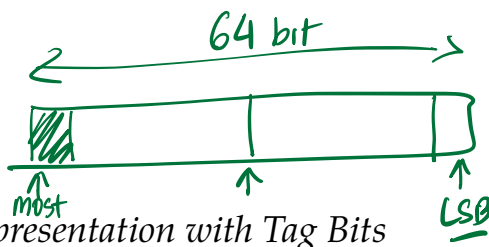
Data Representation

5 + true ?
true + false = true

Why is it a **bad** idea to represent true as 1 and false as 0?

true ⇒ 1 }
false ⇒ 0 }

How can we **track type** at runtime?

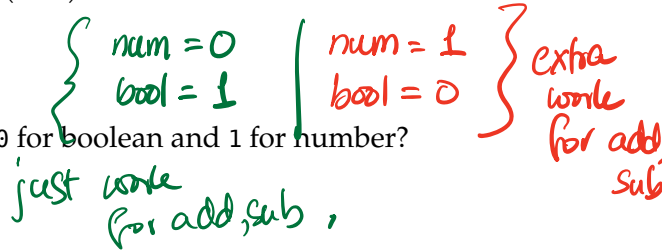


Data Representation with Tag Bits

Can distinguish **two types** – number v bool – with a **one bit**.

Least Significant Bit (LSB) is

- 0 for number
- 1 for boolean



Question: Why not 0 for boolean and 1 for number?

Tagged Numbers

So number is the binary representation shifted left by 1 bit

- Lowest bit is always 0
- Remaining bits are number's binary representation

Value	Binary Repr.	Hex Repr.
3	0b00000110	0x06
5	0b00001010	0x0a
12	0b00011000	0x18
42	0b01010100	0x54

Tagged Booleans

Least Significant Bit (LSB) is

1 for true 0 for false

Value	Binary Repr.	Hex Repr.
true	0b <u>1</u>	0x03
false	0b <u>0</u>	0x01

6 12
3x6

QUIZ: Updating Compilation with Tags

isar

Which parts of the compiler do we need to update to account for tagged data representation?

- multiply
- add 1, sub 1 ✓
- **check type before op!**
- if cmp change! ✓
- equal (actual max value) ✓

- print output ✓
- input ✓

Update Runtime Too!

We need to update the `start.rs` runtime to account for tagged data representation too!

```
fn main() {
  let res: i64 = unsafe { our_code_starts_here() };
  match res {
    3 => println!("true"),
    1 => println!("false"),
    _ => println!("{}", res >> 1),
  }
}
```

Semantics of Numbers and Booleans

Wait a minute!

Program	Result
<code>(= 10 true)</code>	<code>false</code>
<code>(+ 10 true)</code>	<code>err</code>
<code>(if 10 20 30)</code>	<code>.err</code>

Runtime Tag Checking

Avoid garbage results by **checking tags** at runtime.

We have a special `label_error` to jump to on type error.

```
global our_code_starts_here
extern snek_error
label_error:      ;; special error label
  push rsp        ;; save stack pointer
  call snek_error ;; call back into runtime
```

Where `snek_error` in our runtime/`start.rs` just prints an error message and exits the program.

```
#[export_name = "\x01snek_error"]
fn snek_error(code: i64) {
  println!("YIKES! A run-time error {code} 🤪");
  std::process::exit(1);
}
```

QUIZ: Compiler Tag Checking Strategy

Write the asm to check if rax's value has a given <tag>

= 0 (num)
= 1 (bool)

```

mov rbx, rax ; save rax
and rbx, 1 ; extract LSB
cmp rbx, <TAG> ; compare with <tag>
mov rcx, <CODE> ; jump to err if not-eq
cmovne rdi, rcx
jne label.error!

```

Update compile_expr for if, eq, + to check tags for operands.