Boolean expressions

- So far we’ve considered only a trivial arithmetic language $L_{Arith}$
- Let’s extend $L_{Arith}$ with equality tests and Boolean true/false values:

  $$e ::= \cdots \mid b \in \mathbb{B} \mid e_1 == e_2$$

- We write $\mathbb{B}$ for the set of Boolean values $\{true, false\}$
- Basic idea: $e_1 == e_2$ should evaluate to $true$ if $e_1$ and $e_2$ have equal values, $false$ otherwise
What use is this?

- **Examples:**
  - $2 + 2 == 4$ should evaluate to **true**
  - $3 \times 3 + 4 \times 4 == 5 \times 5$ should evaluate to **true**
  - $3 \times 3 == 4 \times 7$ should evaluate to **false**
  - How about $true == true$? Or $false == true$?

- So far, there’s not much we can do.

- We can evaluate a numerical expression for its value, or a Boolean equality expression to true or false

- We can’t write an expression whose result depends on evaluating a comparison.
  - We lack an “if then else” (conditional) operation.

- We also can’t “and”, “or” or negate Boolean values.
Conditionals

- Let’s also add an “if then else” operation:

\[ e ::= \cdots \mid b \in \mathbb{B} \mid e_1 == e_2 \mid \text{if } e \text{ then } e_1 \text{ else } e_2 \]

- We define \( \mathcal{L}_{\text{If}} \) as the extension of \( \mathcal{L}_{\text{Arith}} \) with booleans, equality and conditionals.

- Examples:
  - if true then 1 else 2 should evaluate to 1
  - if 1 + 1 == 2 then 3 else 4 should evaluate to 3
  - if true then false else true should evaluate to false

- Note that if \( e \) then \( e_1 \) else \( e_2 \) is the first expression that makes nontrivial “choices”: whether to evaluate the first or second case.
We consider the Boolean values true and false to be values:

\[ v ::= n \in \mathbb{N} \mid b \in \mathbb{B} \]

and we add the following evaluation rules:

\[
\begin{align*}
  e \Downarrow v & \quad \text{for L}\_\text{If} \\
  e_1 \Downarrow v \quad e_2 \Downarrow v & \quad e_1 \Downarrow v_1 \quad e_2 \Downarrow v_2 \quad v_1 \neq v_2 \\
  e_1 \Downarrow \text{true} & \quad e_1 \Downarrow \text{false} \\
  e \Downarrow \text{true} & \quad e_1 \Downarrow v_1 \\
  \text{if } e \text{ then } e_1 \text{ else } e_2 \Downarrow v_1 & \quad \text{if } e \Downarrow \text{false} \quad e_2 \Downarrow v_2 \\
  & \quad \text{if } e \text{ then } e_1 \text{ else } e_2 \Downarrow v_2
\end{align*}
\]
Extending the interpreter

To interpret $L_{lf}$, we need new expression forms:

```scala
case class Bool(n: Boolean) extends Expr
case class Eq(e1: Expr, e2:Expr) extends Expr
case class IfThenElse(e: Expr, e1: Expr, e2: Expr) extends Expr
```

and different types of values (not just Ints):

```scala
abstract class Value
case class NumV(n: Int) extends Value
case class BoolV(b: Boolean) extends Value
```

(Technically, we could encode booleans as integers, but in general we will want to separate out the kinds of values.)
Extending the interpreter

// helpers
def add(v1: Value, v2: Value): Value =
    (v1,v2) match {
        case (NumV(v1), NumV(v2)) => NumV (v1 + v2)
    }
def mult(v1: Value, v2: Value): Value = ...
def eval(e: Expr): Value = e match {
    // Arithmetic
    case Num(n) => NumV(n)
    case Plus(e1,e2) => add(eval(e1),eval(e2))
    case Times(e1,e2) => mult(eval(e1),eval(e2))
    ... }
Extending the interpreter

// helper
def eq(v1: Value, v2: Value): Value = (v1,v2) match {
case (NumV(n1), NumV(n2)) => BoolV(n1 == n2)
case (BoolV(b1), BoolV(b2)) => BoolV(b1 == b2)
}
def eval(e: Expr): Value = e match {
  ...
case Bool(b) => BoolV(b)
case Eq(e1,e2) => eq (eval(e1), eval(e2))
case IfThenElse(e,e1,e2) => eval(e) match {
  case BoolV(true) => eval(e1)
  case BoolV(false) => eval(e2)
}
}
Aside: Other Boolean operations

- We can add Boolean and, or and not operations as follows:

\[ e ::= \cdots | e_1 \land e_2 | e_1 \lor e_2 | \neg(e) \]

- with evaluation rules:

\[
\begin{array}{c}
\frac{e_1 \Downarrow v_1 \quad e_2 \Downarrow v_2}{e_1 \land e_2 \Downarrow v_1 \land_B v_2} \quad \frac{e_1 \Downarrow v_1 \quad e_2 \Downarrow v_2}{e_1 \lor e_2 \Downarrow v_1 \lor_B v_2}
\end{array}
\]

- where again, \( \land_B \) and \( \lor_B \) are the mathematical “and” and “or” operations

- These are definable in \( L_{lf} \), so we will leave them out to avoid clutter.
Aside: Shortcut operations

- Many languages (e.g. C, Java) offer *shortcut* versions of “and” and “or”:

  \[ e ::= \cdots \mid e_1 \&\& e_2 \mid e_1 \mid\mid e_2 \]

- \( e_1 \&\& e_2 \) stops early if \( e_1 \) is false (since \( e_2 \)’s value then doesn’t matter).

- \( e_1 \mid\mid e_2 \) stops early if \( e_1 \) is true (since \( e_2 \)’s value then doesn’t matter).

- We can model their semantics using rules like this:

  \[
  \begin{align*}
  e_1 \downarrow \text{false} & \quad e_1 \&\& e_2 \downarrow \text{false} \\
  e_1 \downarrow \text{true} & \quad e_1 \downarrow \text{true} \quad e_2 \downarrow v_2 \\
  e_1 \mid\mid e_2 \downarrow \text{true} & \quad e_1 \downarrow \text{false} \quad e_2 \downarrow v_2
  \end{align*}
  \]
What else can we do?

We can also do strange things like this:

\[ e_1 = 1 + (2 == 3) \]

Or this:

\[ e_2 = \text{if } 1 \text{ then } 2 \text{ else } 3 \]

What should these expressions evaluate to?

There is no \( v \) such that \( e_1 \downarrow v \) or \( e_2 \downarrow v \):

- the *Totality* property for \( L_{\text{Arith}} \) fails, for \( L_{\text{If}} \);
- If we try to run the interpreter: we just get an error
One answer: Conversions

- In some languages (notably C, Java), there are built-in conversion rules
  - For example, “if an integer is needed and a boolean is available, convert true to 1 and false to 0”
  - Likewise, “if a boolean is needed and an integer is available, convert 0 to false and other values to true”
  - LISP family languages have a similar convention: if we need a Boolean value, nil stands for “false” and any other value is treated as “true”

- Conversion rules are convenient but can make programs less predictable
- We will avoid them for now, but consider principled ways of providing this convenience later on.
Another answer: Types

- Should programs like:

  \[ 1 + (2 == 3) \quad \text{if} \ 1 \ \text{then} \ 2 \ \text{else} \ 3 \]

  even be allowed?

- Idea: use a \textit{type system} to define a subset of “well-formed” programs

- Well-formed means (at least) that at run time:
  - arguments to arithmetic operations (and equality tests) should be numeric values
  - arguments to conditional tests should be Boolean values
Consider an expression $e$

- If $e = n$, then $e$ has type “integer”
- If $e = e_1 + e_2$, then $e_1$ and $e_2$ must have type “integer”. If so, $e$ has type “integer” also, else error.
- If $e = e_1 \times e_2$, then $e_1$ and $e_2$ must have type “integer”. If so, $e$ has type “integer” also, else error.
Typing rules, informally: booleans, equality and conditionals

- Consider an expression $e$
  - If $e = \text{true}$ or $\text{false}$, then $e$ has type “boolean”
  - If $e = e_1 == e_2$, then $e_1$ and $e_2$ must have the same type. If so, $e$ has type “boolean”, else error.
  - If $e = \text{if } e_0 \text{ then } e_1 \text{ else } e_2$, then $e_0$ must have type “boolean”, and $e_1$ and $e_2$ must have the same type. If so, then $e$ has the same type as $e_1$ and $e_2$, else error.

- Note 1: Equality arguments have the same (unknown) type.
- Note 2: Conditional branches have the same (unknown) type. This type determines the type of the whole conditional expression.
Concise notation for typing rules

- We can define the possible types using a BNF grammar, as follows:

  \[ T{ype} \ni \tau ::= \text{int} \mid \text{bool} \]

  For now, we will consider only two possible types, “integer” (int) and “boolean” (bool).

- We can also use \textit{rules} to describe the types of expressions:

\[ \text{Definition (Typing judgment} \vdash e : \tau \text{)} \]

We use the notation \[ \vdash e : \tau \] to say that \( e \) is a well-formed term of type \( \tau \) (or “\( e \) has type \( \tau \)”.

Typing rules, more formally: arithmetic

- If $e = n$, then $e$ has type "integer"
- If $e = e_1 + e_2$, then $e_1$ and $e_2$ must have type "integer". If so, $e$ has type "integer" also, else error.
- If $e = e_1 \times e_2$, then $e_1$ and $e_2$ must have type "integer". If so, $e$ has type "integer" also, else error.

\[ \vdash e : \tau \quad \text{for } L_{\text{Arith}} \]

\[
\begin{align*}
\text{n} & \in \mathbb{N} & \vdash e_1 : \text{int} & \vdash e_2 : \text{int} \\
\vdash n : \text{int} & & \vdash e_1 + e_2 : \text{int} \\
\vdash e_1 : \text{int} & \vdash e_2 : \text{int} & \vdash e_1 \times e_2 : \text{int}
\end{align*}
\]
Typing rules, more formally: equality and conditionals

We indicate that the types of subexpressions of `==` must be equal by using the same `τ`.

Similarly, we indicate that the result of a conditional has the same type as the two branches using the same `τ` for all three.
Typing judgments: examples

\[ \vdash 1 : \text{int} \quad \vdash 2 : \text{int} \]
\[ \vdash 1 + 2 : \text{int} \quad \vdash 4 : \text{int} \]
\[ \vdash 1 + 2 == 4 : \text{bool} \]

\[ \vdash 1 + 2 == 4 : \text{bool} \quad \vdash 42 : \text{int} \quad \vdash 17 : \text{int} \]
\[ \vdash \text{if } 1 + 2 == 4 \text{ then } 42 \text{ else } 17 : \text{int} \]

\[ \vdash \text{if } 1 + 2 == 4 \text{ then } 42 \text{ else } 17 : \text{int} \quad \vdash 100 : \text{int} \]
\[ \vdash (\text{if } 1 + 2 == 4 \text{ then } 42 \text{ else } 17) + 100 : \text{int} \]
Typing judgments: non-examples

But we also want some things \textbf{not} to typecheck:

\[
\vdash 1 == \text{true} : \tau
\]

\[
\vdash \text{if } 42 \text{ then } e_1 \text{ else } e_2 : \tau
\]

These judgements do not hold for any \(e_1, e_2, \tau\).
Fundamental property of typing

- The point of the typing judgment is to ensure *soundness*: if an expression is well-typed, then it evaluates “correctly.”
- That is, evaluation is well-behaved on well-typed programs.

**Theorem (Type soundness for L_{\text{If}})**

\[ \text{If } \vdash e : \tau \text{ then } e \Downarrow v \text{ and } \vdash v : \tau. \]

- For a language like L_{If}, soundness is fairly easy to prove by induction on expressions. We’ll present soundness for more realistic languages in detail later.
Static vs. dynamic typing

- Some languages proudly advertise that they are “static” or “dynamic”

**Static typing:**
- not all expressions are well-formed; some sensible programs are not allowed
- types can be used to catch errors, improve performance

**Dynamic typing:**
- all expressions are well-formed; any program can be run
- type errors arise dynamically; higher overhead for tagging and checking

- These are rarely-realized extremes: most “statically” typed languages handle some errors dynamically

- In contrast, any “dynamically” typed language can be thought of as a statically typed one with just one type.
Aside: Operators, operators everywhere

- We have now considered several *binary operators*
  
  \[
  + \quad \times \quad \land \quad \lor \quad \equiv
  \]

- as well as a unary one (\(\neg\))

- It is tiresome to write their syntax, evaluation rules, and typing rules explicitly, every time we add to the language

- We will sometimes represent such operations using *schematic syntax* \(e_1 \oplus e_2\) and rules:

  \[
  \frac{e_1 \downarrow v_1 \quad e_2 \downarrow v_2}{e_1 \oplus e_2 \downarrow v_1 \oplus_A v_2} \quad \vdash e_1 : \tau_1 \quad \vdash e_2 : \tau_2 \quad \oplus : \tau_1 \times \tau_2 \rightarrow \tau
  \]

  \[
  \vdash e_1 \oplus e_2 : \tau
  \]

- where \(\oplus : \tau_1 \times \tau_2 \rightarrow \tau\) means that operator \(\oplus\) takes arguments \(\tau_1, \tau_2\) and yields result of type \(\tau\)

- (e.g. \(+ : \text{int} \times \text{int} \rightarrow \text{int}\), \(== : \tau \times \tau \rightarrow \text{bool}\))
Summary

- In this lecture we covered:
  - Boolean values, equality tests and conditionals
  - Extending the interpreter to handle them
  - Typing rules

- Next time:
  - Variables and let-binding
  - Substitution, environments and type contexts