Boolean expressions

- So far we’ve considered only a trivial arithmetic language $L_{\text{Arith}}$.
- Let’s extend $L_{\text{Arith}}$ 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 $\{\text{true}, \text{false}\}$.
- Basic idea: $e_1 == e_2$ should evaluate to $\text{true}$ if $e_1$ and $e_2$ have equal values, $\text{false}$ otherwise.

What use is this?

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

- 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 $L_{\text{If}}$ as the extension of $L_{\text{Arith}}$ with booleans, equality and conditionals.
- Examples:
  - if $\text{true}$ then $1$ else $2$ should evaluate to $1$.
  - if $1 + 1 == 2$ then $3$ else $4$ should evaluate to $3$.
  - if $\text{true}$ then $\text{false}$ else $\text{true}$ should evaluate to $\text{false}$.

- 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:

\[ e \downarrow v \]

- \( e_1 \downarrow v \) \( e_2 \downarrow v \)
- \( e_1 = e_2 \downarrow \text{true} \)
- \( e_1 = e_2 \downarrow \text{false} \)
- \( e \downarrow \text{true} \)
- \( e \downarrow \text{false} \)
- \if e \then e_1 \else e_2 \downarrow v_1 \)
- \if e \then e_1 \else e_2 \downarrow v_2 \)

To interpret \( L_{\text{if}} \), we need new expression forms:

- case class \( \text{Bool}(n: \text{Boolean}) \) extends Expr
- case class \( \text{Eq}(e_1: \text{Expr}, e_2: \text{Expr}) \) extends Expr
- case class \( \text{IfThenElse}(e: \text{Expr}, e_1: \text{Expr}, e_2: \text{Expr}) \) extends Expr

and different types of values (not just Ints):

- abstract class \( \text{Value} \)
- case class \( \text{NumV}(n: \text{Int}) \) extends Value
- case class \( \text{BoolV}(b: \text{Boolean}) \) extends Value

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

```
// helpers
def add(v1: Value, v2: Value): Value =
  (v1,v2) match {
    case (NumV(n1), NumV(n2)) => NumV (n1 + n2)
  }

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))
  ... }
```

```scala
// 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:
  \[
  \begin{array}{c}
  e_1 \downarrow v_1 \quad e_2 \downarrow v_2 \\
  e_1 \land e_2 \downarrow v_1 \land_B v_2 \\
  e_1 \lor e_2 \downarrow v_1 \lor_B v_2 \\
  \end{array}
  \]
- where again, $\land_B$ and $\lor_B$ are the mathematical “and” and “or” operations
- These are definable in $L_{If}$, so we will leave them out to avoid clutter.

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 $L_{Arith}$ fails, for $L_{If}$!
- 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.

Aside: Shortcut operations

- Many languages (e.g. C, Java) offer shortcut versions of “and” and “or”:
  \[
  e ::= \cdots | e_1 \& e_2 | e_1 \| e_2
  \]
- $e_1 \& e_2$ stops early if $e_1$ is false (since $e_2$’s value then doesn’t matter).
- $e_1 \| e_2$ 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:
  \[
  \begin{array}{c}
  e_1 \downarrow \text{false} \\
  e_1 \& e_2 \downarrow \text{false} \\
  e_1 \downarrow \text{true} \\
  e_1 \| e_2 \downarrow \text{true} \\
  \end{array}
  \]
  \[
  \begin{array}{c}
  e_1 \downarrow \text{false} \\
  e_1 \& e_2 \downarrow \text{false} \\
  e_1 \downarrow \text{true} \\
  e_1 \| e_2 \downarrow \text{false} \\
  \end{array}
  \]
Another answer: Types

- Should programs like:
  
  \[ 1 + (2 == 3) \text{ if } 1 \text{ then } 2 \text{ else } 3 \]

  even be allowed?

- Idea: use a 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

Typing rules, informally: arithmetic

- 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:
  
  \[ \text{Type } \ni \tau ::= \text{int} \mid \text{bool} \]

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

- We can also use rules to describe the types of expressions:

  **Definition (Typing judgment \( \vdash e : \tau \))**

  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.

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

\[
\begin{array}{c}
\vdash n : \text{int} \\
\vdash e_1 : \text{int} \quad \vdash e_2 : \text{int} \\
\vdash e_1 + e_2 : \text{int} \\
\end{array}
\]

Typing rules, more formally: equality and conditionals

- We indicate that the types of subexpressions of \( == \) must be equal by using the same \( \tau \)
- Similarly, we indicate that the result of a conditional has the same type as the two branches using the same \( \tau \) for all three.

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

\[
\begin{array}{c}
\vdash b : \text{bool} \\
\vdash e_1 : \text{int} \quad \vdash e_2 : \text{int} \\
\vdash e : \text{bool} \\
\vdash e_1 : \text{int} \quad \vdash e_2 : \text{int} \\
\vdash \text{if } e \text{ then } e_1 \text{ else } e_2 : \tau \\
\end{array}
\]

Typing judgments: examples

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

Typing judgments: non-examples

But we also want some things **not** to typecheck:

\[ \vdash 1 == \text{true} : \tau \]
\[ \vdash \text{if } 42 \text{ then } e_1 \text{ else } e_2 : \tau \]

These judgments 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_{if})

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

- 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.

### 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