CS 476 – Programming Language Design
Questions?
Writing Functions on Syntax

• Step 1: write down what’s in the language in English
• Step 2: write a grammar that describes all possible programs
• Step 3: write a datatype that abstracts the grammar
• Result: a datatype of programs in the language, so we can write functions that operate on programs

• Let’s try it out on a programming language!
Language #1: Expressions

• Simple arithmetic and boolean operations
• Every term computes to a *value*, either int or bool

• Arithmetic operators: plus, minus, times
• Boolean operators: and, or, not, comparison, if-then-else

• \(3 + 5 \times 9\) should compute to 48
• if 1 = 0 or 1 = 1 then 2 else 4 should compute to 2
Expressions: Syntax

- Arithmetic operators: plus, minus, times
- Boolean operators: and, or, not, comparison, if-then-else
Expressions: Syntax

\[ E ::= \# \]

| \( E + E \) | \( E - E \) | \( E \times E \) |
| <bool> |
| \( E \) and \( E \) | \( E \) or \( E \) |
| not \( E \) |
| \( E = E \) |
| if \( E \) then \( E \) else \( E \) |

\[
\text{type exp} = \text{Num of int} \\
\text{Add of exp} \times \text{exp} | \ldots \\
\text{Bool of bool} \\
\text{And of exp} \times \text{exp} | \ldots \\
\text{Not of exp} \\
\text{Eq of exp} \times \text{exp} \\
\text{If of exp} \times \text{exp} \times \text{exp} \times \text{exp}
\]
Interpreters

• An interpreter is a function that takes a program and returns its result

• One way to implement a programming language!
  — Interpreted languages: Python, Javascript, JVM bytecode, ...
  — Alternative to compiling
  — Usually less efficient, but easier to write

• Even for compiled languages, useful as a reference
  — like https://github.com/WebAssembly/spec/tree/master/interpreter
Expressions: Syntax

\[ E ::= \langle\#\rangle \]

\[ \mid E + E \mid E - E \mid E \times E \]

\[ \mid \langle\text{bool}\rangle\]

\[ \mid E \text{ and } E \mid E \text{ or } E \]

\[ \mid \text{not } E \]

\[ \mid E = E \]

\[ \mid \text{if } E \text{ then } E \text{ else } E \]

\[ \text{type } \text{exp} = \text{Num of int} \]

\[ \mid \text{Add of exp \times exp} \mid \ldots \]

\[ \mid \text{Bool of bool} \]

\[ \mid \text{And of exp \times exp} \mid \ldots \]

\[ \mid \text{Not of exp} \]

\[ \mid \text{Eq of exp \times exp} \]

\[ \mid \text{If of exp \times exp \times exp} \]
Expressions: Interpreter

• Every term computes to a value, either int or bool

  type value = IntVal of int | BoolVal of bool

let rec eval (e : exp) : value = (* let rec eval e = *)
  match e with
  | Num i ->
  | Add (e1, e2) ->
  | ...

Expressions: Interpreter

• Every term computes to a value, either int or bool

  type value = IntVal of int | BoolVal of bool

let rec eval (e : exp) : value = (* let rec eval e = *)
  match e with
  | Num i -> IntVal i
  | Add (e1, e2) ->
  | ...

Expressions: Interpreter

• Every term computes to a *value*, either int or bool

  type value = IntVal of int | BoolVal of bool

  let rec eval (e : exp) : value = (* let rec eval e = *)
    match e with
    | Num i -> IntVal i
    | Add (e1, e2) -> eval e1 + eval e2
    | ...          Error: This expression has type value but an expression was expected of type int
let rec eval (e : exp) : value =
  match e with
  | Num i -> IntVal i
  | Add (e1, e2) ->
    (match eval e1, eval e2 with
     | IntVal i1, IntVal i2 -> IntVal (i1 + i2)
     | _, _ -> ?)

• Exercise: What should happen if we try to add things that aren’t integers? (course code: ERXJND)
let rec eval (e : exp) : value =
  match e with
  | Num i -> IntVal i
  | Add (e1, e2) ->
    (match eval e1, eval e2 with
     | IntVal i1, IntVal i2 -> IntVal (i1 + i2)
     | _, _ -> None)

type 'a option = Some of 'a | None
let rec eval (e : exp) : value option =
  match e with
  | Num i -> IntVal i
  | Add (e1, e2) ->
    (match eval e1, eval e2 with
     | IntVal i1, IntVal i2 -> IntVal (i1 + i2)
     | __, __ -> None)

type 'a option = Some of 'a | None
Expressions: Interpreter with Errors

let rec eval (e : exp) : value option =
  match e with
  | Num i -> Some (IntVal i)
  | Add (e1, e2) ->
    (match eval e1, eval e2 with
     | Some (IntVal i1), Some (IntVal i2) -> Some (IntVal (i1 + i2))
     | _, _ -> None)

type 'a option = Some of 'a | None
Questions?

Top
let rec eval (e : exp) : value option =
match e with
| ... |
| Bool b -> Some (BoolVal b) |
| And (e1, e2) ->
  (match eval e1, eval e2 with
   | Some (BoolVal b1), Some (BoolVal b2) ->
     Some (BoolVal (b1 && b2))
   | _, _ -> None)
Expressions: Interpreter with Errors

let rec eval (e : exp) : value option =
  match e with
  | ... |
  | Eq (e1, e2) ->
    (match eval e1, eval e2 with
     | Some (IntVal i1), Some (IntVal i2) -> Some (BoolVal (i1 = i2))
     | ...) |

• What kinds of results should we be able to compare?
Expressions: Interpreting Comparison

let rec eval (e : exp) : value option =
  match e with
  | ... |
  | ... |
  | Eq (e1, e2) ->
    (match eval e1, eval e2 with
     | Some v1, Some v2 -> Some (BoolVal (v1 = v2))
     | ... )
Expressions: Interpreting Comparison

let rec eval (e : exp) : value option =
  match e with
  | ... |
  | ... |
  | Eq (e1, e2) -> Some (BoolVal (eval e1 = eval e2))

• Should we be able to compare ints and bools? Should two erroneous expressions be equal? Depends on what kind of language we want!
let rec eval (e : exp) : value option =
  match e with
  | ... |
  | If (e, e1, e2) ->
    (match eval e with
     | Some (BoolVal b) -> if b then eval e1 else eval e2
     | _ -> None)
Expressions: Int-only Interpreter

let rec eval (e : exp) : int =
match e with
| Num i -> i
| Add (e1, e2) -> eval e1 + eval e2
| Bool b -> if b then 1 else 0
| ...   
| If (e, e1, e2) -> if eval e <> 0 then eval e1 else eval e2

• Simpler interpreter, but behavior may surprise programmers!
Questions?