Interpreters

Nate Foster
Spring 2019

Today’s music: *Step by Step* by New Kids on the Block
Attendance question

A. Dragon
B. Phoenix
C. Unicorn
D. Huh?
The Goal of 3110

Become a better programmer
though study of
programming languages
Review

Previously in 3110:
• functional programming
• modular programming
• data structures

Today:
• new unit of course: interpreters
code as data: the compiler is code that operates on data; that data is itself code
the compiler goes away; not needed to run the program
the interpreter stays; needed to run the program
Compilers:
• primary job is translation
• better performance

vs.

Interpreters:
• primary job is execution
• easier implementation
Architecture

Two phases:
• **Front end:** translate source code into *abstract syntax tree* (AST) then into *intermediate representation* (IR)
• **Back end:** translate AST into machine code

Front end of compilers and interpreters largely the same:
• *Lexical analysis* with lexer
• *Syntactic analysis* with parser
• *Semantic analysis*
Front end

Character stream:

if x=0 then 1 else fact(x-1)

Token stream:

if x = 0 then 1 else fact ( x - 1 )
Front end

Token stream:

```
if x = 0 then 1 else fact (x - 1)
```

Abstract syntax tree:
Front end

Abstract syntax tree:

```
if-then-else
  =
    x
      0
  1
  apply
    fact
    -
    x
      1
```

Semantic analysis:

- accept or reject program
- create *symbol tables* mapping identifiers to types
- *decorate* AST with types
- etc.
Next

Might translate AST into an *intermediate representation* (IR) that is a kind of abstract machine code.

Then:

- **Interpreter** executes AST or IR
- **Compiler** translates IR into machine code
Implementation

Functional languages are well-suited to implement compilers and interpreters

• **Code** easily represented by tree data types
• **Compilation/execution** easily defined by pattern matching on trees
EXPRESSION INTERPRETER
Arithmetic expressions

Goal: write an interpreter for expressions involving integers and addition

Path to solution:
• let's assume lexing and parsing is already done
• need to take in AST and interpret it
• intuition:
  – an expression $e$ takes a single step to a new expression $e'$
  – expression keeps stepping until it reaches a value
let rec step = function
  | Int n -> failwith "Does not step"
  | Add(Int n1, Int n2) -> Int (n1 + n2)
  | Add(e1, e2) -> ???

A. Add(step e1, e2)
B. Add(e1, step e2)
C. Add(step e1, step e2)
D. step e1 + step e2

Hint: given (4+5)+(6+7), what should the first step be?
Arithmetic expressions

**Goal:** extend interpreter to *let* expressions

**Path to solution:**
- extend AST with a variant for *let* and for variables
- add branches to *step* to handle those
- that requires *substitution*...
**let expressions** [from lec 2]

`let x = e1 in e2`

**Evaluation:**

– Evaluate `e1` to a value `v1`
– **Substitute** `v1` for `x` in `e2`, yielding a new expression `e2'`
– Evaluate `e2'` to `v`
– Result of evaluation is `v`
$e^{v/x}$ means $e$ with $v$ substituted for $x$
Substitution

Instead of:
"Substitute \( v_1 \) for \( x \) in \( e_2 \), yielding a new expression \( e_2' \); Evaluate \( e_2' \) to \( v \)"

Write:
"Evaluate \( e_2 \{ v_1 / x \} \) to \( v \)"
Upcoming events

• [Wednesday 11:59pm]: Peer evals due
• [Thursday 11:59pm]: Project charter due

This is open to interpretation.

THIS IS 3110