Interpreters

Prof. Clarkson
Fall 2016

Today's music: Substitute by The Who
Review

Previously in 3110:
• functional programming
• modular programming
• imperative programming

Today:
• new unit of course: interpreters
• substitution model of interpretation
COMPILERS AND INTERPRETERS
code as data: the compiler is code that operates on data; that data is itself code
Compilation

the compiler goes away; not needed to run the program
the interpreter stays; needed to run the program
Compilation vs. interpretation

• Compilers:
  – primary job is *translation*
  – typically lead to better performance of program

• Interpreters:
  – primary job is *execution*
  – typically lead to easier implementation of language
    • maybe better error messages and better debuggers
Mixed compilation and interpretation

The VM is the interpreter; needed to run the program; Java and OCaml can both work this way.
Architecture

Architecture of a compiler is pipe and filter
• Compiler is one long chain of filters, which can be split into two phases
• **Front end**: translate source code into a tree data structure called *abstract syntax tree* (AST)
• **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:

\[
\text{if } x = 0 \text{ then } 1 \text{ else } \text{fact}(x - 1)
\]

Abstract syntax tree:
Front end

Abstract syntax tree:

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

Semantic analysis:
- accept or reject program
- decorate AST with types
- etc.
After the front end

- **Interpreter** begins executing code using the abstract syntax tree (AST)
- **Compiler** begins translating code into machine language
  - Might involve translating AST into a simpler *intermediate representation* (IR)
  - Eventually produce *object code*
Implementation

Functional languages are well-suited to implement compilers and interpreters

- **Code** easily represented by tree data types
- **Compilation** passes easily defined pattern matching on trees
- **Semantics** naturally implemented with language constructs
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*
AST

type expr =
  | Int of int
  | Add of expr * expr

e.g.
  • Int 5 represents the source expression 5
  • Add (Int 5)
    (Add (Int 6) (Int 7))
  represents 5+(6+7)
Evaluation by stepping

(* A single step of evaluation: *)

val step : expr -> expr

(* The reflexive, transitive closure of *)

val multistep : expr -> expr
Multistep

let rec multistep e =
  if is_value e then e
  else multistep (step e)

(* [is_value e] is whether *
 * [e] is a syntactic value *)

let is_value = function
  Int _ -> true
  Add _ -> false
Question

Given \((4+5)+(6+7)\), what should the first step be?

A. \(9+(6+7)\)

B. \((4+5)+13\)
Given \((4+5)+(6+7)\), what should the first step be?

A. \(9+(6+7)\)

B. \((4+5)+13\)

Answer: It doesn't matter!

(especially in the absence of side effects)

But we have to make an implementation choice...
Step, Choice A

```ocaml
let rec step = function
| Int n          -> failwith "Does not step"
| Add(e1, e2)    -> Add(step e1, e2)
```
Step, Choice A

let rec step = function
  | Int n -> failwith "Does not step"
  | Add(e1, e2) -> Add(step e1, e2)
  | Add(Int n1, e2) -> Add(Int n1, step e2)

Stop: we already have a bug
How will 5+(6+7) step?
Step, Choice A

\[
\text{let rec step } = \text{ function}
\]
\[
\begin{align*}
\text{Int n } &\rightarrow \text{ failwith "Does not step"} \\
\text{Add(Int n1, e2) } &\rightarrow \text{ Add(Int n1, step e2)} \\
\text{Add(e1, e2) } &\rightarrow \text{ Add(step e1, e2)}
\end{align*}
\]
Step, Choice A

let rec step = function
  | Int n -> failwith "Does not step"
  | Add(Int n1, Int n2) -> Int (n1+n2)
  | Add(Int n1, e2) -> Add(Int n1, step e2)
  | Add(e1, e2) -> Add(step e1, e2)
let rec step = function
| Int n -> failwith "Does not step"
| Add(Int n1, Int n2) -> Int (n1+n2)
| Add(e1, Int n2) -> Add(step e1, Int n2)
| Add(e1, e2) -> Add(e1, step e2)
EXTENDED EXPRESSION INTERPRETER
Arithmetic expressions

Goal: extend interpreter to \texttt{let} expressions

Path to solution:
• extend AST with a variant for \texttt{let} and for variables
• add branches to \texttt{step} to handle those
• that requires \textit{substitution}...
let expressions [from lec 4]

let \( x = e_1 \) in \( e_2 \)

Evaluation:

- Evaluate \( e_1 \) to a value \( v_1 \)
- Substitute \( v_1 \) for \( x \) in \( e_2 \), yielding a new expression \( e_2' \)
- Evaluate \( e_2' \) to \( v \)
- Result of evaluation is \( v \)
Substitution

• Notation: $e\{v/x\}$ means $e$ with $v$ substituted for $x$
  – e.g., $(x+5)\{4/x\}$ means $(x+5)$ with 4 substituted for $x$
  – which would be $(4+5)$

• In let semantics:
  – Instead of: "Substitute $v_1$ for $x$ in $e_2$, yielding a new expression $e_2'$; Evaluate $e_2'$ to $v$"
  – Could now write: "Evaluate $e_2\{v_1/x\}$ to $v$"
Extended AST

type expr =
  | Int of int
  | Add of expr * expr
  | Var of string
  | Let of string * expr * expr

e.g.
• Var "x" represents the source expression x
• Let "x" (Int 5) (Add (Var "x") (Int 1)) represents let x = 5 in x+1
Multistep

let rec multistep e =
  if is_value e then e
  else multistep (step e)

let is_value = function
  | Int _ -> true
  | Add _ | Var _ | Let _ -> false
Step

let rec step = function
    | Int n -> failwith "Does not step"
    | Add(Int n1, Int n2) -> Int (n1 + n2)
    | Add(Int n1, e2) -> Add(Int n1, step e2)
    | Add(e1, e2) -> Add(step e1, e2)
Step

```ml
let rec step = function
  | Int n -> failwith "Does not step"
  | Add(Int n1, Int n2) -> Int (n1 + n2)
  | Add(Int n1, e2) -> Add (Int n1, step e2)
  | Add(e1, e2) -> Add (step e1, e2)
  | Var _ -> failwith "Unbound variable"
```

Why? Equivalent to just typing "x;;" into fresh utop session
let rec step = function
| Int n -> failwith "Does not step"
| Add(Int n1, Int n2) -> Int (n1 + n2)
| Add(Int n1, e2) -> Add (Int n1, step e2)
| Add(e1, e2) -> Add (step e1, e2)
| Var _ -> failwith "Unbound variable"
| Let(x, e1, e2) -> Let (x, step e1, e2)
Step

let rec step = function
  | Int n -> failwith "Does not step"
  | Add(Int n1, Int n2) -> Int (n1 + n2)
  | Add(Int n1, e2) -> Add (Int n1, step e2)
  | Add(e1, e2) -> Add (step e1, e2)
  | Var _ -> failwith "Unbound variable"
  | Let(x, Int n, e2) -> e2{(Int n)/x}
  | Let(x, e1, e2) -> Let (x, step e1, e2)
Substitution

(* \[\text{subst } e \ v \ x\] is \(e\{v/x\}\), that is,
* \[e\] with \(v\) substituted for \(x\). *)

\[
\begin{align*}
\textbf{let rec subst } & \ e \ v \ x = \ \textbf{match } \ e \ \textbf{with} \\
\text{Var } y & \rightarrow \ \textbf{if } x=y \ \textbf{then } v \ \textbf{else } e \\
\text{Int } n & \rightarrow \ \text{Int } n \\
\text{Add}(el,er) & \rightarrow \\
& \ \text{Add}(\text{subst } el \ v \ x, \ \text{subst } er \ v \ x) \\
\text{Let}(y,ebind,ebody) & \rightarrow \\
& \ \textbf{let } ebind' = \text{subst } ebind \ v \ x \ \textbf{in} \\
& \ \textbf{if } x=y \\
& \ \textbf{then } \text{Let}(y, ebind', ebody) \\
& \ \textbf{else } \text{Let}(y, ebind', \ \text{subst } ebody \ v \ x)
\end{align*}
\]
let rec step = function
    | Int n -> failwith "Does not step"
    | Add(Int n1, Int n2) -> Int (n1 + n2)
    | Add(Int n1, e2) -> Add (Int n1, step e2)
    | Add(e1, e2) -> Add (step e1, e2)
    | Var _ -> failwith "Unbound variable"
    | Let(x, Int n, e2) -> subst e2 (Int n) x
    | Let(x, e1, e2) -> Let (x, step e1, e2)
Upcoming events

• [Mon & Tue] Fall Break
• [Wed] Prelim review for recitations
• [next Thursday] lecture canceled; Prelim 1; make sure you've read the pinned Piazza post

This is not a substitute.

THIS IS 3110
FORMAL SYNTAX
Notation

- The code we've written is one way of defining the syntax and semantics of a language.
- Programming language designers have another more compact notation that's independent of the implementation language of interpreter...
Abstract syntax of expression lang.

\[ e ::= x | i | e + e \]
\[ \quad | \text{let } x = e_1 \text{ in } e_2 \]

\[ e, x, i: \text{ meta-variables } \] that stand for pieces of syntax
- \( e \): expressions
- \( x \): program variables
- \( i \): integers

\[ ::= \text{ and } | \text{ are meta-syntax: used to describe syntax of language} \]

notation is called \textit{Backus-Naur Form} (BNF) from its use by Backus and Naur in their definition of Algol-60
Backus and Naur

John Backus (1924-2007)
ACM Turing Award Winner 1977
“For profound, influential, and lasting contributions to the design of practical high-level programming systems”

Peter Naur (b. 1928)
ACM Turing Award Winner 2005
“For fundamental contributions to programming language design”
Abstract syntax of expr. lang.

e ::= x | i | e+e
    | let x = e1 in e2

Note how closely the BNF resembles the OCaml variant we used to represent it!