Interpreters

Prof. Clarkson
Summer 2016

Today’s music: Substitute by The Who
Review

Previously in 3110:
• functional programming: most of the core functional language of OCaml in just three days!
• syntax and semantics of language features

Today:
• implementing interpreters based on semantics and substitution
COMPILERS AND INTERPRETERS
Compilation

Source program

Compiler

Target program

*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

- **Source program**
- **Compiler**
- **Intermediate program**
- **Virtual machine**
  - **Input** → **Virtual machine** → **Output**

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:

```
if x = 0 then 1 else 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

• Tree data types
• Functions defined by pattern matching on trees
• Semantics leads naturally to implementation with functions
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*

**Solution:** see `interp1.ml`
Arithmetic expressions

Goal: extend interpreter to \texttt{let} expressions

Path to solution:
• extend AST with a variant for \texttt{let}
• add a branch to \texttt{step} to handle \texttt{let}
• that requires \textit{substitution}...
let expressions [from lec 3]

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
Arithmetic expressions

**Goal:** extend interpreter to \texttt{let} expressions

**Path to solution:**
- extend AST with a variant for \texttt{let}
- add a branch to \texttt{step} to handle \texttt{let}
- that requires \texttt{substitution}...
- hence a \texttt{substitution model} interpreter

**Solution:** see \texttt{interp2.ml}