Interpreters

Prof. Clarkson
Fall 2015

Today's music: Substitute by The Who
Review

Previously in 3110:
• functional programming
• modular programming

Today:
• new unit of course: interpreters
• substitution model of interpretation
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  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:

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

Abstract syntax tree:

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

Parser
Front end

Abstract syntax tree:

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

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 **let** expressions

**Path to solution:**
- extend AST with a variant for **let**
- add a branch to **step** to handle **let**
- that requires **substitution**...
let expressions [from lec 3]

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 \( v \)
Arithmetic expressions

Goal: extend interpreter to let expressions

Path to solution:
• extend AST with a variant for let
• add a branch to step to handle let
• that requires substitution...
• hence a substitution model interpreter

Solution: see interp2.ml
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...
FORMAL SYNTAX
Abstract syntax of expression lang.

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

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

::= and | are meta-syntax: used to describe syntax of language

notation is called 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.

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

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

• [next Thursday] A3 due

This is not a substitute.

THIS IS 3110