Lecture 21: Interpreters I

Marvin Zhang
07/27/2016

Announcements

Roadmap

Introduction
Functions
Data
Mutability
Objects
Interpretation
Paradigms
Applications

• This week (Interpretation), the goals are:
  • To learn a new language, Scheme, in two days!
  • To understand how interpreters work, using Scheme as an example

Introduction

• Computers can execute programs written in many different programming languages. How?
• Computers only deal with machine languages (0s and 1s), where statements are direct commands to the hardware
• Programs written in languages like Python are compiled, or translated, into these machine languages
• Python programs are first compiled into Python bytecode, which has the benefit of being system-independent
• You can look at Python bytecode using the dis module

```
Python 3
def square(x):
    return x * x
```

```
Python 3 Bytecode
def square(x):
    return x * x
```

```
Python 3
from dis import dis
dis(square)
```

```
Python 3 Bytecode
from dis import dis
```

Programming Languages (demo)

Interpretation

• Compilers are complicated, and the topic of future courses
• In this course, we will focus on interpreters, programs that execute other programs written in a particular language
• The Python interpreter is a program written in C
  • After compiling it to machine code, it can be run to interpret Python programs
  • The last project in this course is to write a Scheme interpreter in Python
  • The Scheme interpreter can then be run using the Python interpreter to interpret Scheme programs
• To create a new programming language, we either need a:
  • Specification of the syntax and semantics of the language
  • Canonical implementation of either a compiler or interpreter for the language

The Scheme Interpreter

• An interpreter for Scheme must take in text (Scheme code) as input and output the values from interpreting the text

Text → Parser → Expressions → Evaluator → Values

• The job of the parser is to take in text and perform syntactic analysis to convert it into expressions that the evaluator can understand
• The job of the evaluator is to read in expressions and perform semantic analysis to evaluate the expressions and output the corresponding values
Building an interpreter for a language is a lot of work. Today, we'll build an interpreter for a subset of Scheme. We will support +, -, *, /, integers, and floats. We will call this simple language Calculator. In lab, discussion, and next lecture, we will look at more complicated examples.

```
calc> (/ (+ 8 7) 5) 3.0
```

```
calc> (+ (* 3
(+ (* 2 4)
(+ 3 5)))
(+ (= 10 7)
6)) 57
```

---

**Parsing**

From text to expressions

---

**Lexical Analysis**

- Tokenization takes in a string and converts it into a list of tokens by splitting on whitespace.
- This step also removes excess whitespace.
- An error is raised if the number of open and closed parentheses are unequal.
- Each token is checked iteratively to ensure it is valid.
  - For Calculator, each token must be a parenthesis, an operator, or a number.
  - Otherwise, an error is raised.

---

**Syntactic Analysis**

- Syntactic analysis uses a read function to identify the hierarchical structure of an expression.
- Each call to the read function consumes the input tokens for exactly one expression, and returns the expression.

```python
def read_exp(tokens):
    """Returns the first calculator expression."""
    ...

def read_tail(tokens):
    """Reads up to the first mismatched close parenthesis."""
    ...
```

```
["(", "+", 1, "+", "-", 23, "]", "+", "*", 4, 5.6, "]", "]"]
```

Resulting expression:

---

**Evaluation**

From expressions to values
Evaluation

- Evaluation is performed by an evaluate function, which takes in an expression (the output of our parser) and computes and returns the value of the expression.
  - In Calculator, the value is always an operator or a number.
- If the expression is primitive, we can return the value of the expression directly.
- Otherwise, we have a call expression, and we follow the rules for evaluating call expressions:
  1. Evaluate the operator to get a function.
  2. Evaluate the operands to get their values.
  3. Apply the function to the values of the operands to get the final value.
- This hopefully looks very familiar!

The Evaluate and Apply Functions

```python
def calc_eval(exp):
    if isinstance(exp, Pair):
        return calc_apply(calc_eval(exp.first), list(exp.second.map(calc_eval)))
    elif exp in OPERATORS:
        return OPERATORS[exp]
    else:
        return exp

def calc_apply(op, args):
    return op(*args)
```

- Why define `calc_apply`? It’s not really necessary, since the Calculator language is so simple.
- For real languages, applying functions is more complex.
- With user-defined functions, the apply function has to call the evaluate function! This mutual recursion is called the eval-apply loop.

Putting it all together

A Calculator interactive interpreter!

The Read–Eval–Print Loop

- Interactive interpreters all follow the same interface:
  1. Print a prompt.
  2. Read text input from the user.
  3. Parse the input into an expression.
  4. Evaluate the expression into a value.
  5. Report any errors, if they occur, otherwise.
  6. Print the value and return to step 1.
- This is known as the read–eval–print loop (REPL).

Handling Exceptions

- Various exceptions may be raised throughout the REPL:
  - Lexical analysis: The token 2.3.4 raises SyntaxError.
  - Syntactic analysis: A misplaced } raises SyntaxError.
  - Evaluation: No arguments to \(-\) raises TypeError.

- An interactive interpreter prints information about each error that occurs.
- A well-designed interactive interpreter should not halt completely on an error, so that the user has an opportunity to try again in the current environment.

Summary

- We built an interpreter today!
  - It was for a very simple language, but the same ideas and principles will allow us to build an interpreter for Scheme, a much more complicated language.
  - More complicated examples are coming soon.
- Interpreters are separated into a parser and an evaluator:
  - The parser takes in text input and outputs the corresponding expressions, using tokens as a midpoint.
  - The evaluator takes in an expression and outputs the corresponding value.
  - The read–eval–print loop completes our interpreter.