CSCI 275: Programming Abstractions

Lecture 16: MiniScheme Start

Fall 2024

Stephen Checkoway
Slides from Molly Q Feldman

Functional Language of the Week: Kotlin

- Started by JetBrains
  - Industry problem, industry solution
  - JetBrains makes lots of SE tools (e.g. IntelliJ, PyCharm IDEs)
  - 28th on the top 50 languages list
  - Open Source, funded by JetBrains, Google, etc.

Main use case? **Android programming!**

- Since 2019, preferred Android development language
  https://developer.android.com/kotlin

Functional Language of the Week: Kotlin

https://play.kotlinlang.org/byExample/04_functional/02_Lambdas

```
// All examples create a function object that performs upper-casing.
// So it's a function from String to String

val upperCase1: (String) -> String = { str: String -> str.uppercase() } // 1

val upperCase2: (String) -> String = { str -> str.uppercase() }         // 2

val upperCase3 = { str: String -> str.uppercase() }                     // 3

// val upperCase4 = { str -> str.uppercase() }                          // 4

val upperCase5: (String) -> String = { it.uppercase() }                 // 5

val upperCase6: (String) -> String = String::uppercase                  // 6

println(upperCase1("hello"))
println(upperCase2("hello"))
println(upperCase3("hello"))
println(upperCase5("hello"))
println(upperCase6("hello"))
```

Showcases type inference (i.e. inferring types in a language that is statically typed)

Interplay between “research features” and language design? Read more: https://blog.sigplan.org/2022/05/19/language-design-in-the-real-world/

MiniScheme

MiniScheme Project

You're going to *build an interpreter* for a subset of Scheme (called MiniScheme)

What does an interpreter do? *Executes* a program

Grammar

We need a way to specify the language of a valid program

Parser

We need to determine if a given program is valid

Evaluator

We need to evaluate a given program

Interpreters You’ve Encountered

- Python interpreter
- DrRacket interpreter

Why does this matter?

Languages are written by people.

You can write languages.

You have the power to make interesting decisions.

Why does this matter?

Languages are written by people.

You can write languages.

You have the power to make interesting decisions.

Here are some examples.

https://www.youtube.com/watch?v=sH4XF6pKKmk
and the Bernhardt talk mentioned

DrRacket Interpreter

The DrRacket Interpreter is a REPL

Evaluate the parsed terms into their final expressions.

(+ 1 2) becomes 3.

Read Eval Print Loop

Read in the characters that the user types. *Parse* them into terms that make sense to Racket

Show them to the user!

So what are you going to do?

You're going to *build an interpreter for MiniScheme!*

The project has two primary functions:

`(parse exp)` creates a tree structure that represents the expression `exp`

`(eval-exp tree environment)` evaluates the given expression `tree` within the given `environment` and returns its value

MiniScheme Project

You’re going to *build an interpreter* for a subset of Scheme (called MiniScheme)

What does an interpreter do? *Executes* a program

Grammar

We need a way to specify the language of a valid program

Parser

We need to determine if a given program is valid

Evaluator

We need to evaluate a given program

How do we understand what a program is?

e.g.

Why do we say

```
(if (< 2 3) 3 4)
```

is a program, but

```
< 2 ( if 3) ( 4 3)
```

and å®çéπ¬

are not?

Yes, this feels philosophical. But think about it concretely.

Things we need to understand programs

- Set of symbols
- Rules for combining the symbols

With those ideas, *certain symbols can elicit certain meanings*

PEOPLE make these rules!

The fact (< 2 3) is a valid program in Racket, but not in Python, comes from this idea

Grammars: Set of Symbols & Rules

A grammar for a language is a (mathematical) tool for specifying which words over an alphabet belong to the language

Grammars are often used to determine the meaning of words in the language

Grammars are *very old*!

- Dating back to at least the Indian linguist Yāska (7th–5th century BCE)
- One of many ways Programming Languages borrows from Natural Language

Grammars, slightly more formally

A grammar is a set of rules that describe how to *generate* a string

Grammars have **three basic components**

- A set of variables or **nonterminals** which *expand* into strings
- A set of **terminal symbols** from which the final word is to be constructed
- A set of **production rules** which describe how a nonterminal can be expanded

Example: Variables = $\{S, A\}$; terminals = $\{\mathtt{x}, \mathtt{z}\}$

$S \rightarrow \mathtt{x}S\mathtt{x}$

$S \rightarrow A$

$A \rightarrow \mathtt{z}A$

$A \rightarrow \mathtt{z}$

You will (or have!) spent a lot of time with grammars in CSCI 383: Theory of Computation

Why do we care in *this* class?

We're going to specify a grammar for MiniScheme

We'll use this to:

- Communicate what needs to be implemented in each part of the project

- Make sure we know what a valid program does (or does not) look like

MiniScheme's Full Grammar

EXP → number
  | symbol
  | (`if` *EXP* *EXP* *EXP*)
  | (`let` (*LET-BINDINGS*) *EXP*)
  | (`letrec` (*LET-BINDINGS*) *EXP*)
  | (`lambda` (*PARAMS*) *EXP*)
  | (`set!` symbol *EXP*)
  | (`begin` EXP^*)
  | (EXP^+)

LET-BINDINGS → $LET\text{-}BINDING^*$

LET-BINDING → [symbol *EXP*]

PARAMS → symbol^*

* Means 0 or more times

\+ means 1 or more

Can

```
(if (if 0 1 2)
    (if 3 4 5)
    (if x y z))
```

be generated by the grammar for MiniScheme?

A. Yes

B. No. `(if …)` cannot appear as the first expression of another `if`

C. No. `(if …)` cannot appear as the "then" or "else" expressions in another `if`

D. No. `x`, `y`, and `z` aren't defined

EXP → number
| symbol
| (`if` *EXP EXP EXP*)
| (`let` (*LET-BINDINGS*) *EXP*)
| (`letrec` (*LET-BINDINGS*) *EXP*)
| (`lambda` (*PARAMS*) *EXP*)
| (`set!` symbol *EXP*)
| (`begin` EXP^*)
| (EXP^+)

LET-BINDINGS → $LET\text{-}BINDING^*$

LET-BINDING → [symbol *EXP*]

PARAMS → $symbol^*$

Are we done? No!

Challenge: Syntactically valid but semantically invalid

Consider the invalid Scheme program

```
(let ([x 5]
      [y 32])
  (+ z 2))
```

This is *syntactically* valid - i.e., it's a valid string generated by the MiniScheme grammar but *semantically* meaningless.

Stay tuned about how we fix this!

Shape of the Task & The Content

- We will be working on MiniScheme **one part at a time**
- We’ll implement the language incrementally, building the grammar as we go

Now let’s do it!
By the end of next week, we’ll be able to do `(+ 1 2)` evaluates to `3`

In Python, how do we know what `1 * 2 + 3` means?

A. We look up the term “1 * 2 + 3” in a dictionary

B. Python finds `+` as the addition operator, `*` as the multiplication operator, and applies them

C. B *and* we have order of operation rules

D. Something else in addition to C

E. None of the above

We *parse* terms before we evaluate them

- It is *so much easier* to be able to have one format to distinguish `1 * 2 + 3` from `1 * (2 + 3)`

- It is great to have a format that is consistent across the whole language

Parsing creates a tree structure for language syntax called an *abstract syntax tree*

MiniScheme programs are straightforward to parse!

Consider the program

```
(let ([x 10]
      [y 20])
  (+ x y))
```

Everything is prefix notation & everything is a structured list!

This is just a structured list containing the symbols `let`, `f`, `x`, `y`, and `+` and the numbers 10 and 20

Start simple: only numbers

Start simple: only numbers

EXP → number parse into `lit-exp`

We're going to need a data type to represent literal expression (and the only type of literals we have are numbers)

We're going to want something which gives

```
(lit-exp num) ; constructor
(lit-exp? exp) ; recognizer
(lit-exp-num exp) ; accessor
```

Putting them together

```
> (parse 107)
(lit-exp 107)

> (lit-exp 107)
(lit-exp 107)

> (eval-exp (lit-exp 107) empty-env)
107

> (eval-exp (parse 107) empty-env)
107
```

Practically, how to implement MiniScheme

For each new type of expression:

- Add a new data type
  - `ite-exp`
  - `let-exp`
  - etc.
- Modify `parse` to produce those
- Modify `eval-exp` to interpret them

EXP → number
  | symbol
  | (`if` *EXP EXP EXP*)
  | (`let` (*LET-BINDINGS*) *EXP*)
  | (`letrec` (*LET-BINDINGS*) *EXP*)
  | (`lambda` (*PARAMS*) *EXP*)
  | (`set!` symbol *EXP*)
  | (`begin` *EXP**)
  | (*EXP EXP**)

LET-BINDINGS → *LET-BINDING**

LET-BINDING → [symbol *EXP*]

PARAMS → symbol*