Building an Evaluator

Michael McThrow

San Jose State University
Computer Science Department
CS 152 – Programming Paradigms

September 22, 2021

Table of Contents

Table of Contents

What Is an Interpreter?

- An interpreter (a.k.a., *evaluator*) evaluates a sequence of expressions.

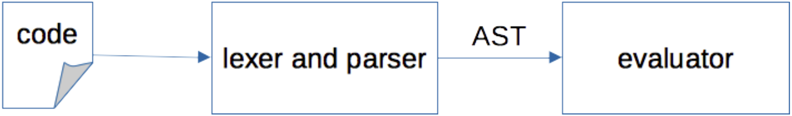
What Is an Interpreter?

- An interpreter (a.k.a., *evaluator*) evaluates a sequence of expressions.
- The difference between an interpreter and a compiler is that an interpreter runs the program, while a compiler translates expressions to another language, usually assembly language or machine code.

What Is an Interpreter?

- An interpreter (a.k.a., *evaluator*) evaluates a sequence of expressions.
- The difference between an interpreter and a compiler is that an interpreter runs the program, while a compiler translates expressions to another language, usually assembly language or machine code.
- CS 152 focuses on interpreters; however, we will explore the basics of compilation next week. CS 153 is a full-fledged course on compilers.

Flowchart



You already have experience writing an interpreter via Project 1, where you wrote a calculator that can handle postfix and infix expressions. Now, let's walk through how you'd write an interpreter for a full-fledged programming language: Scheme.

Table of Contents

Basic Scheme Functions

- All operations in Scheme, such as `define`, `cond`, `car`, `display`, are function calls.
- But which functions must be built (hardcoded) into the interpreter?

Built-in Scheme Functions

It turns out that we can write a minimal Scheme interpreter that implements the following built-in (hardcoded) functions:

- `define`
- `lambda`
- `quote`
- `if` or `cond`
- `cons`, `car`, `cdr`.
- Equality and inequality functions
- Logical operators (e.g., `and`, `or`, `not`)
- Basic arithmetic operators
- Type predicates

How do we go about writing a Scheme interpreter?

Step 1: Parsing

A Scheme program is a sequence of S-expressions. Each S-expression has the following (simplified) grammar:

```
<S-expr> ::= <atomic-symbol>
           | '(' <S-expr> '.' <S-expr> ')'
           | '(' (<S-expr>)+ ')'
```

where `<atomic-symbol>` could be an alphanumerical value with some special characters supported. Note that the special quote syntax is not in this grammar definition. Note that there is an odd exception: `(1 2 3 . 4)` is valid in Scheme, which is equivalent to `(cons 1 (cons 2 (cons 3 4)))`.

Step 1: Parsing

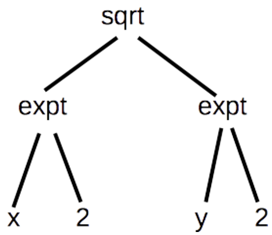
Thankfully, for Project 3, you don't have to write your own S-expression parser; Scheme has a built-in one called `read` that inputs a string and outputs an S-expression that is a list of symbols.

Step 2: Evaluating the AST

Another big advantage of building an interpreter in Scheme: S-expressions make a nice AST.

Example of an AST

(sqrt (expt x 2) (expt y 2))



Example of an AST

If we weren't using Scheme, we'd have to construct our own AST by traversing the parse tree. Here is a possible Java example:

```
// FuncCall, Symbol, and Number all implement
// the AST interface
AST ast = new FuncCall(new Symbol("sqrt"),
 new ArgsList(new FuncCall(new Symbol("expt"), args1),
                          new ASTList(
                           new Symbol("x"),
                           new Number(2))),
             new FuncCall(new Symbol("expt"),
                          new ASTList(
                           new Symbol("y"),
                           new Number(2)))));
```

Environments

- Remember our last lesson on environments? Environments are crucial to the construction of a Scheme evaluator.

Environments

- Remember our last lesson on environments? Environments are crucial to the construction of a Scheme evaluator.
- Recall that an environment consists of a frame (a table of mappings of names to values) and a reference to its enclosing environment.

Example of a Java Implementation of an Environment

```
public class Environment {
  private HashMap<String, AST> frame;
  private Environment enclosing;
}
```

Note that you have many implementation choices; for example, the frame doesn't have to be a HashMap (it could be any type of data structure that enables lookups), and you don't need to have a literal reference/pointer to an Environment object; you could assign each environment an ID value and maintain a global mapping between IDs and Environment objects.

Environments

The global environment will remain throughout the lifetime of the interpreter. Recall that the global environment has no enclosing environment.

Let's begin evaluating simple Scheme expressions, starting with simple literals.

Numbers and Boolean Values

- Numbers evaluate to themselves, just like in the Project 1 calculator.
  - Example: `5` ⇒ `5`
  - Example: `3.141593` ⇒ `3.141593`
  - Example: `-2.4` ⇒ `-2.4`
- Boolean values `#t` and `#f` also evaluate to themselves

Unquoted Symbols

What happens when the interpreter encounters an unquoted symbol, such as `x` or `PI`?

Unquoted Symbols

What happens when the interpreter encounters an unquoted symbol, such as `x` or `PI`?

The interpreter performs an environment lookup in order to get the value associated with the symbol.

Environment Lookup in Java

```
// method inside Environment class
public AST lookup(String name) {
  AST value = frame.get(name);
  if (value != null)
    return value;
  else if (enclosing != null)
    return enclosing.lookup(name);
  else throw new SymbolNotFoundException();
}
```

Function Calls

How can an interpreter tell in Scheme whether an expression is a function call?

Function Calls

How can an interpreter tell in Scheme whether an expression is a function call?

Any time the interpreter sees an unquoted list, which can be recognized by its parentheses.

Function Call Evaluation

How does the interpreter evaluate a function call?

Function Call Evaluation

How does the interpreter evaluate a function call?

1. Create a new environment with an empty frame and where the enclosing environment is the current environment.

Function Call Evaluation

How does the interpreter evaluate a function call?

1. Create a new environment with an empty frame and where the enclosing environment is the current environment.
2. The first element of the list is the function name. The rest of the elements, if any, make up the function's arguments.

Function Call Evaluation

How does the interpreter evaluate a function call?

1. Create a new environment with an empty frame and where the enclosing environment is the current environment.
2. The first element of the list is the function name. The rest of the elements, if any, make up the function's arguments.
3. Perform a lookup of the name of the function and return its value.

Function Call Evaluation

How does the interpreter evaluate a function call?

1. Create a new environment with an empty frame and where the enclosing environment is the current environment.
2. The first element of the list is the function name. The rest of the elements, if any, make up the function's arguments.
3. Perform a lookup of the name of the function and return its value.
4. If the value is a built-in, then perform the evaluation rules of that built-in.

Function Call Evaluation

How does the interpreter evaluate a function call?

1. Create a new environment with an empty frame and where the enclosing environment is the current environment.
2. The first element of the list is the function name. The rest of the elements, if any, make up the function's arguments.
3. Perform a lookup of the name of the function and return its value.
4. If the value is a built-in, then perform the evaluation rules of that built-in.
5. Else, if the value is an anonymous function, then perform the evaluation rules for a `lambda` (will describe later).

Function Call Evaluation

How does the interpreter evaluate a function call?

1. Create a new environment with an empty frame and where the enclosing environment is the current environment.
2. The first element of the list is the function name. The rest of the elements, if any, make up the function's arguments.
3. Perform a lookup of the name of the function and return its value.
4. If the value is a built-in, then perform the evaluation rules of that built-in.
5. Else, if the value is an anonymous function, then perform the evaluation rules for a `lambda` (will describe later).
6. Else, throw an error since the value is not a function.

`define` Built-in

Definition (`define`)

`(define name expr)` creates a binding of the key `name` to the value `expr`. `(define (function-name x1 ... xN) body)` is syntactic sugar for `(define function-name (lambda (x1 ... xN) body))`.

`define` Evaluation

1. Check to see if the number of parameters to the call to `define` is equal to 2. If it is not equal to 2, throw an error.

`define` Evaluation

1. Check to see if the number of parameters to the call to `define` is equal to 2. If it is not equal to 2, throw an error.
2. If the first parameter is a list, then we know `define` is defining a new function. Create a new anonymous function where its parameters consist of all elements of the list except the first, and where the body consists of the second parameter of the `define` call.

`define` Evaluation

1. Check to see if the number of parameters to the call to `define` is equal to 2. If it is not equal to 2, throw an error.
2. If the first parameter is a list, then we know `define` is defining a new function. Create a new anonymous function where its parameters consist of all elements of the list except the first, and where the body consists of the second parameter of the `define` call.
3. Else, if the first parameter is a symbol, then evaluate `expr` and keep its result as a variable.

`define` Evaluation

1. Check to see if the number of parameters to the call to `define` is equal to 2. If it is not equal to 2, throw an error.
2. If the first parameter is a list, then we know `define` is defining a new function. Create a new anonymous function where its parameters consist of all elements of the list except the first, and where the body consists of the second parameter of the `define` call.
3. Else, if the first parameter is a symbol, then evaluate `expr` and keep its result as a variable.
4. No matter what, in the current environment, assign the key `name` or `function-name` to its value.

Evaluating a Function Call to an Anonymous Function

How do we evaluate function calls like `((lambda (x y) (+ x x y y)) 2 3)`?

Evaluating a Function Call to an Anonymous Function

1. Check if the number of arguments is equal to the number of parameters. If they are not equal, throw an error.

Evaluating a Function Call to an Anonymous Function

1. Check if the number of arguments is equal to the number of parameters. If they are not equal, throw an error.
2. For each parameter, in the current environment assign each parameter (the key) to its argument (the value; for example, $x = 2$ and $y = 3$).

Evaluating a Function Call to an Anonymous Function

1. Check if the number of arguments is equal to the number of parameters. If they are not equal, throw an error.
2. For each parameter, in the current environment assign each parameter (the key) to its argument (the value; for example, $x = 2$ and $y = 3$).
3. Evaluate the body of the anonymous function.

Evaluating `quote`

- When encountering a call to the `quote` function, do not evaluate its parameter.
- What `quote` means is to leave whatever is inside unevaluated.
- `(quote ())` is how Scheme defines empty lists.
- In full-fledged Scheme implementations, the ' character is used as shorthand for `quote`, but this is not required in Project 3.

Metacircular Evaluators

Definition (Metacircular Evaluator)

An evaluator that is said to be *metacircular* is one that is implemented in the same language that is being interpreted.

Project 3 is a metacircular evaluator; you will be writing your Scheme interpreter in Scheme.

eval and apply

Scheme and other Lisp languages offer an `eval` function that evaluates any S-expression and a `apply` function that performs a function call given a function and its arguments.

```
; Note that eval requires
; an environment
(eval '(+ 1 2 3) env) ; returns 6
(apply + '(1 2 3)) ; returns 6
```

The Power of `eval` and `apply`

The functions `eval` and `apply` make it possible for Scheme programs to arbitrarily execute Scheme expressions that are not part of the source code, which can be very powerful.

The Power of `eval` and `apply`

The functions `eval` and `apply` make it possible for Scheme programs to arbitrarily execute Scheme expressions that are not part of the source code, which can be very powerful. In Project 3, you will be writing your own `eval` and `apply` functions called `my-eval` and `my-apply`, respectively.

Table of Contents

Agenda for Next Lecture:

- Discuss Project 1 grades and answers.
- Discuss some implementation tips for Project 3, including vector data types in Scheme.
- Go over examples of evaluating Scheme expressions using environments.