CS 61A Summer 2024

Interpreters

Discussion 9: July 25, 2024

More Scheme

The `define` form is used to assign values to symbols. It has the following syntax:

```
(define <symbol> <expression>)
```

```
scm> (define pi (+ 3 0.14))
pi
scm> pi
3.14
```

To evaluate the `define` expression:

1. Evaluate the final sub-expression (`<expression>`), which in this case evaluates to `3.14`.
2. Bind that value to the symbol (`symbol`), which in this case is `pi`.
3. Return the symbol.

The `define` form can also define new procedures, described in the "Defining Functions" section. The `define` form can create a procedure and give it a name:

```
(define (<symbol> <param1> <param2> ...) <body>)
```

For example, this is how we would define the `double` procedure:

```
scm> (define (double x) (* x 2))
double
scm> (double 3)
6
```

Here's an example with three arguments:

```
scm> (define (add-then-mul x y z)
       (* (+ x y) z))
scm> (add-then-mul 3 4 5)
35
```

When a `define` expression is evaluated, the following occurs: 1. Create a procedure with the given parameters and `<body>`. 2. Bind the procedure to the `<symbol>` in the current frame. 3. Return the `<symbol>`.

The following two expressions are equivalent:

```
scm> (define add (lambda (x y) (+ x y)))
add
scm> (define (add x y) (+ x y))
add
```

Atomic expressions (also called *atoms*) are expressions without sub-expressions, such as numbers, boolean values, and symbols.

```
scm> 1234    ; integer
1234
scm> 123.4   ; real number
123.4
scm> #f      ; the Scheme equivalent of False in Python
#f
```

A Scheme *symbol* is equivalent to a Python name. A symbol evaluates to the value bound to that symbol in the current environment. (They are called symbols rather than names because they include `+` and other arithmetic symbols.)

```
scm> quotient      ; A symbol bound to a built-in procedure
#[quotient]
scm> +             ; A symbol bound to a built-in procedure
#[+]
```

In Scheme, *all* values except `#f` (equivalent to `False` in Python) are true values (unlike Python, which has other false values, such as `0`).

```
scm> #t
#t
scm> #f
#f
```

The `if` special form evaluates one of two expressions based on a predicate.

```
(if <predicate> <if-true> <if-false>)
```

The rules for evaluating an `if` special form expression are as follows:

1. Evaluate the `<predicate>`.
2. If the `<predicate>` evaluates to a true value (anything but `#f`), evaluate and return the value of the `<if-true>` expression. Otherwise, evaluate and return the value of the `<if-false>` expression.

For example, this expression does not error and evaluates to 5, even though the sub-expression `(/ 1 (- x 3))` would error if evaluated.

```
scm> (define x 3)
x
scm> (if (> (- x 3) 0) (/ 1 (- x 3)) (+ x 2))
5
```

The `<if-false>` expression is optional.

```
scm> (if (= x 3) (print x))
3
```

Let's compare a Scheme `if` expression with a Python `if` statement:

- In Scheme:

```
(if (> x 3) 1 2)
```

- In Python:

```
if x > 3:
    1
else:
    2
```

The Scheme `if` expression evaluates to a number (either 1 or 2, depending on `x`). The Python statement does not evaluate to anything, and so the 1 and 2 cannot be used or accessed.

Another difference between the two is that it's possible to add more lines of code into the suites of the Python `if` statement, while a Scheme `if` expression expects just a single expression in each of the `<if-true>` and `<if-false>` positions.

One final difference is that in Scheme, you cannot write `elif` clauses.

Q1: Increasing Rope

Definition: A *rope* in Scheme is a non-empty list containing only numbers except for the last element, which may either be a number or a rope.

Implement `up`, a Scheme procedure that takes a positive integer `n`. It returns a rope containing the digits of `n` that is the shortest rope in which each pair of adjacent numbers in the same list are in increasing order.

Reminder: the `quotient` procedure performs floor division, like `//` in Python. The `remainder` procedure is like `%` in Python.

```
(define (up n)
    (define (helper n result)
        (if (zero? n) result
            (helper
                (quotient n 10)
                (let ((first (remainder n 10)))
                  'YOUR-CODE-HERE




                    ))))
    (helper
      (quotient n 10)
      'YOUR-CODE-HERE


    ))

(expect (up 314152667899) (3 (1 4 (1 5 (2 6 (6 7 8 9 (9)))))))
```

Compare `first` to `(car result)` to decide whether to `cons` the value `first` onto the `result` or whether to form a new list that contains `first` and `result` as elements.

To correctly call `helper` from within `up`, build a rope that only contains the last digit of `n`: `(remainder n 10)`.

Q2: (Tutorial) Interpreters Review

Discuss the follow questions with your tutorial group - they will be helpful for your understanding of the Scheme project! If you wish to take notes, we recommend you take notes on a separate document so it won't accidentally get erased.

What are the four parts of an interpreter (Hint: what does REPL stand for)? What does each part do? What parts did you work on implementing in the discussion?

For the Calculator interpreter implemented in discussion, for the following executed code, what would be the input into the "Read" portion of the interpreter?

```
calc> (+ 2 3)
5
```

What would be the output of the "Read" portion for the same code?

How does the evaluate stage work in Calculator? How do we know if an input into `calc_eval` is a call expression?

Interpreters

An interpreter is a program that allows you to interact with the computer in a certain language. It understands the expressions that you type in through that language, and performs the corresponding actions in some way, usually using an underlying language.

In Project 4, you will use Python to implement an interpreter for Scheme. The Python interpreter that you've been using all semester is written (mostly) in the C programming language. The computer itself uses hardware to interpret machine code (a series of ones and zeros that represent basic operations like adding numbers, loading information from memory, etc).

When we talk about an interpreter, there are two languages at work:

1. **The language being interpreted/implemented.** In this lab, you will implement the PyCombinator language.
2. **The underlying implementation language.** In this lab, you will use Python to implement the PyCombinator language.

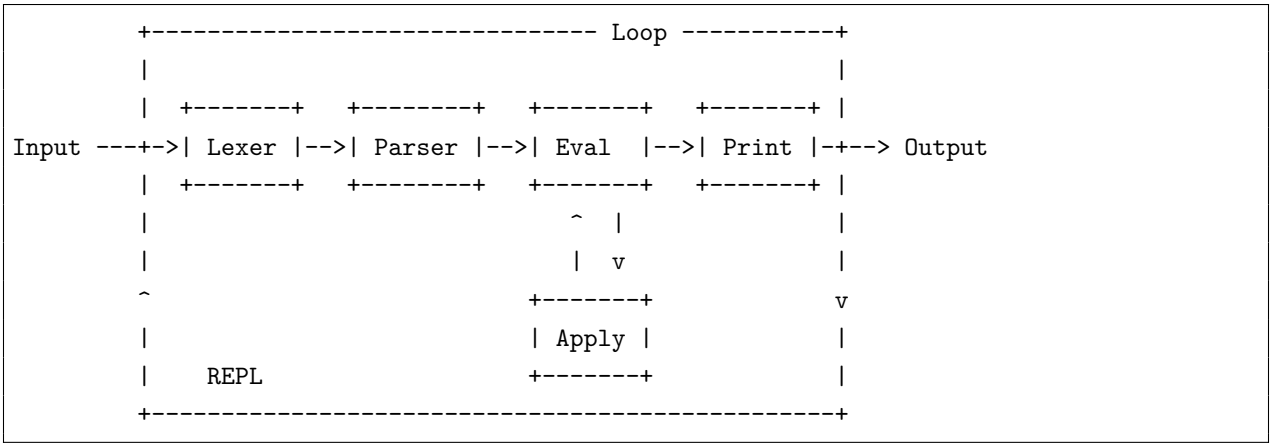
Note that the underlying language need not be different from the implemented language. In fact, in this lab we are going to implement a smaller version of Python (PyCombinator) using Python! This idea is called Metacircular Evaluation.

Many interpreters use a Read-Eval-Print Loop (REPL). This loop waits for user input, and then processes it in three steps:

- **Read:** The interpreter takes the user input (a string) and passes it through a lexer and parser.
  - The *lexer* turns the user input string into atomic pieces (tokens) that are like "words" of the implemented language.
  - The *parser* takes the tokens and organizes them into data structures that the underlying language can understand.
- **Eval:** Mutual recursion between eval and apply evaluate the expression to obtain a value.
  - *Eval* takes an expression and evaluates it according to the rules of the language. Evaluating a call expression involves calling `apply` to apply an evaluated operator to its evaluated operands.
  - *Apply* takes an evaluated operator, i.e., a function, and applies it to the call expression's arguments. Apply may call `eval` to do more work in the body of the function, so `eval` and `apply` are *mutually recursive*.

- **Print:** Display the result of evaluating the user input.

Here's how all the pieces fit together:

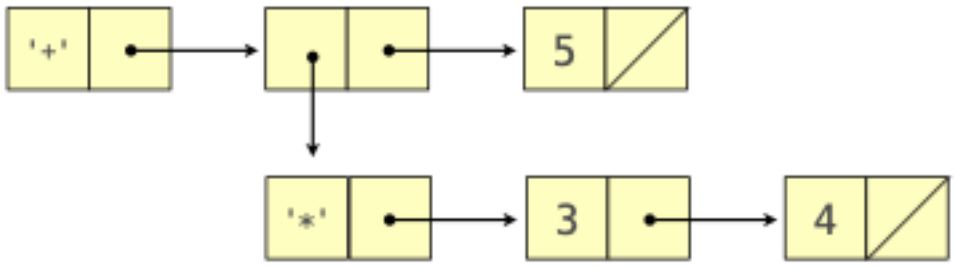


Scheme Call Expressions

A Scheme call expression is a Scheme list that is represented using a `Pair` instance in Python.

For example, the call expression `(+ (* 3 4) 5)` is represented as:

```
Pair('+', Pair(Pair('*', Pair(3, Pair(4, nil))), Pair(5, nil)))
```



(+ (* 3 4) 5)

The `Pair` class and `nil` object are defined in pair.py of the Scheme project.

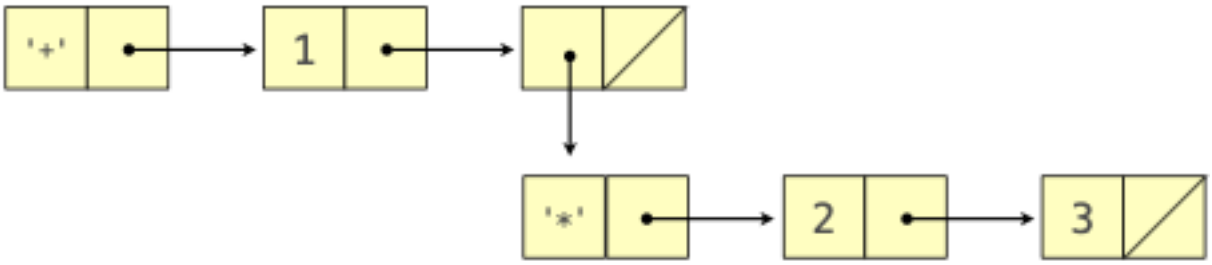
```
class Pair:
    "A Scheme list is a Pair in which rest is a Pair or nil."
    def __init__(self, first, rest):
        self.first = first
        self.rest = rest

    ... # There are also __str__, __repr__, and map methods, omitted here.
```

Q3: Representing Expressions

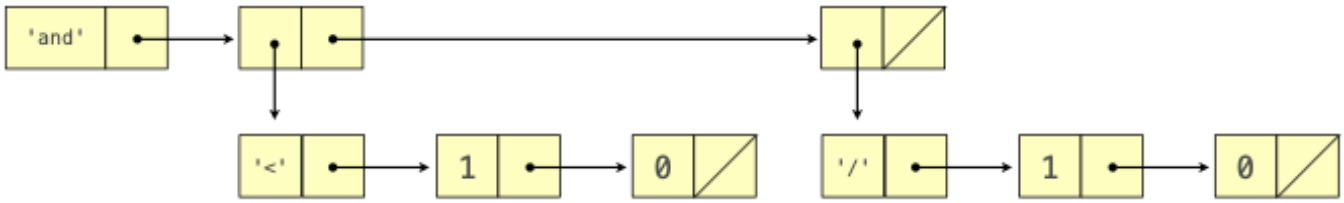
Write the Scheme expression in Scheme syntax represented by each `Pair` below. Try drawing the linked list diagram too. The first one is done for you.

```
Pair('+', Pair(Pair('*', Pair(3, Pair(4, nil))), Pair(5, nil)))
```

```
>>> Pair('+', Pair(1, Pair(Pair('*', Pair(2, Pair(3, nil))), nil)))
```




```
>>> Pair('and', Pair(Pair('<', Pair(1, Pair(0, nil))), Pair(Pair('/', Pair(1, Pair(0, nil
    ))), nil)))
```



Evaluation

To evaluate the expression `(+ (* 3 4) 5)` using the Project 4 interpreter, `scheme_eval` is called on the following expressions (in this order):

1. `(+ (* 3 4) 5)`
2. `+`
3. `(* 3 4)`
4. `*`
5. `3`
6. `4`
7. `5`

The `*` is evaluated because it is the operator sub-expression of `(* 3 4)`, which is an operand sub-expression of `(+ (* 3 4) 5)`.

By default, `*` evaluates to a procedure that multiplies its arguments together. But `*` could be redefined at any time, and so the symbol `*` must be evaluated each time it is used in order to look up its current value.

```
scm> (* 2 3)  ; Now it multiplies
6
scm> (define * +)
*
scm> (* 2 3)  ; Now it adds
5
```

Q4: Evaluation

Which of the following are evaluated when `scheme_eval` is called on `(if (< x 0) (- x) (if (= x -2) 100 y))` in an environment in which `x` is bound to -2? (Assume `<`, `-`, and `=` have their default values.)

- `if`
- `<`
- `=`
- `x`
- `y`
- `0`
- `-2`
- `100`
- `-`
- `(`
- `)`

Q5: Print Evaluated Expressions

Define `print_evals`, which takes a Scheme expression `expr` that contains only numbers, `+`, `*`, and parentheses. It prints all of the expressions that are evaluated during the evaluation of `expr`. They are printed in the order that they are passed to `scheme_eval`.

Note: Calling `print` on a `Pair` instance will print the Scheme expression it represents.

```
>>> print(Pair('+', Pair(Pair('*', Pair(3, Pair(4, nil))), Pair(5, nil))))
(+ (* 3 4) 5)
```

```
    def print_evals(expr):
        """Print the expressions that are evaluated while evaluating expr.

        expr: a Scheme expression containing only (, ), +, *, and numbers.

        >>> nested_expr = Pair('+', Pair(Pair('*', Pair(3, Pair(4, nil))), Pair(5, nil)))
        >>> print_evals(nested_expr)
        (+ (* 3 4) 5)
        +
        (* 3 4)
        *
        3
        4
        5
        >>> print_evals(Pair('*', Pair(6, Pair(7, Pair(nested_expr, Pair(8, nil))))))
        (* 6 7 (+ (* 3 4) 5) 8)
        *
        6
        7
        (+ (* 3 4) 5)
        +
        (* 3 4)
        *
        3
        4
        5
        8
        """
        if not isinstance(expr, Pair):
            "*** YOUR CODE HERE ***"


        else:
            "*** YOUR CODE HERE ***"
```

If `expr` is not a pair, then it is a number or `'+'` or `'*'`. In all of these cases, the `expr` should be printed to indicate that it would be evaluated.

If `expr` is a pair, then it is a call expression. Print it. Then, the operator and operands are evaluated. These are the elements in the list `expr`. So, iterate through `expr` (using either a `while` statement or `expr.map(...)`) and call `print_evals` on each element.

Submit Attendance

You're done! Excellent work this week. Please be sure to ask your section TA for the attendance form link and fill it out for credit. (one submission per person per section).