CS 61A
Summer 2023

Interpreters

Discussion 9: July 27, 2023

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:

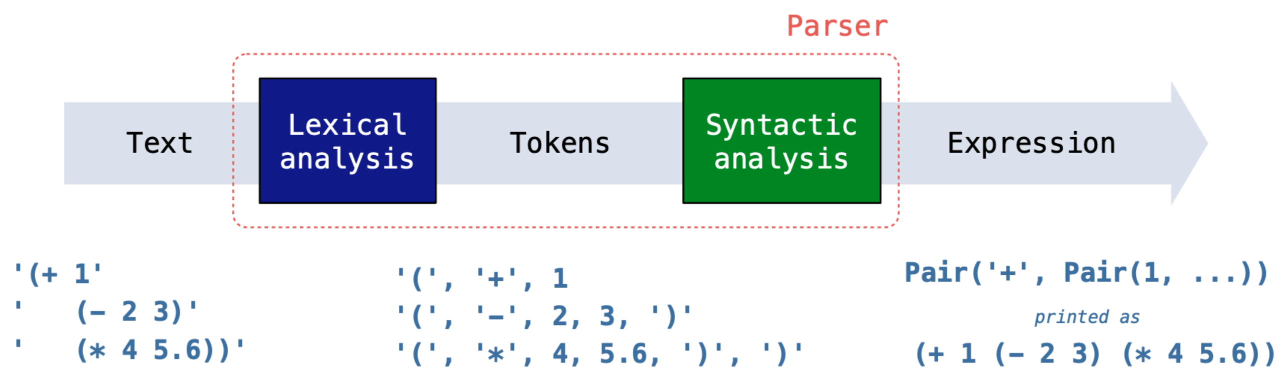
1. **The language being interpreted/implemented.** For Project 4, we are interpreting the Scheme language.
2. **The underlying implementation language.** For Project 4, we will implement an interpreter for Scheme using Python.

REPL

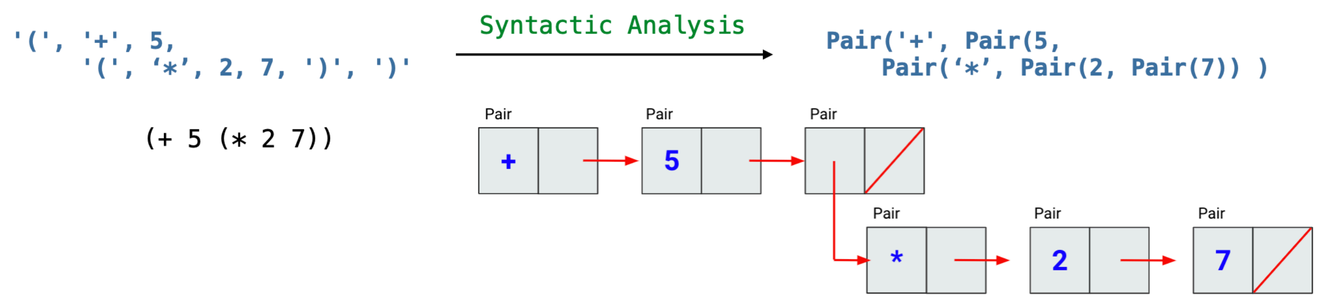
Many interpreters use a Read-Eval-Print Loop (REPL). This loop waits for user input, and then processes it in three steps:

- **Read:** The interpreter takes the user input (a string) and passes it through a parser. The parser processes the input in two steps:
  - The *lexical analysis* step turns the user input string into tokens that are like "words" of the implemented language. Tokens represent the **smallest** units of information.
  - The *syntactic analysis* step takes the tokens from the previous step and organizes them into a data structure that the underlying language can understand. For our Scheme interpreter, we create a `Pair` object (similar to a Linked List) from the tokens to represent the original call expression.
    - The first item in the `Pair` represents the operator of the call expression. The subsequent items are the operands of the call expression, or the arguments that the operator will be applied to. Note that operands themselves can also be nested call expressions.

Below is a summary of the read process for a Scheme expression input:

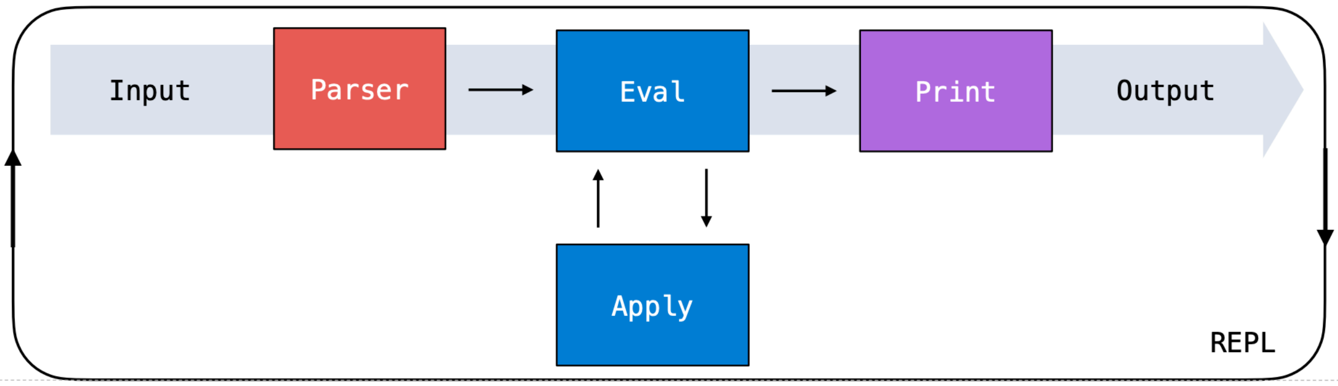


And here is a summary of how syntactic analysis converts input tokens into `Pair` objects:



- **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:



Q1: From Pair to Expression

Write out the Scheme expression with proper syntax that corresponds to the following `Pair` constructor calls.

```
>>> Pair('+', Pair(1, Pair(2, Pair(3, Pair(4, nil)))))
```

(+ 1 2 3 4)

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

(+ 1 (* 2 3))

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

(and (< 1 0) (/ 1 0))

Pair(___ , Pair(___ , Pair(___ , nil)))

Note: This worksheet is a problem bank—most TAs will not cover all the problems in discussion section.

Scheme Eval/Apply

Scheme evaluates an expression by obeying the following evaluation rules:

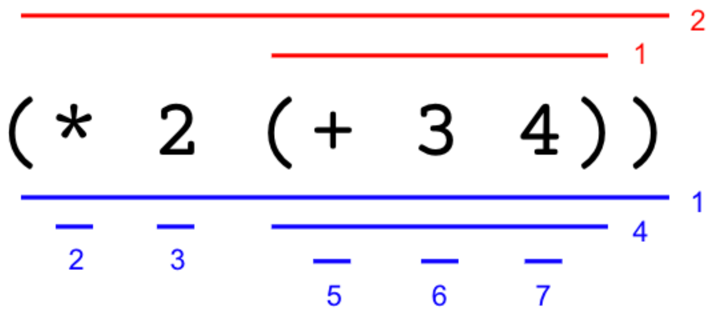
- Call `scheme_eval` on the input expression
  - If the input expression is self-evaluating (i.e. number, boolean), output that value
  - Else, the input is a call expression (Scheme list)
- If the input is a call expression:
  - Call `scheme_eval` on the operator (determine what operation is being executed)
  - Call `scheme_eval` recursively on each operand (arguments to the call expression)
  - Call `scheme_apply` to the evaluated operator on the list of evaluated operands

Take the following Scheme expression as an example: `(* 2 (+ 3 4))`

To evaluate this entire expression, we...

- Call `scheme_eval` on `(* 2 (+ 3 4))`
  - Call `scheme_eval` on `*`
  - Call `scheme_eval` on `2`
  - Call `scheme_eval` on `(+ 3 4)`
    - ∗ Call `scheme_eval` on `+`
    - ∗ Call `scheme_eval` on `3`
    - ∗ Call `scheme_eval` on `4`
    - ∗ Call `scheme_apply` to apply `+` to `(3 4)` –> `(+ 3 4)` evaluates to `7`
  - Call `scheme_apply` to apply `*` to `(2 7)` –> `(* 2 (+ 3 4))` evaluates to `14`

In total, there are 7 calls to `scheme_eval` and 2 calls to `scheme_apply` to get our final result of `14`. A visualization of these specific calls are shown below, where each underline is a call to `scheme_eval` and each overline is a call to `scheme_apply`:



Q2: Counting Eval and Apply

How many calls to `scheme_eval` and `scheme_apply` would it take to evaluate each of the following expressions?

	eval	apply
`scm> (+ 1 2)`	4	1

For this particular prompt please list out the inputs to `scheme_eval` and `scheme_apply`.

`scm> (+ 2 4 6 8)`	6	1
`scm> (+ 2 (* 4 (- 6 8)))`	10	3
`scm> (and 1 (+ 1 0) 0)`	7	1
`scm> (and (+ 1 0) (< 1 0) (/ 1 0))`		

Tail Recursion

When writing a recursive procedure, it's possible to write it in a **tail recursive** way, where all of the recursive calls are tail calls. A **tail call** occurs when a function calls another function as the last action of the current frame.

Consider this implementation of `factorial` that is *not* tail recursive:

```
(define (factorial n)
  (if (= n 0)
      1
      (* n (factorial (- n 1)))))
```

The recursive call occurs in the last line, but it is not the last expression evaluated. After calling `(factorial (- n 1))`, the function still needs to multiply that result with `n`. The final expression that is evaluated is a call to the multiplication function, not `factorial` itself. Therefore, the recursive call is `not` a tail call.

Here's a visualization of the recursive process for computing `(factorial 6)` :

```
(factorial 6)
(* 6 (factorial 5))
(* 6 (* 5 (factorial 4)))
(* 6 (* 5 (* 4 (factorial 3))))
(* 6 (* 5 (* 4 (* 3 (factorial 2)))))
(* 6 (* 5 (* 4 (* 3 (* 2 (factorial 1))))))
(* 6 (* 5 (* 4 (* 3 (* 2 1)))))
(* 6 (* 5 (* 4 (* 3 2))))
(* 6 (* 5 (* 4 6)))
(* 6 (* 5 24))
(* 6 120)
720
```

The interpreter first must reach the base case and only then can it begin to calculate the products in each of the earlier frames.

We can rewrite this function using a helper function that remembers the temporary product that we have calculated so far in each recursive step.

```
(define (factorial n)
  (define (fact-tail n result)
    (if (= n 0)
        result
        (fact-tail (- n 1) (* n result))))
  (fact-tail n 1))
```

`fact-tail` makes a single recursive call to `fact-tail`, and that recursive call is the last expression to be evaluated, so it is a tail call. Therefore, `fact-tail` is a tail recursive process.

Here's a visualization of the tail recursive process for computing `(factorial 6)`:

```
(factorial 6)
(fact-tail 6 1)
(fact-tail 5 6)
(fact-tail 4 30)
(fact-tail 3 120)
(fact-tail 2 360)
(fact-tail 1 720)
(fact-tail 0 720)
720
```

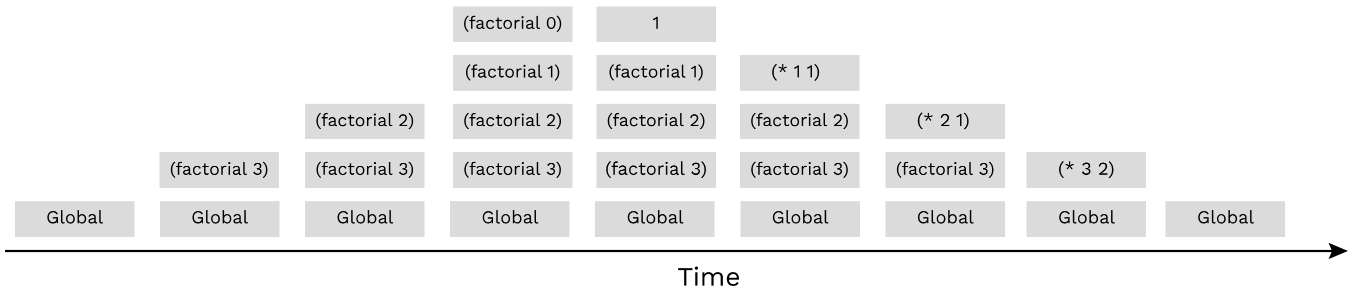
The interpreter needed less steps to come up with the result, and it didn't need to re-visit the earlier frames to come up with the final product.

In this example, we've utilized a common strategy in implementing tail-recursive procedures which is to pass the result that we're building (e.g. a list, count, sum, product, etc.) as a argument to our procedure that gets changed across recursive calls. By doing this, we do not have to do any computation to build up the result after the recursive call in the current frame, instead any computation is done *before* the recursive call and the result is passed to the next frame to be modified further. Often, we do not have a parameter in our procedure that can store this result, but in these cases we can define a helper procedure with an extra parameter(s) and recurse on the helper. This is what we did in the `factorial` procedure above, with `fact-tail` having the extra parameter `result`.

Tail Call Optimization

When a recursive procedure is not written in a tail recursive way, the interpreter must have enough memory to store all of the previous recursive calls.

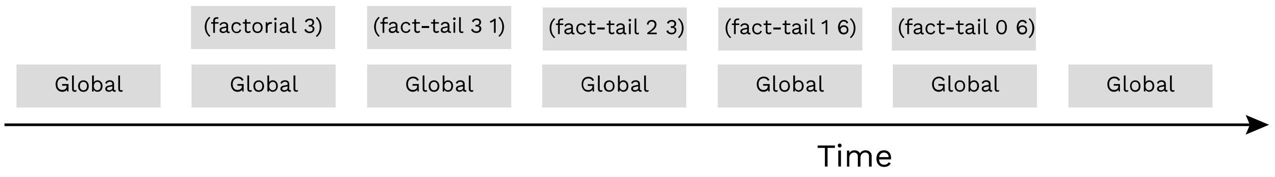
For example, a call to the `(factorial 3)` in the non tail-recursive version must keep the frames for all the numbers from 3 down to the base case, until it's finally able to calculate the intermediate products and forget those frames:



For non tail-recursive procedures, the number of active frames grows proportionally to the number of recursive calls. That may be fine for small inputs, but imagine calling `factorial` on a large number like 10000. The interpreter would need enough memory for all 1000 calls!

Fortunately, proper Scheme interpreters implement **tail-call optimization** as a requirement of the language specification. TCO ensures that tail recursive procedures can execute with a constant number of active frames, so programmers can call them on large inputs without fear of exceeding the available memory.

When the tail recursive `factorial` is run in an interpreter with tail-call optimization, the interpreter knows that it does not need to keep the previous frames around, so it never needs to store the whole stack of frames in memory:



Tail-call optimization can be implemented in a few ways:

1. Instead of creating a new frame, the interpreter can just update the values of the relevant variables in the current frame (like `n` and `result` for the `fact-tail` procedure). It reuses the same frame for the entire calculation, constantly changing the bindings to match the next set of parameters.
2. How our 61A Scheme interpreter works: The interpreter builds a new frame as usual, but then *replaces* the current frame with the new one. The old frame is still around, but the interpreter no longer has any way to get to it. When that happens, the Python interpreter does something clever: it *recycles* the old frame so that the next time a new frame is needed, the system simply allocates it out of recycled space. The technical term is that the old frame becomes "garbage", which the system "garbage collects" behind the programmer's back.

Tail Context

When trying to identify whether a given function call within the body of a function is a tail call, we look for whether the call expression is in **tail context**.

Given that each of the following expressions is the last expression in the body of the function, the following expressions are tail contexts:

1. the second or third operand in an `if` expression
2. any of the non-predicate sub-expressions in a `cond` expression (i.e. the second expression of each clause)
3. the last operand in an `and` or an `or` expression
4. the last operand in a `begin` expression's body
5. the last operand in a `let` expression's body

For example, in the expression `(begin (+ 2 3) (- 2 3) (* 2 3))`, `(* 2 3)` is a tail call because it is the last operand expression to be evaluated.

Tail Calls

Q3: Is Tail Call

For each of the following procedures, identify whether it contains a recursive call in a tail context. Also indicate if it uses a constant number of active frames. → tail recursive

= every recursive call is a tail call

```
(define (question-a x)
  (if (= x 0) 0
      (+ x (question-a (- x 1)))))
```

```
(define (question-b x y)
  (if (= x 0) y
      (question-b (- x 1) (+ y x))))
```

```
(define (question-c x y)
  (if (> x y)
      (question-c (- y 1) x)
      (question-c (+ x 10) y)))
```

```
(define (question-d n)
  (if (question-d n) ←
      (question-d (- n 1))
      (question-d (+ n 10))))
```

```
(define (question-e n)
  (cond ((<= n 1) 1)
        ((question-e (- n 1)) (question-e (- n 2)))
        (else (begin (print 2) (question-e (- n 3))))))
```

Q4: Sum

Write a tail recursive function that takes in a Scheme list and returns the numerical sum of all values in the list. You can assume that the list contains only numbers (no nested lists).

```
scm> (sum '(1 2 3))
6
scm> (sum '(10 -3 4))
11
```

```
(define (sum lst)
  'YOUR-CODE-HERE
   (define (helper lst total)
      (if (null? lst)
           total
           (helper (cdr lst)
                   (+ total (car lst))))
   )
   (helper lst 0 )

                               (= 0 (length lst))
                           ↑
        (cond
          ((null? lst) total)
          (else (helper (cdr lst) (+ total (car lst)))))
)

(expect (sum '(1 2 3)) 6)
(expect (sum '(10 -3 4)) 11)

# You can use more space on the back if you want
```

Q5: Reverse

Write a tail-recursive function `reverse` that takes in a Scheme list a returns a reversed copy. *Hint*: use a helper function!

```
scm> (reverse '(1 2 3))
(3 2 1)
scm> (reverse '(0 9 1 2))
(2 1 9 0)
```

```
(define (reverse lst)                          (append lst1 lst2) -> lst1 + lst2
  'YOUR-CODE-HERE
   (define (helper lst rev)
       (if (null? lst)                        (append rev (list (car lst)))
           rev
           (helper (cdr lst)                  (cons (car lst) rev)  )

    )
    (helper lst nil)                          (r '(1 2 3) nil)
)                                                 ↓
                                              (r '(2 3) '(1))
(expect (reverse '(1 2 3)) (3 2 1))               ↓
(expect (reverse '(0 9 1 2)) (2 1 9 0))
                                              (r '(3) '(2 1))
# You can use more space on the back if you want
```

Note: This worksheet is a problem bank—most TAs will not cover all the problems in discussion section.

Scheme Data Abstractions

Data abstraction is a powerful concept in computer science that allows programmers to treat code as objects. For example, using code to represent cars, chairs, people, and so on. That way, programmers don't have to worry about *how* code is implemented; they just have to know *what* it does.

Data abstraction mimics how we think about the world. If you want to drive a car, you don't need to know how the engine was built or what kind of material the tires are made of to do so. You just have to know how to use the car for driving itself, such as how to turn the wheel or press the gas pedal.

A data abstraction consists of two types of functions:

- **Constructors**: functions that build the abstract data type.
- **Selectors**: functions that retrieve information from the data type.

Programmers design data abstractions to abstract away how information is stored and calculated such that the end user does *not* need to know how constructors and selectors are implemented. The nature of *abstraction* allows whoever uses them to assume that the functions have been written correctly and work as described. Using this idea, developers are able to use a variety of powerful libraries for tasks such as data processing, security, visualization, and more without needing to write the code themselves!

In Python, you primarily worked with data abstractions using Object Oriented Programming, which used Python `Object`s to store the data. Notably, this is not possible in Scheme, which is a functional programming language. Instead, we create and return new structures which represent the current state of the data.

Cities

Say we have an abstract data type for cities. A city has a name, a latitude coordinate, and a longitude coordinate.

Our data abstraction has one **constructor**: * `(make-city name lat lon)`: Creates a city object with the given name, latitude, and longitude.

We also have the following **selectors** in order to get the information for each city:

- `(get-name city)`: Returns the city's name
- `(get-lat city)`: Returns the city's latitude
- `(get-lon city)`: Returns the city's longitude

Here is how we would use the constructor and selectors to create cities and extract their information:

```
scm> (define berkeley (make-city 'Berkeley 122 37))
berkeley
scm> (get-name berkeley)
Berkeley
scm> (get-lat berkeley)
122
scm> (define new-york (make-city 'NYC 74 40))
new-york
scm> (get-lon new-york)
40
```

The point of data abstraction is that we do not need to know how an abstract data type is implemented, but rather just how we can interact with and use the data type.

Q6: Distance

We will now implement the function `distance`, which computes the *Euclidean distance* between two city objects; the Euclidean distance between two coordinate pairs `(x1, y1)` and `(x2, y2)` can be found by calculating the `sqrt((x1 - x2) ** 2 + (y1 - y2) ** 2)`. Use the latitude and longitude of a city as its coordinates; you'll need to use the selectors to access this info!

> You may find the following methods useful: - `(expt base exp)`: calculate `base ** exp` - `(sqrt x)` calculate `sqrt(x)`

```
(define (distance city-a city-b)
    'YOUR-CODE-HERE









)

# You can use more space on the back if you want
```

Q7: Closer City

Next, implement `closer-city`, a function that takes a latitude, longitude, and two cities, and returns the name of the city that is relatively closer to the provided latitude and longitude.

You may only use the selectors and constructors introduced above and the `distance` function you just defined for this question.

> **Hint**: How can you use your `distance` function to find the distance between the given location and each of the given cities?

```
(define (closer-city lat lon city-a city-b)
    'YOUR-CODE-HERE








)

# You can use more space on the back if you want
```

Trees

Here, we have a data abstraction for trees! Recall that a tree instance consists of a label and a list of other tree instances.

Our data abstraction has one **constructor**: * `(tree label branches)`: Creates a tree object with the given label and list of branches. `branches` must be a scheme list of other tree objects, or `nil` if there are no branches.

We also have the following **selectors** in order to get information about a tree:

- `(label t)`: Returns the label of tree `t`
- `(branches t)`: Returns the branchees of tree `t`

Here is an example of using this tree data abstraction:

```
scm> (define t (tree 5 (list (tree 4 nil) (tree 7 nil))))
t
scm> (label t)
5
scm> (label (car (branches t)))
4
scm> (label (car (cdr (branches t))))
7
```

Note that we are unaware of how this tree data abstraction is really implemented, but we are still able to use it through the given constructor and selectors.

Q8: Is Leaf

Implement the `is-leaf?` procedure for the tree data abstraction. `is-leaf?` takes in a tree `t` and returns `#t` if `t` is a leaf and `#f` otherwise.

Recall that a leaf is a tree without any branches.

```
(define (is-leaf? t)
    'YOUR-CODE-HERE



)

# You can use more space on the back if you want
```

Q9: Sum Nodes

Now, consider trees with integer labels. We are interested in finding the sum of all the labels in a tree.

To accomplish this, it would be useful to have the following helper procedure. Implement `sum-list`, which takes in a list of integers `lst` and returns their sum.

```
(define (sum-list lst)
    'YOUR-CODE-HERE









)


# You can use more space on the back if you want
```

Now, implement the procedure `sum-nodes`, which takes in a tree `t` and returns the sum of all of its labels. You may assume `t` only consists of integer labels and that `sum-list` works as designed.

> *Note*: The built-in procedure `map` takes in a one-arg procedure and a list, and it returns a list constructed by calling the procedure on each item in the list.

```
(define (sum-nodes t)
    (define branch-sums (map sum-nodes (branches t)))
    'YOUR-CODE-HERE



)


# You can use more space on the back if you want
```

Q10: Fun Tree

Implement `fun-tree`, which takes in a one-argument procedure `fun` and a tree `t`. It returns a new tree with the same shape as `t`, but each label is the result of applying `fun` to the corresponding label in `t`.

> *Hint*: You may find the `map` procedure useful.

```
(define (fun-tree fun t)
    (define new-label ______________________)
    (define new-branches ______________________)
    (tree new-label new-branches)
)
```