The Invention of Scheme, 1762 (colorized)

Hi! Welcome to 61A Discussion :)

We will begin at **8:10**!

Attendance form and skeleton notes:

cs61a.bencuan.me

Secret word:

Announcements

- HW6 due tonight
- Scheme Part 1 due next Tues.
  - Start early!

Agenda

- Attendance
- Tail Recursion
-

Tail Recursion

The Vocab

- **Tail Context:** the **very last** thing that's done in a Scheme expression
- **Tail Call:** calling a function in tail context
- **Tail Recursion:** recursive tail call (in tail context)

Why do we care about tail recursion?

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

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

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

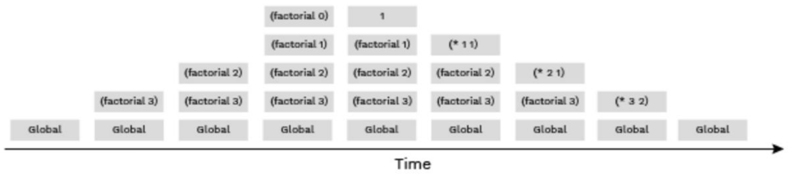
```
(factorial 6)
(* 6 (factorial 5))
(* 6 (* 5 (factorial 4)))
(* 6 (* 5 (* 4 (factorial 3))))
(* 6 (* 5 (* 4 (* 3 (factorial 2)))))
(* 6 (* 5 (* 4 (* 3 (* 2 (factorial 1))))))
(* 6 (* 5 (* 4 (* 3 (* 2 1)))))
(* 6 (* 5 (* 4 (* 3 2))))
(* 6 (* 5 (* 4 6)))
(* 6 (* 5 24))
(* 6 120)
720
```

Factorial, tail recursive version

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

Scheme project EC ideas

- Instead of creating a new frame, replace the old frame (since you don't need it anymore)



Q1: 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.

```
(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))))
```

- **Strategy:**

1. What is the last operation made?
2. Is it a recursive call?
3. Is it normal recursion or tree recursion?

Q1: 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.

```
(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))))))
```

- **Strategy:**

1. What is the last operation made?
2. Is it a recursive call?
3. Is it normal recursion or tree recursion?

Writing tail recursive functions

1. **Create a helper function!**
   a. Parameters: so-far, anything else that changes
2. **Base case:** return so-far
3. **Recursive case:** must be in tail context
   a. Change so-far
   b. Very first operator on the line should be function name
4. **Remember to call the helper function!**

Q2: 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
```

Q3: Reverse

Q3: 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)
```

Interpreters

The Calculator Example

- **Goal**: let’s write an interpreter that understands simple math expressions!
  - (+ 2 2)
  - (- 5)
  - (* (+ 1 2) (+ 2 3))
- Understands +, -, *, /, and nested expressions

Pairs

- Pairs are literally linked lists!! Only differences:
  - **Pair** vs Link
  - **nil** vs Link.empty
  - **Pair(1, nil)** vs **Link(1)**
- Used to represent Scheme code in Python

Operators and Operands

- An **operator** is the function you are trying to apply in Scheme
  - +, list, append...
  - The **very first** element in a Pair list
- An **operand** is a parameter that is passed into the function
  - In (+ 3 5), + is the operator and 3, 5 are both operands

Q4: Using Pair

Answer the following questions about a `Pair` instance representing the Calculator expression `(+ (- 2 4) 6 8)`.

Write out the Python expression that returns a `Pair` representing the given expression:

What is the operator of the call expression?

If the `Pair` you constructed in the previous part was bound to the name `p`, how would you retrieve the operator?

What are the operands of the call expression?

If the `Pair` you constructed was bound to the name `p`, how would you retrieve a list containing all of the operands?

How would you retrieve only the first operand?

Hint: Pairs have .first and .rest just like a linked list! **do not** use car, cdr, cons...

Q4 Solutions

Write out the Python expression that returns a `Pair` representing the given expression:

Draw a box and pointer diagram corresponding to the Pair:

Q4 Solution 2

What is the operator of the call expression?

If the `Pair` you constructed in the previous part was bound to the name `p`, how would you retrieve the operator?

What are the operands of the call expression?

If the `Pair` you constructed was bound to the name `p`, how would you retrieve a list containing all of the operands?

How would you retrieve only the first operand?

Q9: From Pair to Calculator

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

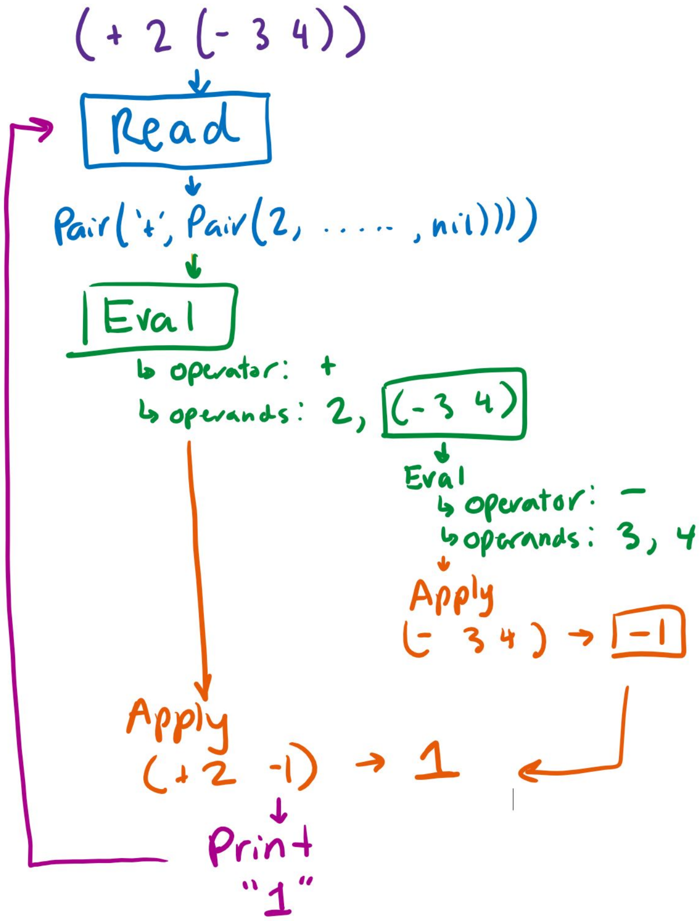
```
>>> Pair('+', Pair(1, Pair(2, Pair(3, Pair(4, nil)))))
```

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

Eval and Apply

- **Eval rules:**
  - Basic elements (numbers, booleans...) - done (base case)
  - Built-in functions (+, -, ...) - lookup in OPERATORS
  - Function:
    - Call eval on operator
    - Recursively call eval on all operands
    - Call apply on operator and operands

Example



(+ 2 (- 3 4))
Read
Pair('+', Pair(2, ... , nil)))
Eval
operator: +
operands: 2, (- 3 4)
Eval
operator: -
operands: 3, 4
Apply
(- 3 4) → -1
Apply
(+ 2 -1) → 1
Print
"1"

Counting eval and apply

- **Entire expression = +1 eval**
- **Operator = +1 eval, +1 apply**
- **Operand = +1 eval**
- **Remember to recursively run eval on nested expressions!**
- Example: (+ 2 (- 3 4)) → 7 eval, 2 apply
- **Do not count special forms**
  - Example: (and 1 2 3) → 2 eval, 0 apply

Q8: Counting Eval and Apply

How many calls to `calc_eval` and `calc_apply` would it take to evaluate each of the following Calculator expressions?

```
scm> (+ 1 2)
```

```
scm> (+ 2 4 6 8)
```

```
scm> (+ 2 (* 4 (- 6 8)))
```

```
scm> (and 1 (+ 1 0) 0)
```

- **Entire expression = +1 eval**
- **Operator = +1 eval, +1 apply**
- **Operand = +1 eval**
- **Remember to recursively run eval on nested expressions!**
  - Example: (+ 2 (- 3 4)) → 7 eval, 2 apply
- **Do not count special forms**
  - Example: (and 1 2 3) → 2 eval, 0 apply

Q8: Counting Eval and Apply

How many calls to `calc_eval` and `calc_apply` would it take to evaluate each of the following Calculator expressions?

```
scm> (+ 1 2)
```

```
scm> (+ 2 4 6 8)
```

Q8: Counting Eval and Apply

How many calls to `calc_eval` and `calc_apply` would it take to evaluate each of the following Calculator expressions?

```
scm> (+ 2 (* 4 (- 6 8)))
```

```
scm> (and 1 (+ 1 0) 0)
```

New Procedure! (Q5)

Suppose we want to add the `//` operation to our Calculator interpreter. Recall from Python that `//` is the floor division operation, so we are looking to add a built-in procedure `//` in our interpreter such that `(// dividend divisor)` returns dividend // divisor. Similarly we handle multiple inputs as illustrated in the following example `(// dividend divisor1 divisor2 divisor3)` evaluates to (((dividend // divisor1) // divisor2) // divisor3). For this problem you can assume you are always given at least 1 divisor. Also for this question do you need to call `calc_eval` inside `floor_div`? Why or why not?

New Procedure! (Q5)

```python
def calc_eval(exp):
    if isinstance(exp, Pair): # Call expressions
        return calc_apply(calc_eval(exp.first), exp.rest.map(calc_eval))
    elif exp in OPERATORS:      # Names
        return OPERATORS[exp]
    else:                       # Numbers
        return exp

def floor_div(expr):
    """
    >>> calc_eval(Pair("//", Pair(10, Pair(10, nil))))
    1
    >>> calc_eval(Pair("//", Pair(20, Pair(2, Pair(5, nil)))))
    2
    >>> calc_eval(Pair("//", Pair(6, Pair(2, nil))))
    3
    """
    "*** YOUR CODE HERE ***"

OPERATORS = { "//": floor_div }
```

New Form! (Q6)

- Can we add <=, >=, etc. without changing calc_eval?
- What about and, or?

```
def calc_eval(exp):
    """Evaluates a Calculator expression represented as a Pair."""
    if isinstance(exp, Pair): # Call expressions
        fn = calc_eval(exp.first)
        args = list(exp.rest.map(calc_eval))
        return calc_apply(fn, args)
    elif exp in OPERATORS:    # Names
        return OPERATORS[exp]
    else:                     # Numbers
        return exp
```

New Form! (Q6)

iii. Now, complete the implementation below to handle `and` expressions. You may assume the conditional operators (e.g. `<`, `>`, `=`, etc) have already been implemented for you.

```
def calc_eval(exp):
    if isinstance(exp, Pair):

        if _______________________: # and expressions
            return eval_and(exp.rest)
        else:                     # Call expressions
            return calc_apply(calc_eval(exp.first), list(exp.rest.map(calc_eval)))
    elif exp in OPERATORS:        # Names
        return OPERATORS[exp]
    else:                         # Numbers
        return exp

def eval_and(operands):
    "*** YOUR CODE HERE ***"
```

Define (Q7)

In the last few questions we went through a lot of effort to add operations so we can do most arithmetic operations easily. However it's a real shame we can't store these values. So for this question let's implement a `define` special form that saves values to variable names. This should work like variable assignment in Scheme; this means that you should expect inputs of the form `(define <variable_name> <value>)` and these inputs should return the symbol corresponding to the variable name.

```
calc> (define a 1)
a
calc> a
1
```

This is a more involved change. Here are the 4 steps involved:

1. Add a `bindings` dictionary that will store the names and correspondings values of variables as key-value pairs of the dictionary.
2. Identify when the define form is given to `calc_eval`.
3. Allow variables to be looked up in `calc_eval`.
4. Write the function `eval_define` which should actually handle adding names and values to the bindings dictionary.