**Dynamic Scope**

The way in which names are looked up in Scheme and Python is called lexical scope (or static scope). You can see what names are in scope by inspecting the definitions.

**Lexical scope**: The parent of a frame is the environment in which a procedure was defined.

**Dynamic scope**: The parent of a frame is the environment in which a procedure was called.

(define f (lambda (x) (+ x y)))
(define g (lambda (x y) (f (+ x x))))
(g 3 7)

**Lexical scope**: The parent for f’s frame is the global frame.

**Dynamic scope**: The parent for f’s frame is g’s frame.

Error: unknown identifier: y

---

**Functional Programming**

All functions are pure functions.

No re-assignment and no mutable data types.

Name-value bindings are permanent.

Advantages of functional programming:

- The value of an expression is independent of the order in which sub-expressions are evaluated.
- Sub-expressions can safely be evaluated in parallel or only on demand (lazily).
- Referential transparency: The value of an expression does not change when we substitute one of its subexpression with the value of that subexpression.

But... no for/while statements! Can we make basic iteration efficient? Yes!

---

**Tail Recursion**

From the Revised Report on the Algorithmic Language Scheme:

"Implementations of Scheme are required to be properly tail-recursive. This allows the execution of an iterative computation in constant space, even if the iterative computation is described by a syntactically recursive procedure."

---

**Recursion and Iteration in Python**

In Python, recursive calls always create new active frames.

```python
def factorial(n, k):
    if n == 0:
        return k
    else:
        return factorial(n-1, n*k)
```

**Tail Recursion**

```python
def factorial(n, k):
    while n > 0:
        n, k = n-1, n*k
    return k
```

Time | Space
--- | ---
Θ(n) | Θ(1)

**Recommendations**

- Use resources like
- Ensure efficient use of resources
Tail Calls

A procedure call that has not yet returned is active. Some procedure calls are tail calls. A Scheme interpreter should support an unbounded number of active tail calls using only a constant amount of space.

A tail call is a call expression in a tail context:
- The last sub-expression in a lambda expression
- Sub-expressions 2 & 3 in a tail context if expression
- All non-predicate sub-expressions in a tail context cond
- The last sub-expression in a tail context and, or, begin, or let

Example: Length of a List

\[
\text{define} \ (\text{length}) \ s \ = \ \begin{cases} \ 0 & \text{null? } s \\ \ + \ (1 \ (\text{length} \ (\text{cdr} \ s))) & \text{else} \end{cases}
\]

A call expression is not a tail call if more computation is still required in the calling procedure.

Linear recursive procedures can often be re-written to use tail calls.

\[
\text{define} \ (\text{length-tail}) \ s \ = \ \begin{cases} \ (\text{length-iter} \ s \ 0) & \text{null? } s \\ \ (\text{length-tail} \ (\text{cdr} \ s)) \ + \ 1 & \text{else} \end{cases}
\]

Eval with Tail Call Optimization

The return value of the tail call is the return value of the current procedure call. Therefore, tail calls shouldn't increase the environment size.

Which Procedures are Tail Recursive?

Which of the following procedures run in constant space? 🎯

• \text{define} \ (\text{factorial}) \ n \ = \ \begin{cases} \ n & \text{if } n \leq 1 \\ \ (\text{factorial} \ (n-1)) \ \ast \ n & \text{else} \end{cases}

• \text{define} \ (\text{contains}) \ s \ v \ = \ \begin{cases} \ \text{false} & \text{null? } s \\ \ \text{if } (\text{= } v \ (\text{car} \ s)) \ \text{true} \ || \ (\text{contains} \ (\text{cdr} \ s) \ v) & \text{else} \end{cases}

• \text{define} \ (\text{has-repeat}) \ s \ = \ \begin{cases} \ \text{false} & \text{null? } s \\ \ \text{if } (\text{contains?} \ (\text{cdr} \ s) \ (\text{car} \ s)) \ \text{true} \ || \ (\text{has-repeat} \ (\text{cdr} \ s)) & \text{else} \end{cases}

Tail Recursion Examples
Map and Reduce

Example: Reduce

```scheme
(define reduce procedure start)
(reduce procedure (cdr s) (procedure start (car s))))
```

Recursive call is a tail call
Space depends on what procedure requires

- `(reduce + '(* 4 5) 2)`
  - 128
- `(reduce (lambda (x y) (cons y x)) '(3 4 5) '(2))`
  - 17

Example: Map with Only a Constant Number of Frames

```scheme
(define map procedure s)
(define map-reverse s m)
(define reverse s)
```

```scheme
(define map-reverse s m)
(define reverse-iter s r)
```

- `(map (lambda (x) (- 5 x)) (list 1 2))`