Order of evaluation, and the stack

CIS 352 — Spring 2020
The stack

- Some expressions can be evaluated in $O(1)$ time, regardless of the values involved; a subset are **atomic expressions**.
  - `(cons x (cdr lst))` is an $O(1)$ operation, but is made of several atomic steps: evaluate $x$, evaluate $lst$, evaluate `(cdr lst)`, call `cons` to build the new cons-cell, return…
  - Variable references (e.g., to $x$, $lst$, $cons$) are considered **atomic expressions** that take just one conceptual step. A lambda expression is also considered atomic.
- Other, complex, expressions, such as function invocation may generally take unbounded time.
  - The stack stores pending data while this work occurs.
The stack

- Consider \((f \ (g \ x) \ (h \ y))\)—how is this evaluated?
  - \(f\) is evaluated atomically
  - \((g \ x)\) is evaluated as the argument to this func. value
    - \(g\) is evaluated atomically; then \(x\)
    - The value of \(g\) is applied on the value of \(x\); …; returns
  - \((h \ y)\) is evaluated as a second argument
    - \(h\) is evaluated atomically; then, \(y\)
    - The value of \(h\) is applied on the value of \(y\); …; returns
  - The value of \(f\) is applied on \(g\)’s and \(h\)’s return values
The stack

• Consider \((f (g \ x) (h \ y))\)—how is this evaluated?
  
  • While the call \((g \ x)\) is being evaluated, we need to remember a few things: the value of \(f\) just evaluated, the expression \((h \ y)\) to be evaluated next; while the call \((h \ y)\) is evaluated, we need to save the value of \(f\) and \((g \ x)\).

  • These values are saved on the stack!

  • As calls&returns form a proper nesting structure, we want to store such pending values in LIFO order.

  • Implemented well, using a stack lends itself to improved cache performance as values used together, sit together.
The stack

\[(f \ (g \ x) \ (h \ y))\]

The expression is reached from a caller or surrounding expression (called the **eval. context** / call **ctxt.**)

... **ctxt** ...

empty
The stack

\[(f \ (g \ x) \ (h \ y))\]

**Control** (the current expression) steps to evaluate the subexpression in **call position** / function position.

The value of \(f\) can be evaluated atomically.
The stack

(f (g x) (h y))

Control (the current expression) steps to evaluate the first subexpression in argument position, (g x). The value for f is saved on the stack.

Its arguments must be evaluated first.
The stack

\[(f \ (g \ x) \ (h \ y))\]

Control (the current expression) steps to evaluate the subexpression in call position.

The value of \(g\) can be evaluated atomically.
The stack

\((f (g \textbf{x}) (h \ y))\)

Control steps to evaluate the subexpression in argument position. The value for \(g\) is saved on the stack.

The value of \(x\) can be evaluated atomically.
The stack

\[(f \ (g \ x) \ (h \ y))\]

The value of \(x\) just evaluated, and the value of \(g\) saved just before, can now be used to apply \(g\) on \(x\), leading to an unbounded amount of work.

This may involve any number of pending expressions being saved and eliminated atop the stack.
The stack

\((f (g x) (h y))\)

g returns with a value that must be saved on the stack while the final argument expression of \((f \ldots)\) is evaluated.
The stack

(f (g x) (h y))

Control (the current expression) steps to evaluate the second subexpression in argument position, (h y). The value for (g x) is now saved on the stack.
The stack

\((f \ (g \ x) \ (h \ y))\)

Control (the current expression) steps to evaluate the subexpression in call position.
The stack

\[(f \ (g \ x) \ (h \ y))\]

Control steps to evaluate the subexpression in argument position. The value for \(h\) is saved on the stack.

The value of \(y\) can be evaluated atomically.
The stack

\[(f \ (g \ x) \ (h \ y))\]

The value of \(y\) just evaluated, and the value of \(h\) saved just before, can now be used to apply \(h\) on \(y\), leading to an unbounded amount of work.

This may involve any number of pending expressions being saved and eliminated atop the stack.
The stack

\[(f \ (g \ x) \ (h \ y))\]

h returns with the final argument value for the original call to f.

The value of f, saved on the stack, is applied on the value of \((g \ x)\), also on the stack, and the value of \((h \ y)\) just returned.
The stack

\[(f \ (g \ x) \ (h \ y))\]

This call may involve any number of pending expressions being saved and eliminated atop the stack.

When f returns, it’s value is returned to the original evaluation context.
Example

For the following code, indicate when each subexpression is reached and returns.

\[(+ (f x) 1 (g (f y)))\]
For the following code, indicate when each subexpression is reached and returns.

\[ (+ (f \ x) \ 1 (g (f \ y))) \]

\[ (+1 (f5 \ x6)^2 1^3 (g^7 (f^9 \ y^{10})^8)^4)^0 \]

With each subexpression labeled.
Example

For the following code, indicate when each subexpression is reached and returns.

\(((\lambda(x) \ (x \ x)) \ (\lambda(x) \ (x \ x)))\)

\((+ \ (f \ x) \ 1 \ (g \ (f \ y))))\)

\((+1 \ (f^5 \ x^6)^2 \ 1^3 \ (g^7 \ (f^9 \ y^{10})^8)^4)^0\)

\([0 \ [2 \ [5 \ [6 \ \text{Call}^f]^2 \ [3]^3 \ [4 \ [7 \ [8 \ [9 \ [10 \ \text{Call}^f]^8 \ \text{Call}^g]^4 \ \text{Call}^+]^0]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]}
Example

For the following code, indicate when each subexpression is reached and returns.

\[
\text{(let ([ls '(0)])}
\text{(if (null? ls) 3 '(()) (cons ls ls))})
\]

\[
\begin{array}{llllllll}
0 & 1 & \text{Bind} & \text{ls} & 2 & 3 & 6 & 7 & \text{Call} \text{null?} & 3 \\
5 & 8 & 9 & 10 & \text{Call} \text{cons} & 5 & 2 & 0
\end{array}
\]