CSC 520, Spring 2020

Principles of Programming Languages

Michelle Strout

Version: 1

February 10, 2020

This Week's Plan

- **Last week: using algebraic laws**
  - To implement functions
  - To prove a functions cost
- **Finish up the method of accumulating parameters**
- **Course Project Proposal and Talk Expectations**
- **NOTE: association lists covered in notes and book but NOT in class**
- **Let constructs**
- **Lambda functions (Wednesday)**

List Reversal

- **Algebraic Laws for list reversal**

```
(reverse '())           == '()
(reverse (cons y ys)) == (append (reverse ys) (list1 y))
```

- **Straight-forward list reversal**

```
(define reverse (xs)
    (if (null? xs)
        '()
        (append (reverse (cdr xs)) (list1 (car xs)))))
```

- **How many cons cells allocated? Let n=|xs|**
  - Calls to append? Need to know number of calls to reverse?
  - Length of lists passed to append?
  - List1 call cost?
  - Study question: how would we prove O(N^2) cost using induction?

Method of accumulating parameters

- O(N^2) is usually too expensive

- Instead write a helper function revapp that takes two arguments that satisfies these algebraic rules

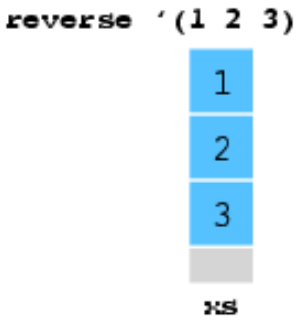
```
(revapp xs ys)          == (append (reverse xs) ys)
(reverse xs)            == (revapp xs '())
```

- ➔ Write the
  - (1) two cases and
  - (2) use algebraic rules that do NOT involve append to
  - (3) implement code for revapp

```
revapp xs     ys          == (reverse xs) .. ys
revapp e      ys          == ys
revapp(z .. zs)   ys      == (reverse zs)..z ..ys
```

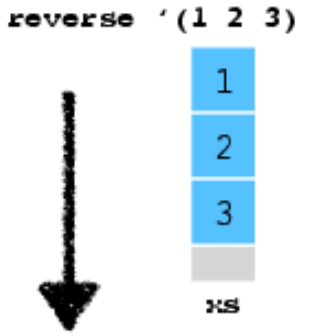
Linear reverse, graphically

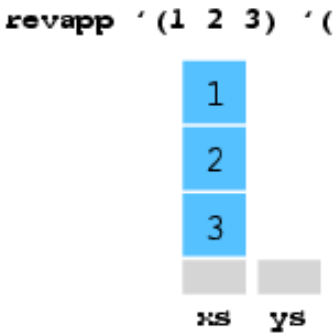

```
(define revapp (xs ys)
   (if (null? xs)
       ys
       (revapp (cdr xs)
               (cons (car xs) ys))))

(define reverse (xs) (revapp xs '()))
```



Reverse calls revapp with ys='()

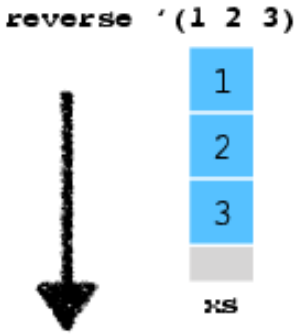

```
(define revapp (xs ys)
   (if (null? xs)
       ys
       (revapp (cdr xs)
               (cons (car xs) ys))))

(define reverse (xs) (revapp xs '()))
```

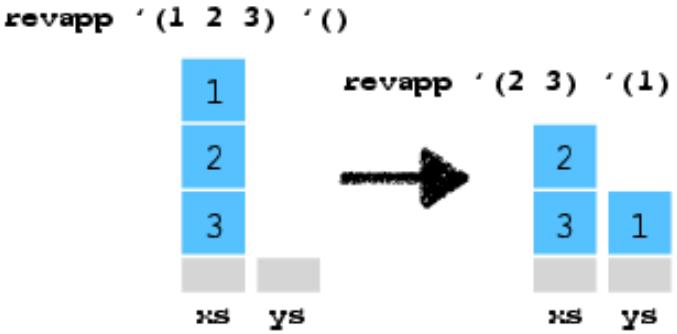




Take 1 off front of xs and put on front of ys

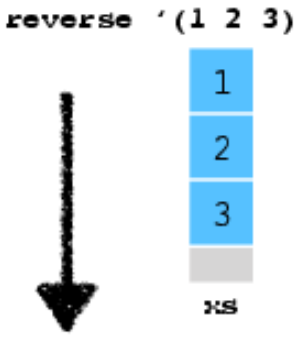

```
(define revapp (xs ys)
   (if (null? xs)
       ys
       (revapp (cdr xs)
               (cons (car xs) ys))))

(define reverse (xs) (revapp xs '()))
```

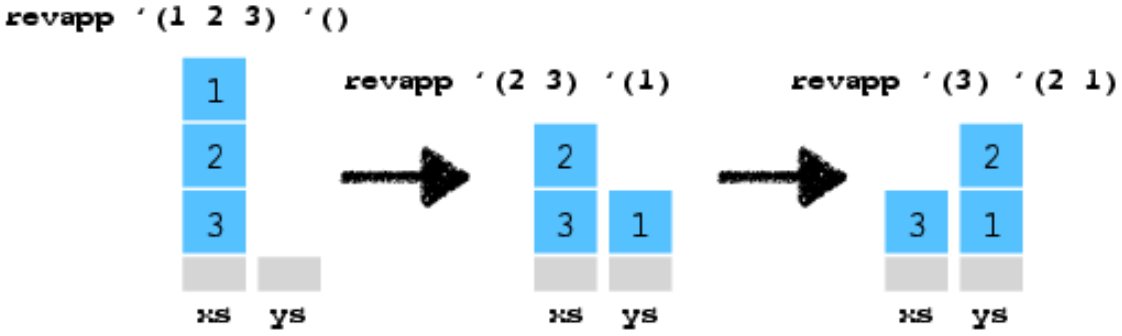




Take 2 off front of xs and put on front of ys

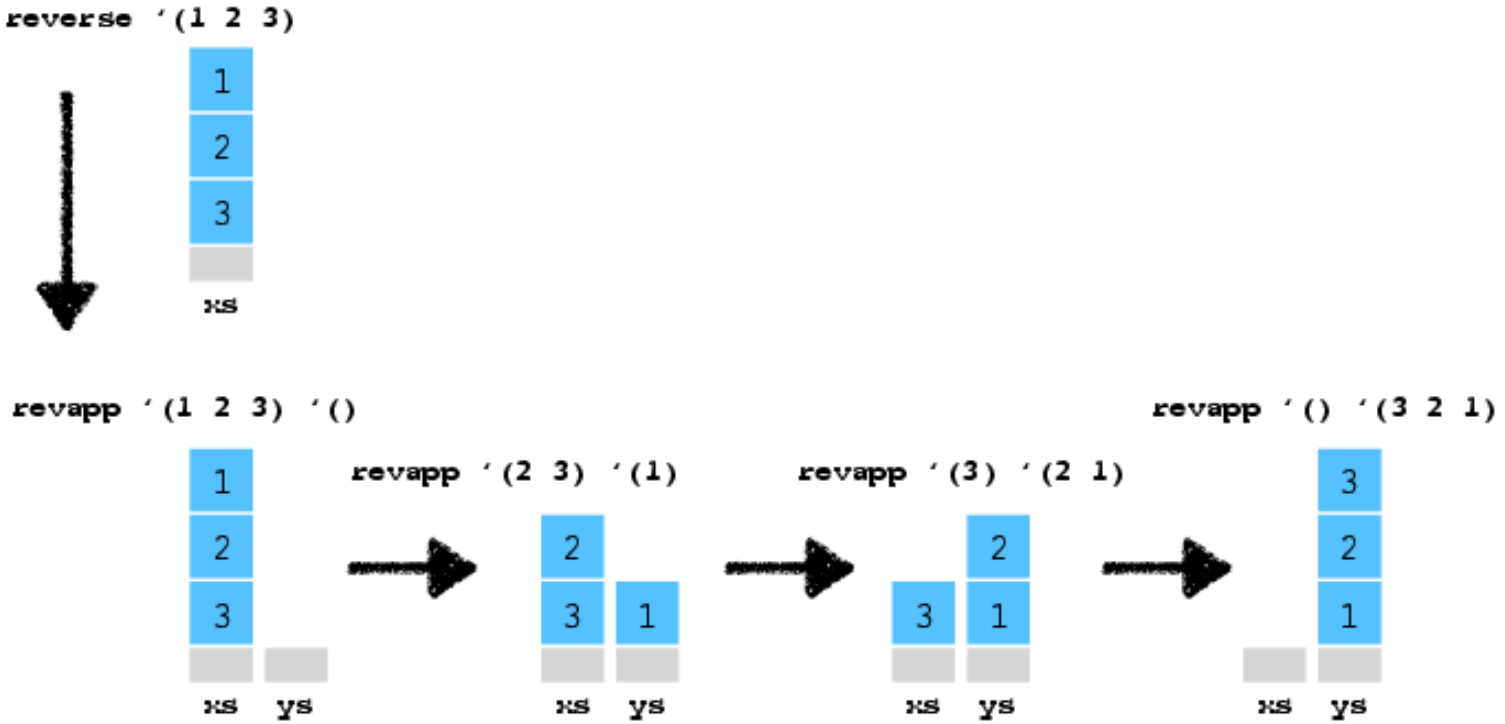

```
(define revapp (xs ys)
   (if (null? xs)
       ys
       (revapp (cdr xs)
               (cons (car xs) ys))))

(define reverse (xs) (revapp xs '()))
```





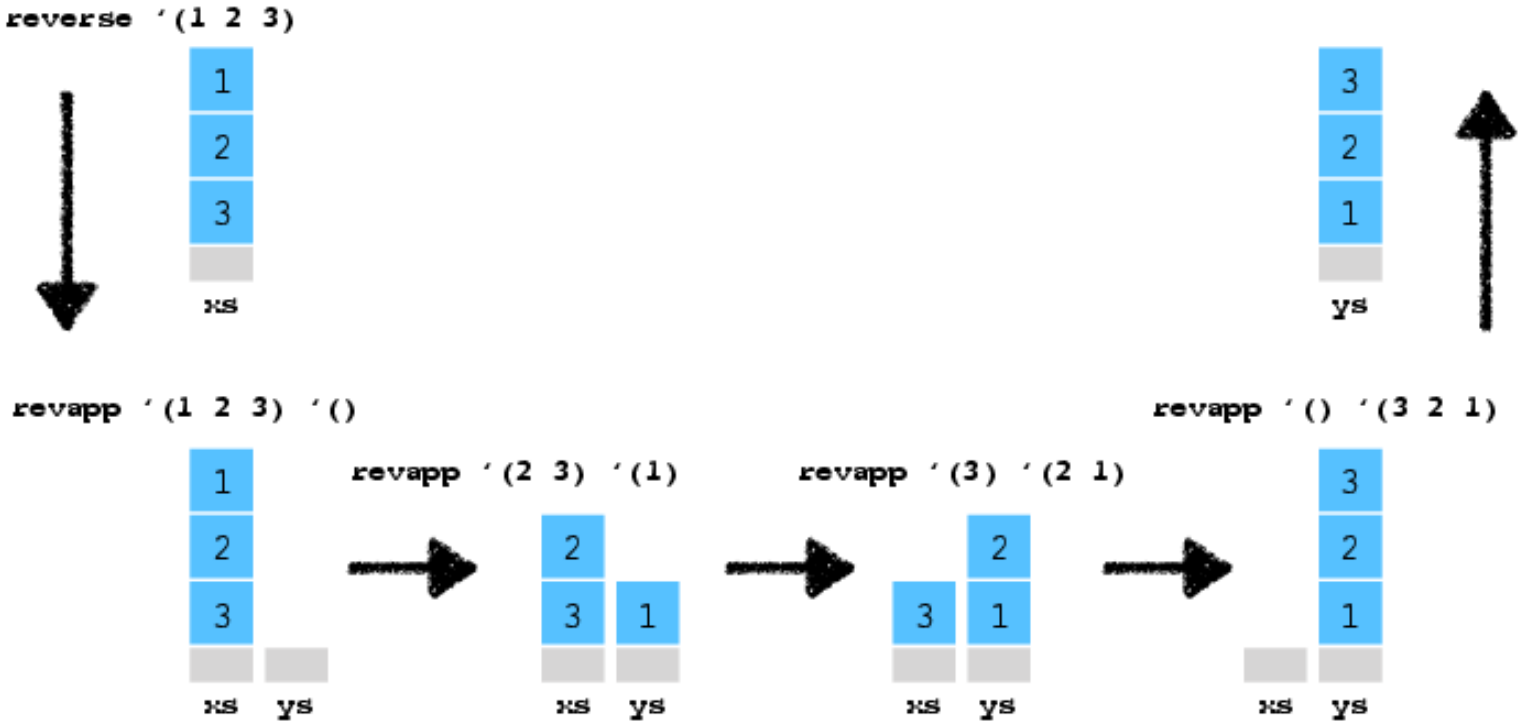
Take 3 off front of xs and put on front of ys


```
(define revapp (xs ys)
   (if (null? xs)
       ys
       (revapp (cdr xs)
               (cons (car xs) ys))))

(define reverse (xs) (revapp xs '()))
```



Finish helper recursion, which returns ys


```
(define revapp (xs ys)
   (if (null? xs)
       ys
       (revapp (cdr xs)
               (cons (car xs) ys))))

(define reverse (xs) (revapp xs '()))
```



Summary: Method of accumulating parameters

- **O(N^2) is usually too expensive**

```
(define reverse (xs)
    (if (null? xs)
        `()
        (append (reverse (cdr xs)) (list1 (car xs)))))
```

- **Instead write a helper function revapp that uses an accumulating parameter, O(N)**

```
(define reverse (xs)
    (revapp xs `()))

(define revapp (xs ys)
    (if (null? xs)
        ys
        (revapp (cdr xs) (cons (car xs) ys))))
```

Course Project

THE UNIVERSITY OF ARIZONA
Computer Science

https://github.com/UofA-CSc-520-Spring-2020/CSc520Spr20-CourseMaterials/blob/master/project.md

- **Project Proposal is due this Friday**
  - Submit as a pdf to Gradescope
  - For TWO papers from previous PLDI, POPL, or ICPF
    - Full citation
    - A paragraph describing the aspects of the paper you plan to present at end of the semester
  - Name of your project partner if relevant
- **Project Talks**
  - Slides are due Tuesday April 28th
  - April 29th and May 4th, Project talks
  - 10 minutes total, 5 minutes per person if done with a partner
  - ➔ See project.md for talk expectations

Example Half of a Project Proposal

"Synthesizing Parallel Graph Programs via Automated Planning," by Prountzos, Manevich, and Pingali, PLDI 2015.

This paper describes an approach for synthesizing efficient parallel graph algorithm implementations by specifying the algorithm in a high-level specification language and then developing constraints for solving the instruction selection, computation scheduling, and parallelization problems as a combined automated planning problem. Table 4a presented an attributed grammar for the algorithm specification language. I will present this table and its terminology and compare the concepts to at least the AST and operational semantics concepts from CSc 520.

"Synthesizing Parallel Graph Programs via Automated Planning," by Prountzos, Manevich, and Pingali, PLDI 2015.

- **Problem:**
  - Vast implementation space for sparse graph algorithms
  - Especially when targeting a variety of parallel architectures
  - Existing high-level specification languages exist, but it is difficult to select from space of possible implementations to generate
- **Motivation: applications to web search and machine learning**
- **Summary of approach**
  - Use automated planning to synthesize correct and efficient parallel graph programs from high-level specification language.
  - Always competitive with hand-optimized implementations and sometimes outperforms.

Concepts from this course in Table 4a

- **Meta variables**
- **Productions specify the AST**
- **The semantic rules are doing a semantic/type analysis**

Meta Variable	Description
x	A program variable $x \in Var$
b	Boolean expression
r	Data range expression
rs	A sequence of range expressions
upd	State update

Attribute	Value Type (inherited/synthesized)
$bnd(\cdot)$	2^{Var} (inherited)
$vars(\cdot)$	2^{Var} (synthesized)

Production	Semantic Rules
$S ::=$ **for** $x : r$ **do**^{L} B_1 **od**^{L}	if $x \in bnd(S)$ then error, if $vars(r) \nsubseteq bnd(S)$ then error, $bnd(B_1) = bnd(S) \cup \{x\}$.
| **while** b **do**^{L} B_2 **od**^{L}	if $vars(b) \nsubseteq bnd(S)$ then error, $bnd(B_2) = bnd(S)$.
| **if** b^L B_t **else**^{L} **skip**^{L} **fi**^{L}	if $vars(b) \nsubseteq bnd(S)$ then error, $bnd(B_t) = bnd(S)$.
$B ::= S$	$bnd(S) = bnd(B)$.
| A	$bnd(A) = bnd(B)$.
$A ::= AtomUpd$	$bnd(AtomUpd) = bnd(A)$.
| **acq** rs_1 **ctx** rs_2^L ; A_1	if $vars(rs_1, rs_2) \nsubseteq bnd(A)$ then error. $bnd(A_1) = bnd(A)$.
| **if** b^L A_t **else**^{L} R **exit**^{L} i	if $vars(b) \nsubseteq bnd(A)$ then error, $bnd(A_t) = bnd(R) = bnd(A)$.
| **for** $x : r$ **do**^{L} A_b **od**^{L}	if $\neg val(r)$ then error, if $vars(r) \nsubseteq bnd(A)$ then error, if $x \in bnd(A)$ then error, $bnd(A_b) = bnd(A) \cup \{x\}$.
$R ::= \epsilon$	
| **rel** rs^L	if $vars(rs) \nsubseteq bnd(R)$ then error.

Table 4a Details

- **Local vars do not hide others & only used in scope**
  - **for production, x in bnd(S) then error shows hiding is an error**
  - **if vars(r) not in bnd(S) then errors enforce using in scope**
- **At most one atomic section**
  - **The "A" nonterminal can only expand to AtomUpd once**
- **Updates only innermost**
  - **From AtomUpd cannot generate an S, B, or A**

Meta Variable	Description
x	A program variable $x \in Var$
b	Boolean expression
r	Data range expression
rs	A sequence of range expressions
upd	State update

Attribute	Value Type (inherited/synthesized)
$bnd(\cdot)$	2^{Var} (inherited)
$vars(\cdot)$	2^{Var} (synthesized)

Production	Semantic Rule
$S ::=$ **for** $x : r$ **do**^{L} B_1 **od**^{L}	if $x \in bnd(S)$ then error, if $vars(r) \not\subseteq bnd(S)$ then error, $bnd(B_1) = bnd(S) \cup \{x\}$.
| **while** b **do**^{L} B_2 **od**^{L}	if $vars(b) \not\subseteq bnd(S)$ then error, $bnd(B_2) = bnd(S)$.
| **if** b^L B_t **else**^{L} **skip**^{L} **fi**^{L}	if $vars(b) \not\subseteq bnd(S)$ then error, $bnd(B_t) = bnd(S)$.
$B ::= S$	$bnd(S) = bnd(B)$.
| A	$bnd(A) = bnd(B)$.
$A ::= AtomUpd$	$bnd(AtomUpd) = bnd(A)$.
| **acq** rs_1 **ctx** rs_2^L; A_1	if $vars(rs_1, rs_2) \not\subseteq bnd(A)$ then error, $bnd(A_1) = bnd(A)$.
| **if** b^L A_t **else**^{L} R **exit**^{L} **fi**^{L}	if $vars(b) \not\subseteq bnd(A)$ then error, $bnd(A_t) = bnd(R) = bnd(A)$.
| **for** $x : r$ **do**^{L} A_b **od**^{L}	if $vars(r)$ then error, if $vars(r) \not\subseteq bnd(A)$ then error, if $x \in bnd(A)$ then error, $bnd(A_b) = bnd(A) \cup \{x\}$.
$R ::= \epsilon$	
| **rel** rs^L	if $vars(rs) \not\subseteq bnd(R)$ then error.
$AtomUpd ::= Upd; R;$ **commit**	$bnd(Upd) = bnd(R) = bnd(AtomUpd)$.
$Upd ::= upd^L$	if $vars(upd) \not\subseteq bnd(Upd)$ then error.
| **acq** rs_1 **ctx** rs_2^L	if $vars(rs_1, rs_2) \not\subseteq bnd(Upd)$ then error.
| $Upd_1; Upd_2$	$bnd(Upd_1) = bnd(Upd_2) = bnd(Upd)$.

Let construct

- **To introduce local names into environment**

```
(let ([x1 e1]
      ...
      [xn en])
   e)
```

- **Evaluate e1 through en, bind answers to x1, …, xn**

$$\frac{\begin{array}{c} x_1, \ldots, x_n \text{ all distinct} \\ \ell_1, \ldots, \ell_n \notin \operatorname{dom} \sigma_n \text{ (and all distinct)} \\ \sigma_0 = \sigma \\ \langle e_1, \rho, \sigma_0 \rangle \Downarrow \langle v_1, \sigma_1 \rangle \\ \vdots \\ \langle e_n, \rho, \sigma_{n-1} \rangle \Downarrow \langle v_n, \sigma_n \rangle \\ \langle e, \rho\{x_1 \mapsto \ell_1, \ldots, x_n \mapsto \ell_n\}, \sigma_n\{\ell_1 \mapsto v_1, \ldots, \ell_n \mapsto v_n\} \rangle \Downarrow \langle v, \sigma' \rangle \end{array}}{\langle \text{LET}(\langle x_1, e_1, \ldots, x_n, e_n \rangle, e), \rho, \sigma \rangle \Downarrow \langle v, \sigma' \rangle} \quad (\text{LET})$$

Let* construct

- **How is let* different than let?**

```
(let* ([x1 e1]
       ...
       [xn en])
    e)
```

$$\frac{\begin{array}{c}\rho_0 = \rho \qquad \sigma_0 = \sigma \\ \langle e_1, \rho_0, \sigma_0\rangle \Downarrow \langle v_1, \sigma_0'\rangle \qquad \ell_1 \notin \operatorname{dom}\sigma_0' \qquad \rho_1 = \rho_0\{x_1 \mapsto \ell_1\} \qquad \sigma_1 = \sigma_0'\{\ell_1 \mapsto v_1\} \\ \vdots \\ \langle e_n, \rho_{n-1}, \sigma_{n-1}\rangle \Downarrow \langle v_n, \sigma_{n-1}'\rangle \\ \ell_n \notin \operatorname{dom}\sigma_{n-1}' \qquad \rho_n = \rho_{n-1}\{x_n \mapsto \ell_n\} \qquad \sigma_n = \sigma_{n-1}'\{\ell_n \mapsto v_n\} \\ \langle e, \rho_n, \sigma_n\rangle \Downarrow \langle v, \sigma'\rangle\end{array}}{\langle \textsc{letstar}(\langle x_1, e_1, \ldots, x_n, e_n\rangle, e), \rho, \sigma\rangle \Downarrow \langle v, \sigma'\rangle} \quad (\textsc{Letstar})$$

Letrec construct

- **How is letrec different than let and let*?**

```
(letrec ([x1 e1]
         ...
         [xn en])
    e)
```

$$\frac{\begin{array}{c}\ell_1, \ldots, \ell_n \notin \operatorname{dom}\sigma \text{ (and all distinct)} \\ x_1, \ldots, x_n \text{ all distinct} \\ e_i \text{ has the form } \textsc{lambda}(\cdots),\ 1 \le i \le n \\ \rho' = \rho\{x_1 \mapsto \ell_1, \ldots, x_n \mapsto \ell_n\} \qquad \sigma_0 = \sigma\{\ell_1 \mapsto \text{unspecified}, \ldots, \ell_n \mapsto \text{unspecified}\} \\ \langle e_1, \rho', \sigma_0\rangle \Downarrow \langle v_1, \sigma_1\rangle \\ \vdots \\ \langle e_n, \rho', \sigma_{n-1}\rangle \Downarrow \langle v_n, \sigma_n\rangle \\ \langle e, \rho', \sigma_n\{\ell_1 \mapsto v_1, \ldots, \ell_n \mapsto v_n\}\rangle \Downarrow \langle v, \sigma'\rangle\end{array}}{\langle \textsc{letrec}(\langle x_1, e_1, \ldots, x_n, e_n\rangle, e), \rho, \sigma\rangle \Downarrow \langle v, \sigma'\rangle} \quad (\textsc{Letrec})$$