Recursion, tail recursion

- Recursion is the lifeblood of repetition in lisp
- Even loop functions are implemented recursively
- Typical function setup:
  - Error-check parameters
  - Check for base cases
  - Handle general/recursive cases
- We’ll discuss recursion’s efficiency issues, then address them through tail recursion

Example: smallest in a list

- Find smallest integer in list, skip non-integers, return most-positive-fixnum if list contains no integers, nil if not a list

```
(defun smallest (L)
   (cond
      ((not (listp L)) nil)
      ((null L) most-positive-fixnum)
      ((not (integerp (car L)) (smallest (cdr L))
      ((< (car L) (smallest (cdr L))) (car L))
      (t (smallest (cdr L)))))
```

Recursion use of stack space

- Major drawback: every recursive call generates a new stack frame: deep recursion can use all your stack space
- Tail-recursive functions, together with a suitable compiler or interpretter, can avoid this
- A function is tail-recursive iff the result of each recursive call is immediately returned, i.e. not processed before returning

Tail recursive: example

- Tail recursive: each recursive call returns immediately

```
(defun f (x)
   (cond
      ((not (numberp x)) nil)
      ((> 0 x) (f (- x)))
      ((< 1 x) (f (/ 1 x))
      (t (sqrt x))))
```

Not tail recursive: example

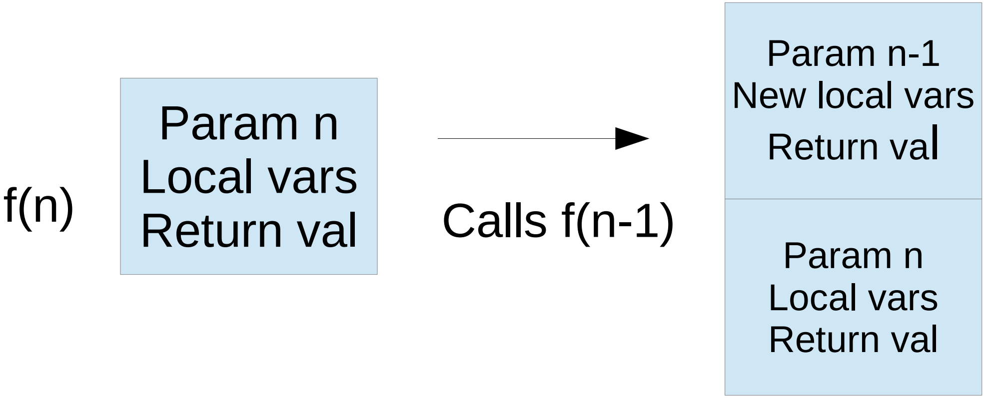
- Not tail recursive: result of at least one recursive call gets processed before the return

```
(defun f (x)
   (cond
      ((not (numberp x)) nil)
      ((> 0 x) (f (- x)))
      ((< 1 x) (* 10 (f (/ 1 x)))
      (t (sqrt x))))
```

How does tail recursion help?

- Function makes its recursive call and immediately returns
- After making the recursive call, the tail recursive function no longer actually uses any of the data in its stack frame
- If compiler recognizes that then it can actually overwrite the stack frame with the stack frame for the recursive call
- Frame profile is exactly the same, same layout for parameters, local vars, etc: just needs to rewrite the actual parameter values for the new call
- The recursive calls thus use no extra stack space

Stack with/without tail rec optimization



f(n)

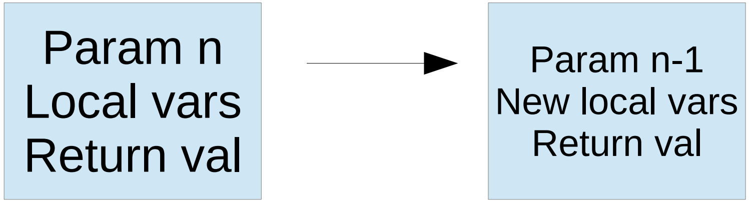
Param n
Local vars
Return val

Calls f(n-1)

Param n-1
New local vars
Return val

Param n
Local vars
Return val

Without optimization



Param n
Local vars
Return val

Param n-1
New local vars
Return val

With optimization

smallest in a list: tail recursive

“smallest” example earlier wasn’t tail recursive, see line:

```
((< (car L) (smallest (cdr L))) (car L))
```

- Rewrite smallest in a tail-recursive fashion
- Add a helper function that takes an extra parameter: the smallest value seen *so far*
- smallest function calls helper function, with most-positive-fixnum as the starting ‘sofar’ value

```
(defun smallest (L)
   (if (listp L) (smallhelper L most-positive-fixnum)))
```

(smallhelper L sofar)

- Smallhelper trusts L is a list, sofar is smallest value so far
- If L is empty return sofar, else compare front element to sofar and call recursively with new smallest so far

```
(defun smallhelper (L sofar)
   (cond
      ((null L) sofar)
      ((not (integerp (car L))) (smallhelper (cdr L) sofar))
      ((< (car L) sofar) (smallhelper (cdr L) (car L)))
      (t (smallhelper (cdr L) sofar))))
```

Accumulators

- The “sofar” parameter we added to simplify our recursive algorithm is called an accumulator
- Accumulators are widely used to create tail recursive algorithms
- In fact, any loop-based function can be turned into a tail recursive one using an appropriate collection of accumulators

Turning loops into tail recursion: C style

```
int f(int x) {
    int y = 0;
    while (y < x) {
        print(y);
        y++;
    }
    return y;
}
```

```
int f(int x, int y = 0) {
    if (y >= x) return y;
    else {
        print(y);
        return f(x, y+1);
    }
}
```

Function locals replaced with parameters,
initialization replaced with default values
Updates to variables in loop replaced with updates to values in recursive call