Recursion in C

What does the following program compute?

```
int f(int n, int v) {
  if (n == 0)
    return v;
  else
    return f(n - 1, v + n);
}

int main() { return f(1000000, 0); }
```

Answer: stack overflow

Loops in C

What does the following program compute?

```
int main() {
  int n = 1000000;
  int v = 0;

  while (n > 0) {
    v = v + n;
    n = n - 1;
  }

  return v;
}
```

Answer: the sum of 0 to 1000000 (in 32-bit two's complement)

Recursion in the Book Language

What does the following program compute?

```
letrec f = proc(n, v)
               if n
                 then (f -(n,1) +(n, v))
                 else v
   in (f 1000000 0)
```

Answer: the sum of 0 to 1000000

Recursion in the Book Language

Why don't we get a stack overflow in the book language?

- This is actually a question about Scheme:
  - We implement recursion for the book language by using Scheme's recursion
  - Similarly, we use Scheme numbers to implement numbers for the book language
- Such an explanation/implementation is called ***meta-circular***
- We don't care so much about numbers, but we don't want to explain away recursion; we want to understand recursion (in Scheme and the book language)

Recursion in Scheme

What does the following program compute?

```
(define (f n v)
  (if (zero? n)
      v
      (f (- n 1) (+ n v))))

(f 1000000 0)
```

Answer: the sum of 0 to 1000000

How?

Recursion in Scheme

```
(define (f n v)
  (if (zero? n)
      v
      (f (- n 1) (+ n v))))

(f 1000000 0)
```

→

```
(define (f n v)
  (if (zero? n)
      v
      (f (- n 1) (+ n v))))

(if (zero? 1000000)
    0
    (f (- 1000000 1) (+ 1000000 0))))
```

Recursion in Scheme

```
(define (f n v)
  (if (zero? n)
      v
      (f (- n 1) (+ n v))))

(if (zero? 1000000)
    0
    (f (- 1000000 1) (+ 1000000 0))))
```

→

```
(define (f n v)
  (if (zero? n)
      v
      (f (- n 1) (+ n v))))

(f (- 1000000 1) (+ 1000000 0))
```

Recursion in Scheme

```
(define (f n v)
  (if (zero? n)
      v
      (f (- n 1) (+ n v))))

(f (- 1000000 1) (+ 1000000 0))
```

→

```
(define (f n v)
  (if (zero? n)
      v
      (f (- n 1) (+ n v))))

(f 999999 1000000)
```

Recursion in Scheme

```
(define (f n v)
  (if (zero? n)
      v
      (f (- n 1) (+ n v))))

(f 999999 1000000)
```

→

```
(define (f n v)
  (if (zero? n)
      v
      (f (- n 1) (+ n v))))

(if (zero? 999999)
    1000000
    (f (- 999999 1) (+ 999999 1000000))))
```

Recursion in Scheme

```
(define (f n v)
  (if (zero? n)
      v
      (f (- n 1) (+ n v))))

(if (zero? 999999)
    1000000
    (f (- 999999 1) (+ 999999 1000000))))
```

→

```
(define (f n v)
  (if (zero? n)
      v
      (f (- n 1) (+ n v))))

(f (- 999999 1) (+ 999999 1000000))
```

Recursion in Scheme

```
(define (f n v)
  (if (zero? n)
      v
      (f (- n 1) (+ n v))))

(f (- 999999 1) (+ 999999 1000000))
```

→

```
(define (f n v)
  (if (zero? n)
      v
      (f (- n 1) (+ n v))))

(f 999998 1999999)
```

Recursion in Scheme

```
(define (f n v)
  (if (zero? n)
      v
      (f (- n 1) (+ n v))))

(f 999998 1999999)
```

→

```
(define (f n v)
  (if (zero? n)
      v
      (f (- n 1) (+ n v))))

...
```

Recursion in Scheme

```
(define (f n v)
  (if (zero? n)
      v
      (f (- n 1) (+ n v))))

...
```

→

```
(define (f n v)
  (if (zero? n)
      v
      (f (- n 1) (+ n v))))

(if (zero? 0)
    500000500000
    (f (- 0 1) (+ 0 500000500000))))
```

Recursion in Scheme

```
(define (f n v)
  (if (zero? n)
      v
      (f (- n 1) (+ n v))))

(if (zero? 0)
    500000500000
    (f (- 0 1) (+ 0 500000500000))))
```

→

```
(define (f n v)
  (if (zero? n)
      v
      (f (- n 1) (+ n v))))

500000500000
```

Recursion in Scheme

- Our definition of Scheme says nothing about a *stack*
- Each step from a small program produces a small program...
- We can forget the old small program after each step...
- So there's no reason to run out of anything!

- Does that mean that we can write anything, without worrying about running out of space?

Recursion in Scheme

What does the following program compute?

```
(define (f2 n)
  (if (zero? n)
      0
      (+ n (f2 (- n 1)))))

(f2 1000000)
```

Answer: out of memory, maybe

Recursion in Scheme

```
(define (f2 n)
  (if (zero? n)
      0
      (+ n (f2 (- n 1)))))
```

```
(f2 1000000)
```

→

```
(define (f2 n)
  (if (zero? n)
      0
      (+ n (f2 (- n 1)))))
```

```
(if (zero? 1000000)
    0
    (+ 1000000 (f2 (- 1000000 1))))
```

Recursion in Scheme

```
(define (f2 n)
  (if (zero? n)
      0
      (+ n (f2 (- n 1)))))
```

```
(if (zero? 1000000)
    0
    (+ 1000000 (f2 (- 1000000 1))))
```

→

```
(define (f2 n)
  (if (zero? n)
      0
      (+ n (f2 (- n 1)))))
```

```
(+ 1000000 (f2 (- 1000000 1)))
```

Recursion in Scheme

```
(define (f2 n)
  (if (zero? n)
      0
      (+ n (f2 (- n 1)))))
```

```
(+ 1000000 (f2 (- 1000000 1)))
```

→

```
(define (f2 n)
  (if (zero? n)
      0
      (+ n (f2 (- n 1)))))
```

```
(+ 1000000 (f2 999999))
```

Recursion in Scheme

```
(define (f2 n)
  (if (zero? n)
      0
      (+ n (f2 (- n 1)))))
```

```
(+ 1000000 (f2 999999))
```

→

```
(define (f2 n)
  (if (zero? n)
      0
      (+ n (f2 (- n 1)))))
```

```
(+ 1000000
   (if (zero? 999999)
       0
       (+ 999999 (f2 (- 999999 1)))))
```

Recursion in Scheme

```
(define (f2 n)
  (if (zero? n)
      0
      (+ n (f2 (- n 1)))))
```

```
(+ 1000000
   (if (zero? 999999)
       0
       (+ 999999 (f2 (- 999999 1)))))
```

→

```
(define (f2 n)
  (if (zero? n)
      0
      (+ n (f2 (- n 1)))))
```

```
(+ 1000000
   (+ 999999 (f2 (- 999999 1))))
```

Recursion in Scheme

```
(define (f2 n)
  (if (zero? n)
      0
      (+ n (f2 (- n 1)))))
```

```
(+ 1000000
   (+ 999999 (f2 (- 999999 1))))
```

→

```
(define (f2 n)
  (if (zero? n)
      0
      (+ n (f2 (- n 1)))))
```

```
(+ 1000000
   (+ 999999 (f2 999998)))
```

Recursion in Scheme

```
(define (f2 n)
  (if (zero? n)
      0
      (+ n (f2 (- n 1)))))
```

```
(+ 1000000
   (+ 999999 (f2 999998)))
```

→

```
(define (f2 n)
  (if (zero? n)
      0
      (+ n (f2 (- n 1)))))
```

```
...
```

Recursion in Scheme

```
(define (f2 n)
  (if (zero? n)
      0
      (+ n (f2 (- n 1)))))
```

...

→

```
(define (f2 n)
  (if (zero? n)
      0
      (+ n (f2 (- n 1)))))
```

```
(+ 1000000
   (+ 999999
      (+ 999998 ... (+ 1 0))))
```

The Cost of Recursion

- Computing **`(f`** *n***`)`** takes O(1) space
- Computing **`(f2`** *n***`)`** takes O(*n*) space

- In Scheme, we write loops and more general forms of recursion in the same way, but there's still a difference in costs
- How does a Scheme programmer write a loop?

Loops in Scheme

- A function (or set of functions) acts like a loop if every self-call is a ***tail call***
- A tail call is a function call whose value is the final result for the caller

```
(define (f n v)
  (if (zero? n)
      v
      (f (- n 1) (+ n v))))
```

a tail call

Loops in Scheme

- A function (or set of functions) acts like a loop if every self-call is a ***tail call***
- A tail call is a function call whose value is the final result for the caller

```
(define (f2 n)
  (if (zero? n)
      0
      (+ n (f2 (- n 1)))))
```

not a tail call

Loops in Scheme

- A function (or set of functions) acts like a loop if every self-call is a ***tail call***
- A tail call is a function call whose value is the final result for the caller

```
(define (odd n)
  (if (zero? n)
      #f
      (even (- n 1))))
(define (even n)
  (if (zero? n)
      #t
      (odd (- n 1))))
```

tail calls

Implementing Control Explicitly

- Ok, so how is it done?

- We'll change our interpreter to make control explicit
- Let's first see what a trace of evaluation should look like

Evaluation

1

exp= 1

env= {}

Done!

Evaluation

+(1, 2)

exp= +(1, 2)
env= {}

exp= 1
env= {}

Done?

Evaluation

+(1, 2)

exp= +(1, 2)
env= {}

exp= 1
env= {}

exp= 2
env= {}

How do we know when we're done?

How do we know what's left to do?

Evaluation with To-Do List

1

exp= 1
env= {}
todo= [**done**]

- Keep a to-do list, passed to evaluator

Evaluation with To-Do List

1

exp= 1
env= {}
todo= [**done**]

val= 1
todo= [**done**]

- When we get a value, go into to-do-checking mode

Evaluation with To-Do List

1

exp= 1
env= {}
todo= [**done**]

val= 1
todo= [**done**]

Done!

Evaluation with To-Do List

+(1, 2)

exp= +(1, 2)
env= {}
todo= [**done**]

exp= 1
env= {}
todo= [**addexp** 2 **in** {} **then** [**done**]]

- When evaluating sub-expressions, extend the to-do list
- **addexp** is an abbreviation for:
  remember the result, evaluate another expression,
  then add the two results

Evaluation with To-Do List

+(1, 2)

exp= +(1, 2)
env= {}
todo= [**done**]

exp= 1
env= {}
todo= [**addexp** 2 **in** {} **then** [**done**]]

val= 1
todo= [**addexp** 2 **in** {} **then** [**done**]]

Evaluation with To-Do List

val= 1
todo= [**addexp** 2 **in** {} **then** [**done**]]

exp= 2
env= {}
todo= [**addval** 1 **then** [**done**]]

- To do **addexp**, we start evaluating the remembered expression in the remembered environment
- Extend to-do list to remember the value we already have, and remember to do an addition later
- **addval** is an abbreviation for:
  add the result with a remembered result

Evaluation with To-Do List

val= 1
todo= [**addexp** 2 **in** {} **then** [**done**]]

exp= 2
env= {}
todo= [**addval** 1 **then** [**done**]]

val= 2
todo= [**addval** 1 **then** [**done**]]

val= 3
todo= [**done**]

Done!

Evaluation with To-Do List

+(1, +(2, 3))

exp= +(1, +(2, 3))
env= {}
todo= [**done**]

Evaluation with To-Do List

+(1, +(2, 3))

exp= +(1, +(2, 3))
env= {}
todo= [**done**]

exp= 1
env= {}
todo= [**addexp** +(2, 3) **in** {} **then** [**done**]]

Evaluation with To-Do List

+(1, +(2, 3))

exp= 1
env= {}
todo= [**addexp** +(2, 3) **in** {} **then** [**done**]]

val= 1
todo= [**addexp** +(2, 3) **in** {} **then** [**done**]]

Evaluation with To-Do List

+(1, +(2, 3))

val= 1

todo= [**addexp** +(2, 3) **in** {} **then** [**done**]]

exp= +(2, 3)

env= {}

todo= [**addval** 1 **then** [**done**]]

Evaluation with To-Do List

+(1, +(2, 3))

exp= +(2, 3)
env= {}
todo= [**addval** 1 **then** [**done**]]

exp= 2
env= {}
todo= [**addexp** 3 **in** {} **then** [**addval** 1 **then** [**done**]]]

Evaluation with To-Do List

+(1, +(2, 3))

exp= 2
env= {}
todo= [**addexp** 3 **in** {} **then** [**addval** 1 **then** [**done**]]]

val= 2
todo= [**addexp** 3 **in** {} **then** [**addval** 1 **then** [**done**]]]

Evaluation with To-Do List

+(1, +(2, 3))

val= 2

todo= [**addexp** 3 **in** {} **then** [**addval** 1 **then** [**done**]]]

exp= 3

env= {}

todo= [**addval** 2 **then** [**addval** 1 **then** [**done**]]]

Evaluation with To-Do List

+(1, +(2, 3))

exp= 3
env= {}
todo= [**addval** 2 **then** [**addval** 1 **then** [**done**]]]

val= 3
todo= [**addval** 2 **then** [**addval** 1 **then** [**done**]]]

Evaluation with To-Do List

+(1, +(2, 3))

val= 3
todo= [**addval** 2 **then** [**addval** 1 **then** [**done**]]]

val= 5
todo= [**addval** 1 **then** [**done**]]

Evaluation with To-Do List

+(1, +(2, 3))

val= 5
todo= [**addval** 1 **then** [**done**]]

val= 6
todo= [**done**]

Evaluation with To-Do List

```
let f = proc(y)y
  in (f 10)
```

exp= **let f** = **proc**(**y**)**y in** (**f** 10)

env= {}

todo= [**done**]

Evaluation with To-Do List

let f = **proc**(**y**)**y**
in (**f** 10)

exp= **let f** = **proc**(**y**)**y in** (**f** 10)
env= {}
todo= [**done**]

exp= **proc**(**y**)**y**
env= {}
todo= [**let f in** (**f** 10) {} **then** [**done**]]

Evaluation with To-Do List

```
let f = proc(y)y
  in (f 10)
```

exp= proc(y)y
env= {}
todo= [let f in (f 10) {} then [done]]

val= <y,y,{}>
todo= [let f in (f 10) {} then [done]]

Evaluation with To-Do List

let f = **proc**(**y**)**y**
in (**f** 10)

val= <**y**,**y**,{}>
todo= [**let f in** (**f** 10) {} **then** [**done**]]

exp= (**f** 10)
env= {**f**=<**y**,**y**,{}>}
todo= [**done**]

Evaluation with To-Do List

```
let f = proc(y)y
  in (f 10)
```

exp= (**f** 10)
env= {**f**=<**y**,**y**,{}>}
todo= [**done**]

exp= **f**
env= {**f**=<**y**,**y**,{}>}
todo= [**apparg** 10 **in** {**f**=<**y**,**y**,{}>} **then** [**done**]]

Evaluation with To-Do List

let f = **proc**(**y**)**y**
in (**f** 10)

exp= **f**
env= {**f**=<**y**,**y**,{}>}
todo= [**apparg** 10 **in** {**f**=<**y**,**y**,{}>} **then** [**done**]]

val= <**y**,**y**,{}>
todo= [**apparg** 10 **in** {**f**=<**y**,**y**,{}>} **then** [**done**]]

Evaluation with To-Do List

let f = **proc**(**y**)**y**
in (**f** 10)

val= <**y**,**y**,{}>
todo= [**apparg** 10 **in** {**f**=<**y**,**y**,{}>} **then** [**done**]]

exp= 10
env= {**f**=<**y**,**y**,{}>}
todo= [**app** <**y**,**y**,{}> **then** [**done**]]

Evaluation with To-Do List

let f = **proc**(**y**)**y**
in (**f** 10)

exp= 10
env= {**f**=<**y**,**y**,{}>}
todo= [**app** <**y**,**y**,{}> **then** [**done**]]

val= 10
todo= [**app** <**y**,**y**,{}> **then** [**done**]]

Evaluation with To-Do List

let f = **proc**(**y**)**y**
in (**f** 10)

val= 10
todo= [**app** <**y**,**y**,{}> **then** [**done**]]

exp= **y**
env= {**y**=10}
todo= [**done**]

Evaluation with To-Do List

```
let f = proc(y)y
  in (f 10)
```

exp= **y**
env= {**y**=10}
todo= [**done**]

val= 10
todo= [**done**]

To-Do Lists

- To-do list is called the ***continuation***
- It makes the Scheme context in our interpreter explicit

The interpreter now consists of two main functions:

- **`eval-expression : expr env cont -> expval`**

exp= 1
env= {}
todo= [**done**]

- **`apply-cont : value cont -> expval`**

val= 1
todo= [**done**]

Continuation Datatype

```
(define-datatype
 continuation continuation?
 (done-cont)
 (app-arg-cont (rand expression?)
               (env environment?)
               (cont continuation?))
 (app-cont (rator value?)
           (cont continuation?))
 ...)
```

Continuation Datatype

[**done**]

`=`

`(done-cont)`

[**addval** 1 **then** [**done**]]

`=`

`(prim-cont (add-prim) 1 (done-cont))`

[**addexp y in** {**y**=10} **then** [**done**]]

`=`

```
(prim-other-cont
  (add-prim)  (var-exp 'y)
  (extend-env '(y) '(10) (empty-env))
  (done-cont))
```

Continuation Datatype

[**let f in** (**f** 10) {} **then** [**done**]]

```
=
(let-cont 'f (app-exp (var-exp 'f)
                      (list-exp 10))
          (empty-env)
          (done-cont))
```

Interpreter

```
(define eval-program
  (lambda (pgm)
   (cases program pgm
     (a-program (body)
        (eval-expression body
                         (init-env)
                         (done-cont))))))
```

Interpreter

```
(define (eval-expression exp env cont)
  (cases expression exp
    (lit-exp (datum)
      (apply-cont cont datum))
    (var-exp (id)
      (apply-cont cont (apply-env env id)))
    (proc-exp (id body-exp)
      (apply-cont cont
                  (closure id body-exp env)))
    ...)))

(define (apply-cont cont val)
  (cases continuation cont
    (done-cont () val)
    ...))
```

Interpreter: Let

```
... ; in eval-expression:
(let-exp (id exp body-exp)
   (eval-expression
    exp env
    (let-cont id body-exp env cont)))
...
... ; in apply-cont:
(let-cont (id body env cont)
   (eval-expression
    body (extend-env (list id) (list val)
                     env)
    cont))
...
```

Interpreter: Primitives

```
... ; in eval-expression:
(primapp-exp (prim rand1 rand2)
   (eval-expression
    rand1 env
    (prim-other-cont prim rand2 env cont)))
...
... ; in apply-cont:
(prim-other-cont (prim arg2 env cont)
   (eval-expression
    arg2 env
    (prim-cont prim val cont)))
(prim-cont (prim arg1-val cont)
   (apply-cont
    cont
    (apply-primitive prim arg1-val val)))
...
```

Interpreter: Application

```
... ; in eval-expression:
(app-exp (rator rand)
  (eval-expression
   rator env
   (app-arg-cont rand env cont)))
...
...  ; in apply-cont:
(app-arg-cont (rand env cont)
   (eval-expression rand env
                    (app-cont val cont)))
 (app-cont (f cont)
   (apply-proc f val cont))
...
```

Interpreter: If

```
... ; in eval-expression:
(if-exp (test then else)
   (eval-expression
    test env
    (if-cont then else env cont)))
...
...  ; in apply-cont:
(if-cont (then else env cont)
   (eval-expression
    (if (zero? val) else then)
    env cont))
...
```

Interpreter: Complete Implementation

(in DrScheme, instrumented to show traces)

Variable and Control Stacks

One more way to look at continuations:

- environment = variable stack
- continuation = control stack

In most imperative languages (e.g., C, Java), the variable stack and control stack are merged

That's why a special form is needed for loops