Control in Sequential Languages

Reading:

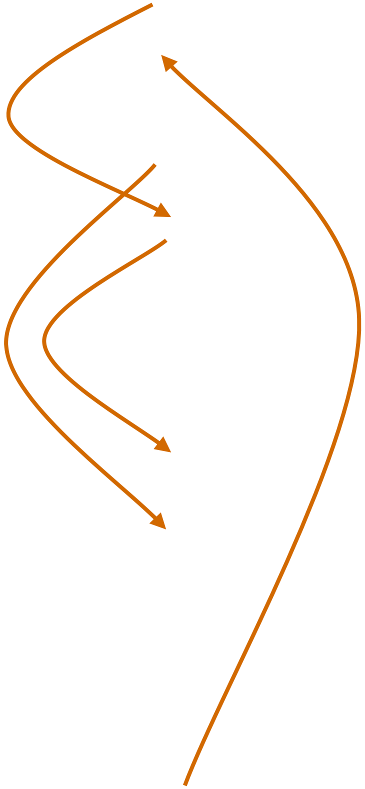
- Chapter 8, Sections 8.1 – 8.3 (only)
- Section 7.3 of The Haskell 98 Report, *Exception Handling in the I/O Monad,* http://www.haskell.org/onlinelibrary/io-13.html (short)
- Chapter 3, Sections 3.3, 3.4.2, 3.4.3, 3.4.5, 3.4.8 (only)

Topics

- Structured Programming
  - Go to considered harmful
- Exceptions
  - “structured” jumps that may return a value
  - dynamic scoping of exception handler
- Continuations
  - Function representing the rest of the program
  - Generalized form of tail recursion
- Heap memory management
  - What is garbage?
  - Standard ways of managing heap memory

Fortran Control Structure

```
10 IF (X .GT. 0.000001) GO TO 20
11 X = -X
   IF (X .LT. 0.000001) GO TO 50
20 IF (X*Y .LT. 0.00001) GO TO 30
    X = X-Y-Y
30  X = X+Y
    ...
50 CONTINUE
    X = A
    Y = B-A
    GO TO 11
    ...
```



Similar structure may occur in assembly code

Historical Debate

- Dijkstra, Go To Statement Considered Harmful
  - Letter to Editor, *C ACM*, March 1968
  - Link on CS242 web site
- Knuth, Structured Prog. with go to Statements
  - You can use goto, but please do so in structured way …
- Continued discussion
  - Welch, "GOTO (Considered Harmful)^n, n is Odd"
- General questions
  - Do syntactic rules force good programming style?
  - Can they help?

Advance in Computer Science

- Standard constructs that structure jumps

  if … then … else … end

  while … do … end

  for … { … }

  case …
- Modern style
  - Group code in logical blocks
  - Avoid explicit jumps except for function return
  - Cannot jump *into* middle of block or function body

Exceptions: Structured Exit

- Terminate part of computation
  - Jump out of construct
  - Pass data as part of jump
  - Return to most recent site set up to handle exception
  - Unnecessary activation records may be deallocated
    - May need to free heap space, other resources
- Two main language constructs
  - Establish exception *handler* to *catch exception*
  - Statement or expression to *raise* or *throw* exception

Often used for unusual or exceptional condition; other uses too

JavaScript Exceptions

```
throw e  //jump to catch, passing exception object

try { …                   //code to try
} catch (e if e == …) { … //catch if first condition true
} catch (e if e == …) { … //catch if second condition true
} catch (e if e == …) { … //catch if third condition true
} catch (e){ …            // catch any exception
} finally { …             //code to execute after everything else
}
```

http://developer.mozilla.org/En/Core_JavaScript_1.5_Guide/
Exception_Handling_Statements

JavaScript Example

```
function invert(matrix) {
    if … throw “Determinant”;
    …
};

try { … invert(myMatrix); …
}
catch (e) { …
    // recover from error
}
```

C++ Example

```
Matrix invert(Matrix m) {
    if … throw Determinant;
    …
};

try { … invert(myMatrix); …
}
catch (Determinant) { …
    // recover from error
}
```

Where is an exception caught?

- Dynamic scoping of handlers
  - Throw to most recent catch on run-time stack
  - Recall: stacks and activation records
    - Which activation record link is used?
      - Access link?   Control link?
- Dynamic scoping is not an accident
  - User knows how to handler error
  - Author of library function does not

ML Exceptions (cover briefly so book is useful to you)

- Declaration

  exception ⟨name⟩ of ⟨type⟩

  gives name of exception and type of data passed when raised

- Raise

  raise ⟨name⟩ ⟨parameters⟩

  expression form to raise and exception and pass data

- Handler

  ⟨exp1⟩ handle ⟨pattern⟩ => ⟨exp2⟩

  evaluate first expression

  if exception that matches pattern is raised,
    then evaluate second expression instead

General form allows multiple patterns.

Exception for Error Condition

```
- datatype ‘a tree = LF of  ‘a | ND of  (‘a tree)*(‘a tree)
- exception No_Subtree;
- fun lsub (LF x) = raise No_Subtree
  |    lsub (ND(x,y)) = x;
> val lsub = fn : ‘a tree -> ‘a tree
```

- This function raises an exception when there is no reasonable value to return
- We’ll look at typing later.

Exception for Efficiency

- Function to multiply values of tree leaves

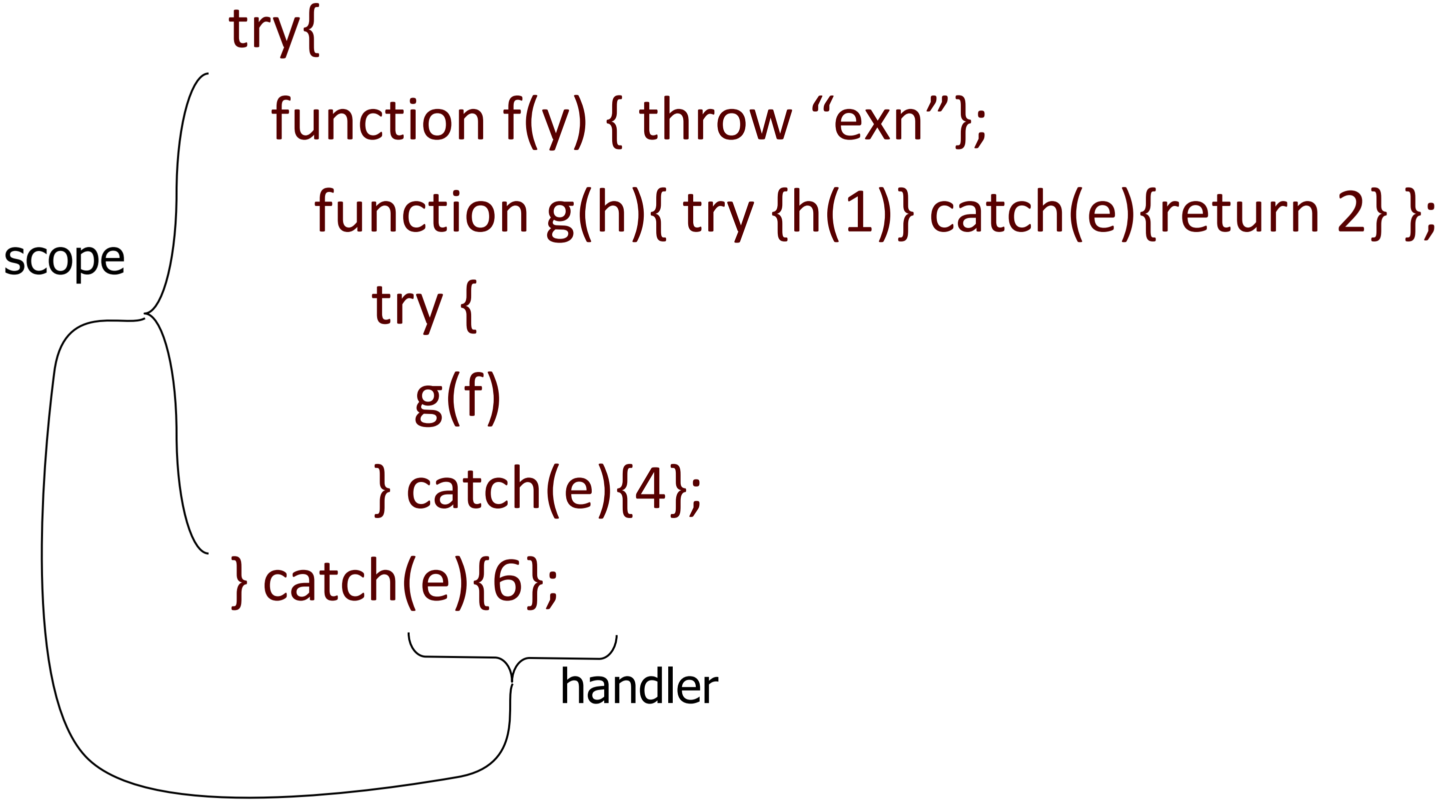
```
fun prod(LF x) = x
|    prod(ND(x,y)) = prod(x) * prod(y);
```

- Optimize using exception

```
fun prod(tree) =
    let exception Zero
        fun p(LF x) = if x=0 then (raise Zero) else x
        |    p(ND(x,y)) = p(x) * p(y)
    in
        p(tree) handle Zero=>0
    end;
```

JavaScript version

Dynamic Scope of Handler

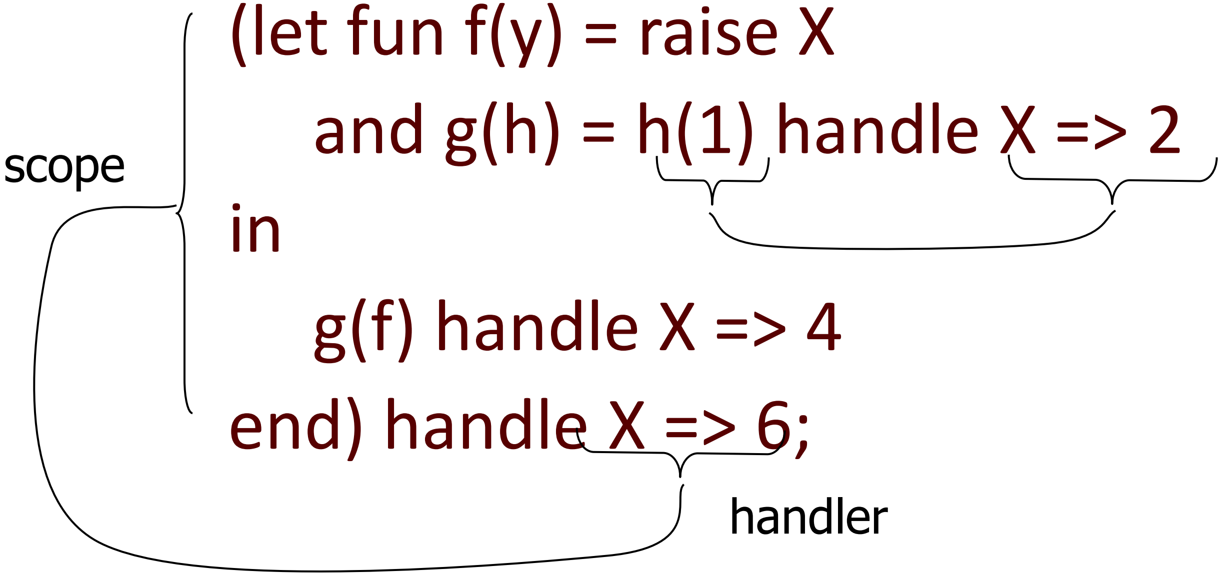


```
try{
  function f(y) { throw “exn”};
    function g(h){ try {h(1)} catch(e){return 2} };
       try {
         g(f)
       } catch(e){4};
} catch(e){6};
```

Which catch catches the throw?

Book version (ML)

Dynamic Scope of Handler



```
exception X;
(let fun f(y) = raise X
    and g(h) = h(1) handle X => 2
in
    g(f) handle X => 4
end) handle X => 6;
```

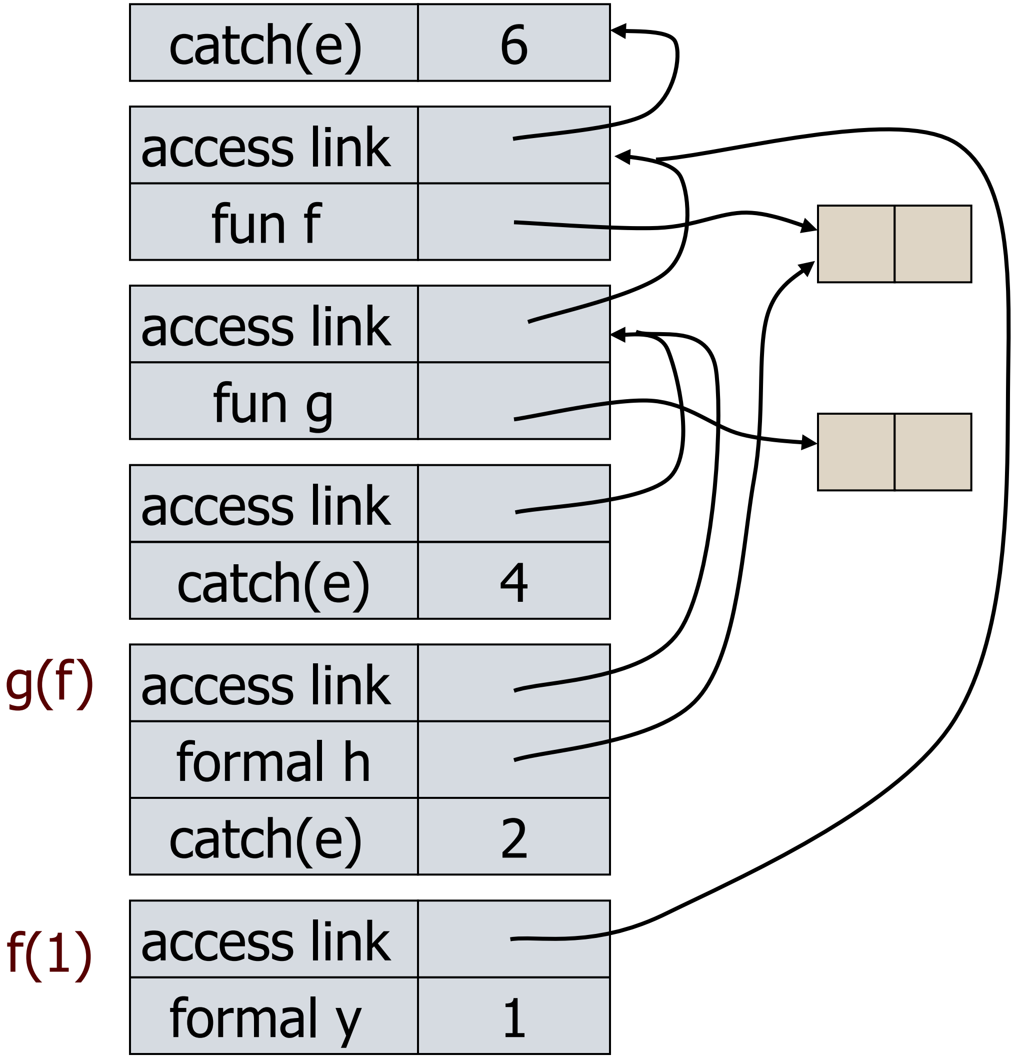
Which handler is used?

JavaScript version

Dynamic Scope of Handler

```
try{
   function f(y) { throw “exn”};
   function g(h){ try {h(1)}
            catch(e){return 2}
   };
   try {
            g(f)
   } catch(e){4};
} catch(e){6};
```

Dynamic scope:
find first handler,
going up the
dynamic call chain

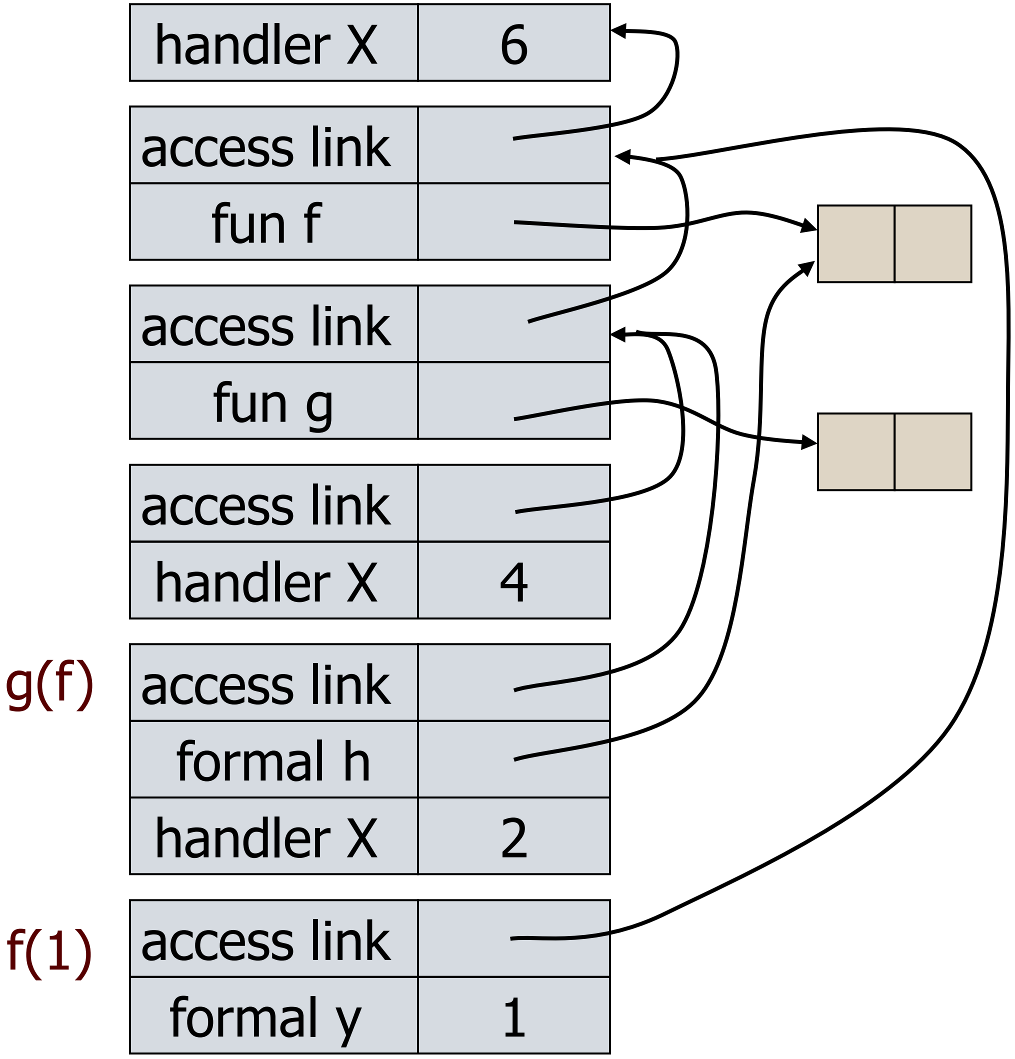


Book version (ML)

Dynamic Scope of Handler

```
exception X;
(let fun f(y) = raise X
    and g(h) = h(1) handle X => 2
in
    g(f) handle X => 4
end) handle X => 6;
```

Dynamic scope:
find first X handler, going up the dynamic call chain leading to raise X.



JavaScript version

Compare to static scope of variables

```
try{
   function f(y) { throw “exn”};
   function g(h){ try {h(1)}
            catch(e){return 2}
   };
   try {
             g(f)
   } catch(e){4};
} catch(e){6};
```

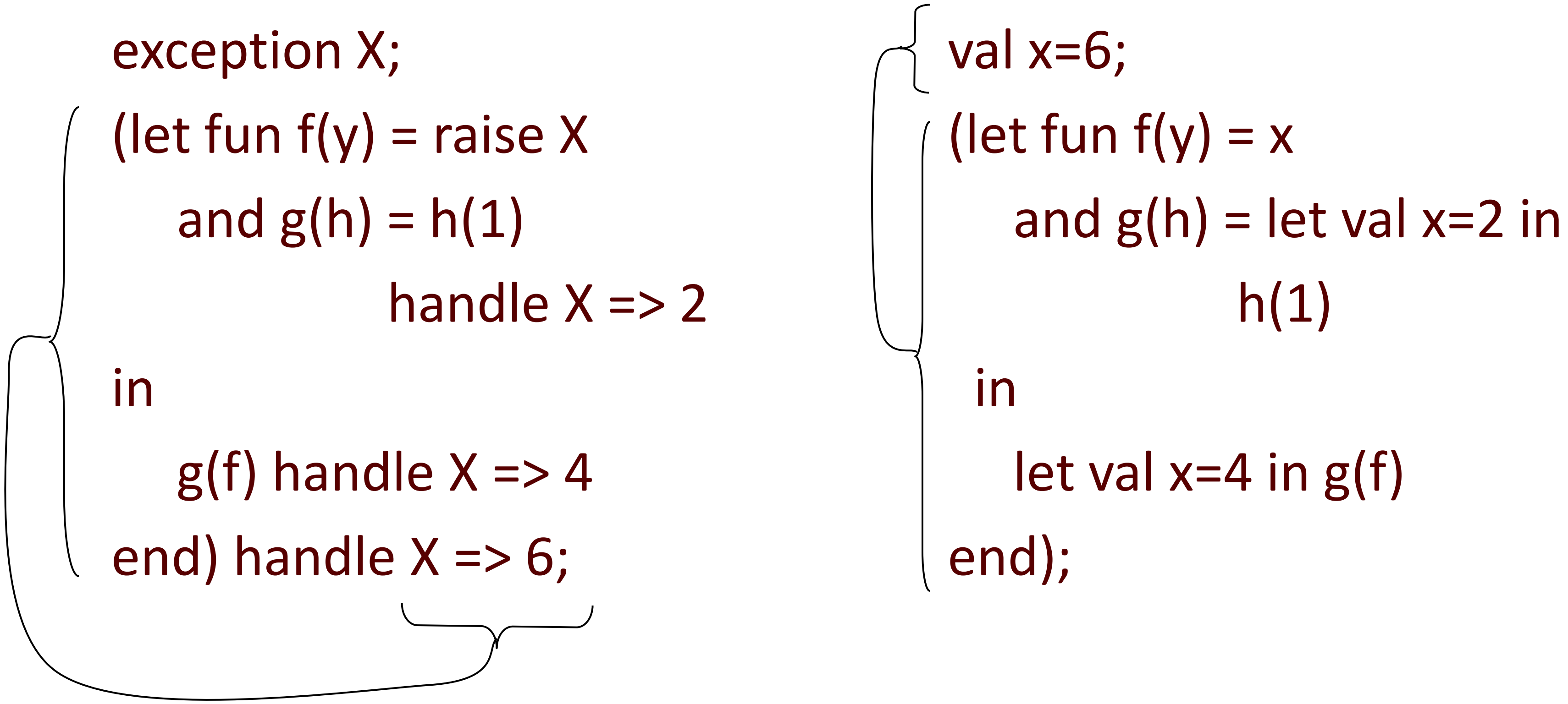
declaration

declaration

```
var x=6;
function f(y) { return x};
function g(h){ var x=2;
         return h(1)
 };
(function (y) {
    var x=4;
    g(f)
 })(0);
```

Book version (ML)

Compare to static scope of variables



```
exception X;
(let fun f(y) = raise X
    and g(h) = h(1)
                 handle X => 2
in
    g(f) handle X => 4
end) handle X => 6;
```

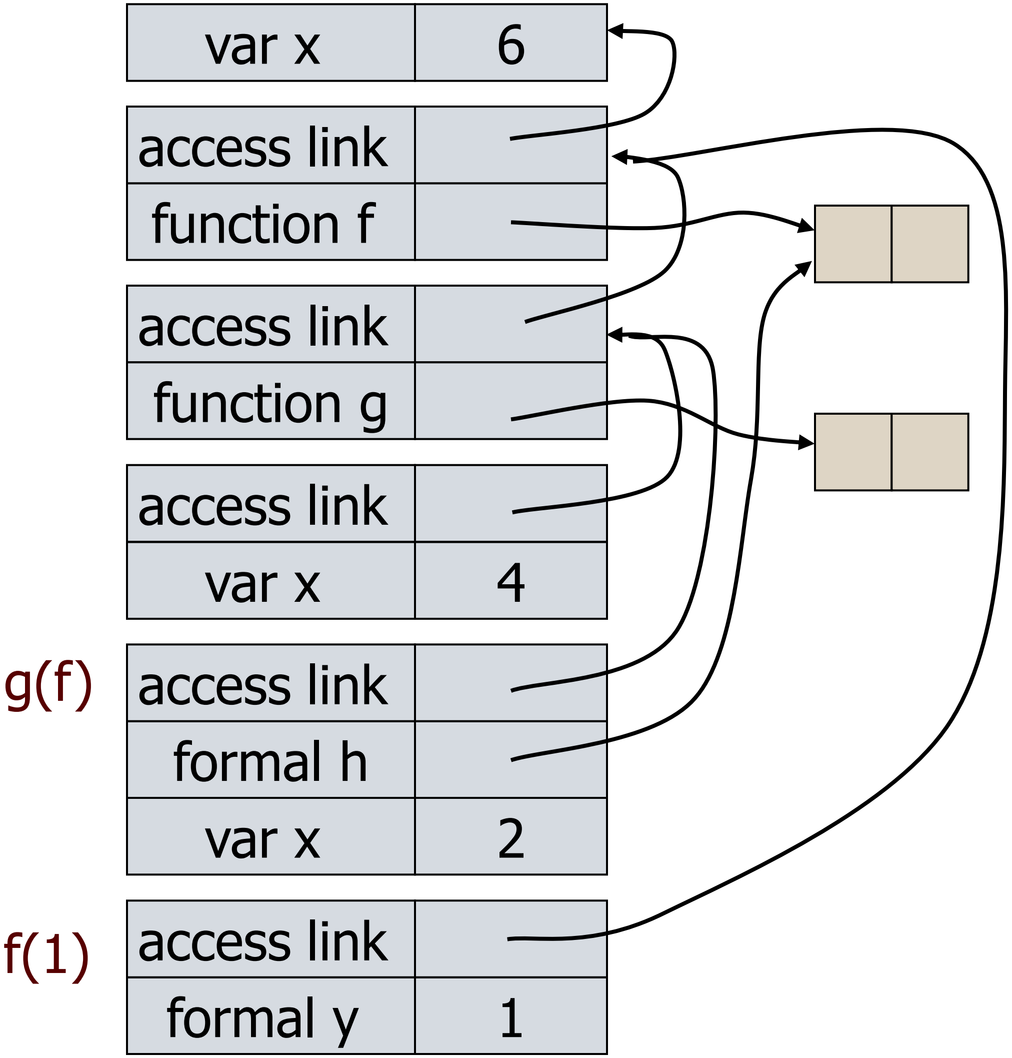
```
val x=6;
(let fun f(y) = x
    and g(h) = let val x=2 in
                 h(1)
  in
    let val x=4 in g(f)
end);
```

JavaScript version

Static Scope of Declarations

```
var x=6;
function f(y) { return x};
function g(h){
       var x=2;  return h(1) };
(function (y) {
      var x=4; g(f)
 })(0);
```

Static scope: find first x, following access links from the reference to X.



var x	6
access link	
function f	
access link	
function g	
access link	
var x	4

g(f)

access link	
formal h	
var x	2

f(1)

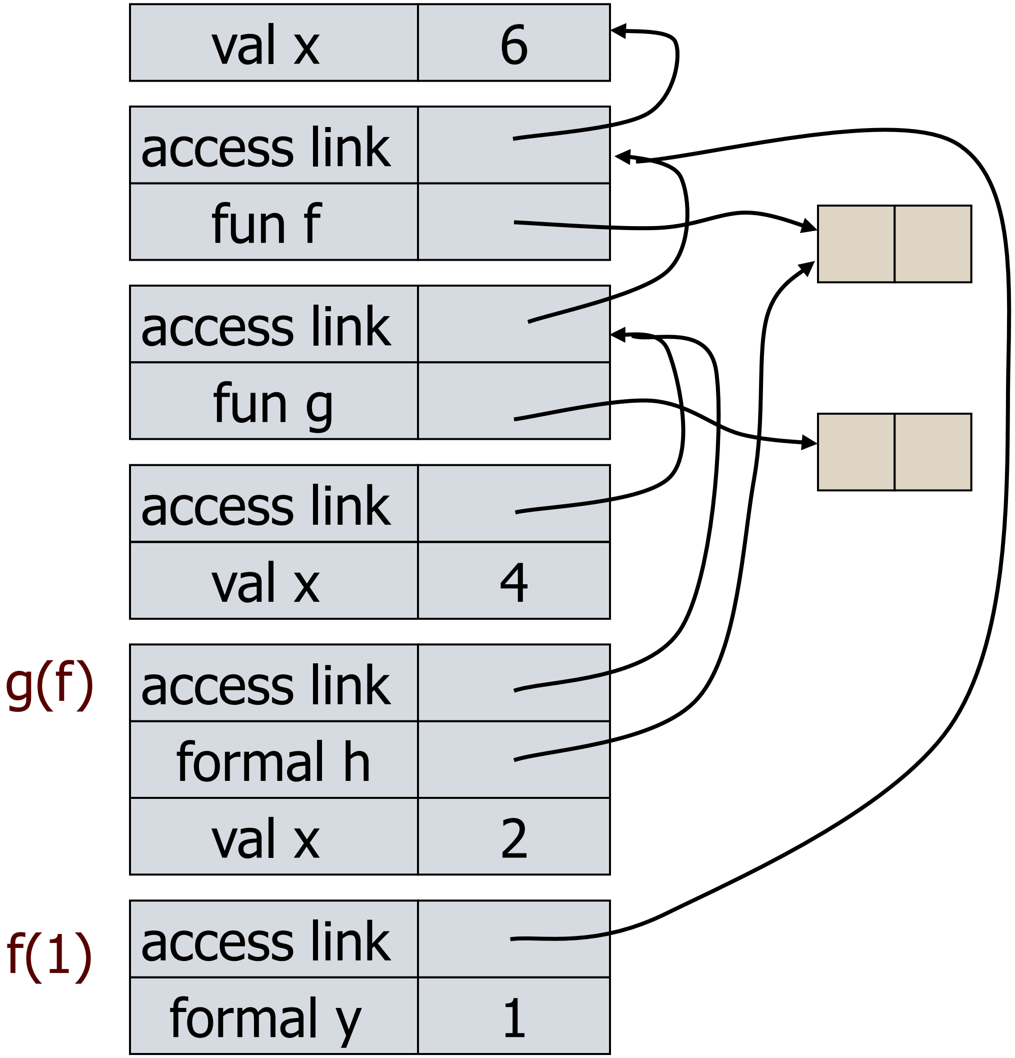
access link	
formal y	1

Book version (ML)

Static Scope of Declarations

```
val x=6;
(let fun f(y) = x
    and g(h) = let val x=2 in
                  h(1)
 in
    let val x=4 in g(f)
end);
```

Static scope: find first x, following access links from the reference to X.



val x	6
access link	
fun f	
access link	
fun g	
access link	
val x	4
access link	
formal h	
val x	2
access link	
formal y	1

Typing of Exceptions (Haskell)

- Special type IOError of exception
  `userError :: String -> IOError`
- Exceptions are raised and caught using
  `ioError :: IOError -> IO a`
  `catch   :: IO a    -> (IOError -> IO a) -> IO a`
- Questions
  - Why is `ioError(userError x)` “any type”?
  - Consider `catch x (\e -> y)` - types must match
- Limitations
  - Propagate by re-raising any unwanted exceptions
  - Only strings are passed (implementation dependent)

ML Typing of Exceptions

- Typing of raise ⟨exn⟩
  - Definition of ML typing

    Expression e has type t if normal termination of e produces value of type t
  - Raising exception is not normal termination

    Example: 1 + raise X
- Typing of handle ⟨exn⟩ => ⟨value⟩
  - Converts exception to normal termination
  - Need type agreement
  - Examples

    1 + ((raise X) handle X => e)   Type of e must be int

    1 + (e_1 handle X => e_2)   Type of $e_{1,}$ e_2 must be int

Exceptions and Resource Allocation

- Resources may be allocated inside try block
- May be "garbage" after exception
- Examples
  - Memory (problem in C/C++)
  - Lock on database
  - Threads
  - …

General problem: no obvious solution

Continuations

- Idea:
  - The continuation of an expression is “the remaining work to be done after evaluating the expression”
  - Continuation of *e* is a function normally applied to *e*
- General programming technique
  - Capture the continuation at some point in a program
  - Use it later: “jump” or “exit” by function call
- Useful in
  - Compiler optimization: make control flow explicit
  - Operating system scheduling, multiprogramming
  - Web site design, other applications

JavaScript version

Example of Continuation Concept

- Expression
  - 2*x + 3*y + 1/x + 2/y
- What is continuation of 1/x?
  - Remaining computation after division

```
var before = 2*x + 3*y;
function cont(d) {return (before + d + 2/y)};
cont (1/x);
```

Book version (ML)

Example of Continuation Concept

- Expression
  - 2*x + 3*y + 1/x + 2/y
- What is continuation of 1/x?
  - Remaining computation after division

```
let val before = 2*x + 3*y
    fun continue(d) = before + d + 2/y
in
    continue (1/x)
end
```

Example: Tail Recursive Factorial

- Standard recursive function

  fact(n) = if n=0 then 1 else n*fact(n-1)

- Tail recursive

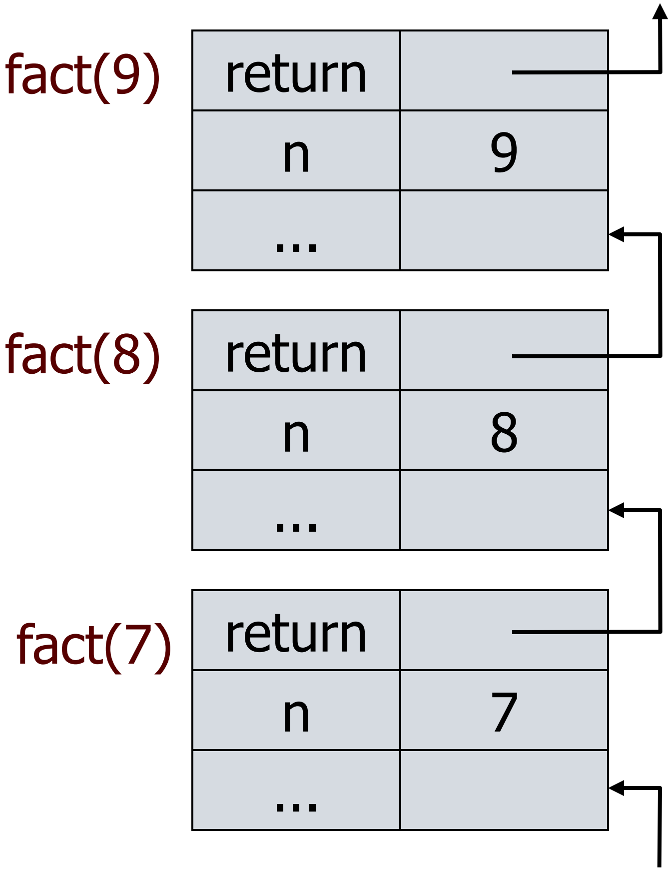
  f(n,k) = if n=0 then k else f(n-1, n*k)

  fact(n) = f(n,1)

- How could we derive this?
  - Transform to continuation-passing form
  - Optimize continuation function to single integer

Continuation view of factorial

fact(n) = if n=0 then 1 else n*fact(n-1)



This invocation multiplies by 9 and returns
Continuation of fact(8) is
λx. 9*x

Multiplies by 8 and returns
Continuation of fact(7) is
λy. (λx. 9*x) (8*y)

Multiplies by 7 and returns
Continuation of fact(6) is
λz. (λy. (λx. 9*x) (8*y)) (7*z)

Derivation of tail recursive form

- Standard function

$$\text{fact}(n) = \text{if } n=0 \text{ then } 1 \text{ else } n*\text{fact}(n-1)$$

- Continuation form

$$\begin{aligned}\text{fact}(n, k) = {} & \text{if } n=0 \text{ then } k(1)\\ & \text{else fact}(n-1, \overbrace{\lambda x.k\,(n*x)}^{\text{continuation}}\,)\end{aligned}$$

$$\text{fact}(n, \lambda x.x) \ \text{ computes } n!$$

- Example computation

$$\begin{aligned}\text{fact}(3,\lambda x.x) & = \text{fact}(2, \lambda y.((\lambda x.x)\,(3*y)))\\ & = \text{fact}(1, \lambda x.((\lambda y.3*y)(2*x)))\\ & = \lambda x.((\lambda y.3*y)(2*x))\ 1 = 6\end{aligned}$$

Tail Recursive Form

- Optimization of continuations

  fact(n,a) = if n=0 then a
  else fact(n-1, n*a)

  Each continuation is effectively $\lambda x.(a*x)$ for some a

- Example computation

  fact(3,1) = fact(2, 3) was fact(2, $\lambda y.3*y$)
  = fact(1, 6) was fact(1, $\lambda x.6*x$)
  = 6

Other uses for continuations

- Explicit control
  - Normal termination -- call continuation
  - Abnormal termination -- do something else
- Compilation techniques
  - Call to continuation is functional form of “go to”
  - Continuation-passing style makes control flow explicit

MacQueen: “Callcc is the closest thing to a
‘come-from’ statement I’ve ever seen.”

Continuations in Mach OS

- OS kernel schedules multiple threads
  - Each thread may have a separate stack
  - Stack of blocked thread is stored within the kernel
- Mach “continuation” approach
  - Blocked thread represented as
    - Pointer to a continuation function, list of arguments
    - Stack is discarded when thread blocks
  - Programming implications
    - Sys call such as msg_recv can block
    - Kernel code calls msg_recv with continuation passed as arg
  - Advantage/Disadvantage
    - Saves a lot of space, need to write “continuation” functions

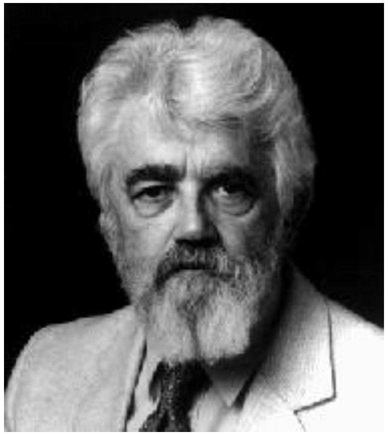
Continuations in compilation

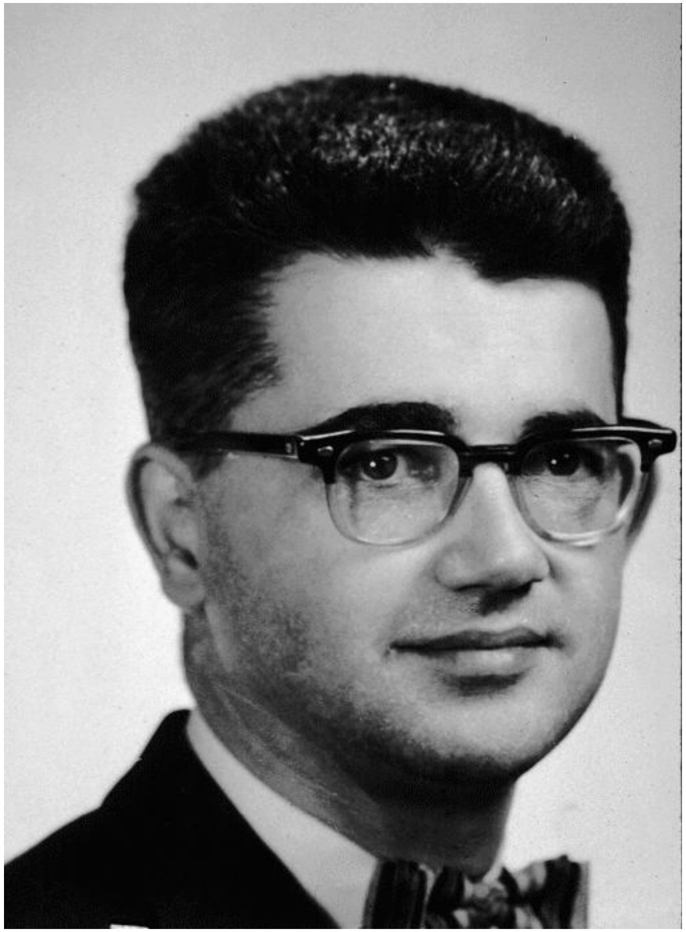
- SML continuation-based compiler [Appel, Steele]
  1) Lexical analysis, parsing, type checking
  2) Translation to λ-calculus form
  3) Conversion to continuation-passing style (CPS)
  4) Optimization of CPS
  5) Closure conversion – eliminate free variables
  6) Elimination of nested scopes
  7) Register spilling – no expression with >n free vars
  8) Generation of target assembly language program
  9) Assembly to produce target-machine program

Summary

- Structured Programming
  - Go to considered harmful
- Exceptions
  - “structured” jumps that may return a value
  - dynamic scoping of exception handler
- Continuations
  - Function representing the rest of the program
  - Generalized form of tail recursion
  - Used in Lisp/Scheme compilation, some OS projects, web application development, …
- Heap memory management
  - What is garbage?
  - Standard ways of managing heap memory

Lisp: John McCarthy





- Pioneer in AI
  - Formalize common-sense reasoning
- Also
  - Proposed timesharing
  - Mathematical theory
  - ….
- Lisp

  stems from interest in symbolic computation (math, logic)

Lisp summary

- Many different dialects
  - Lisp 1.5, Maclisp, …, Scheme, ...
  - CommonLisp has many additional features
  - This course: a fragment of Lisp 1.5, approximately

    But ignore static/dynamic scope until later in course
- Simple syntax

```
(+ 1 2 3)
(+ (* 2 3) (* 4 5))
(f x y)
```

Easy to parse (Looking ahead: programs as data)

Atoms and Pairs

- Atoms include numbers, indivisible “strings”

  <atom> ::= <smbl> | <number>

  <smbl> ::= <char> | <smbl><char> | <smbl><digit>

  <num> ::= <digit> | <num><digit>

- Dotted pairs
  - Write (A . B) for pair
  - Symbolic expressions, called *S-expressions*:

  <sexp> ::= <atom> | (<sexp> . <sexp>)

Note on syntax

Book uses some kind of pidgin Lisp

In Scheme, a pair prints as (A . B), but (A . B) is not an expression

Basic Functions

- Functions on atoms and pairs:

  cons car cdr eq atom

- Declarations and control:

  cond lambda define eval quote

- Example

  (lambda (x) (cond ((atom x) x) (T (cons ‘A x))))

  function f(x) = if atom(x) then x else cons(“A”,x)

- Functions with side-effects

  rplaca rplacd

Evaluation of Expressions

- Read-eval-print loop
- Function call (function arg_1 … arg_n)
  - evaluate each of the arguments
  - pass list of argument values to function
- Special forms do not eval all arguments
  - Example (cond (p_1 e_1) … (p_n e_n))
    - proceed from left to right
    - find the first p_i with value true, eval this e_i
  - Example (quote A) does not evaluate A

Examples

`(+ 4 5)`

expression with value 9

`(+ (+ 1 2) (+ 4 5))`

evaluate 1+2, then 4+5, then 3+9 to get value

`(cons (quote A) (quote B))`

pair of atoms A and B

`(quote (+ 1 2))`

evaluates to list (+ 1 2)

`'(+ 1 2)`

same as (quote (+ 1 2))

Conditional Expressions in Lisp

- Generalized if-then-else

  (cond $(p_1\ e_1)$ $(p_2\ e_2)$ … $(p_n\ e_n)$)

  - Evaluate conditions p_1 … p_n left to right
  - If p_i is first condition true, then evaluate e_i
  - Value of e_i is value of expression

  No value for the expression if no p_i true, or

  p_1 … p_i false and p_{i+1} has no value, or

  relevant p_i true and e_i has no value

Examples

(cond ((< 2 1) 2) ((< 1 2) 1))

has value 1

(cond ((< 2 1) 2) ((< 3 2) 3))

has no value

(cond (*diverge* 1) (true 0))

no value, if *expression* diverge loops forever

(cond (true 0) (*diverge* 1))

has value 0

Function Expressions

- Form

  ```
  (lambda ( parameters )  ( function_body ) )
  ```

- Syntax comes from lambda calculus:

  λf. λx. f (f x)

  ```
  (lambda (f) (lambda (x) (f  (f  x))))
  ```

- Defines a function but does not give it a name

  ```
  ( (lambda (f) (lambda (x) (f  (f  x))))
     (lambda (x) (+ 1 x)))
  )
  ```

Example

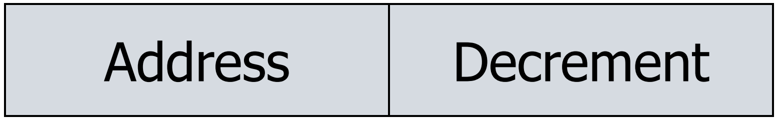
```
(define twice
     (lambda (f) (lambda (x) (f  (f  x))))
)

(define inc (lambda (x) (+ 1 x)))

((twice inc) 2)
⇒  4
```

Lisp Memory Model

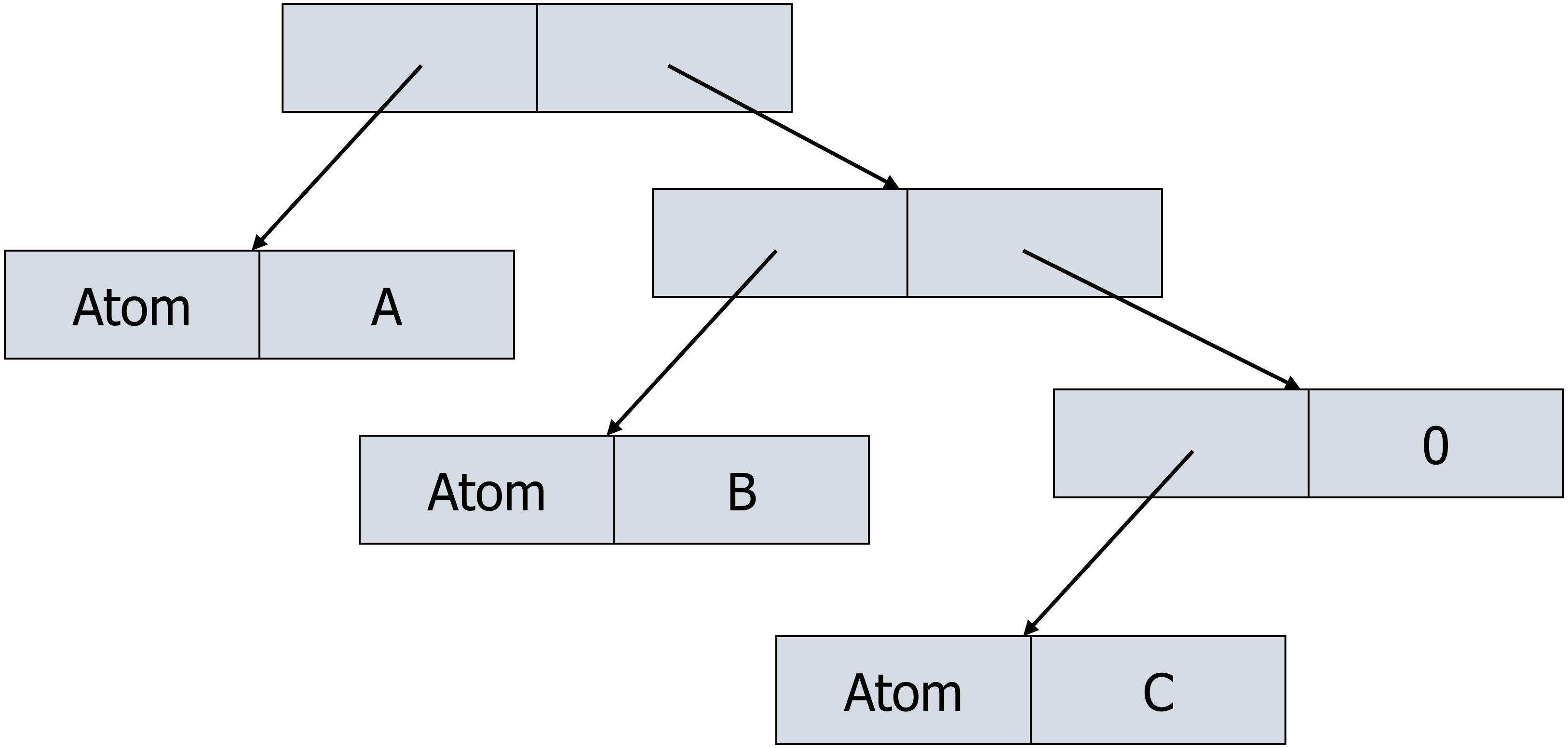
- Cons cells



Address

Decrement

- Atoms and lists represented by cells



Atom

A

Atom

B

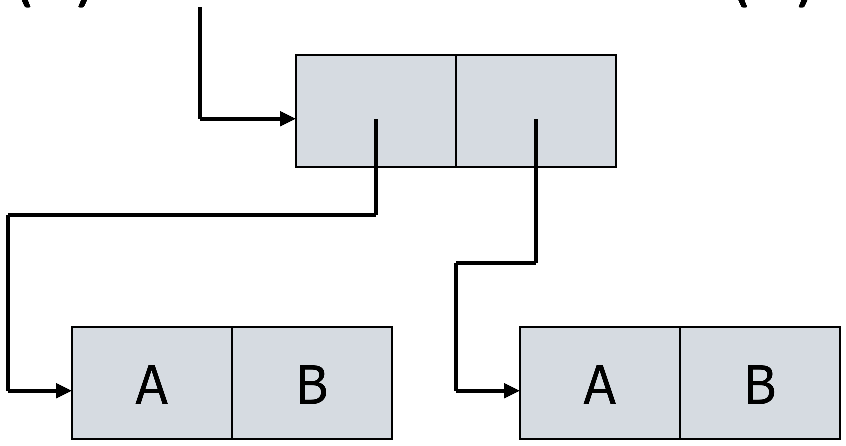
0

Atom

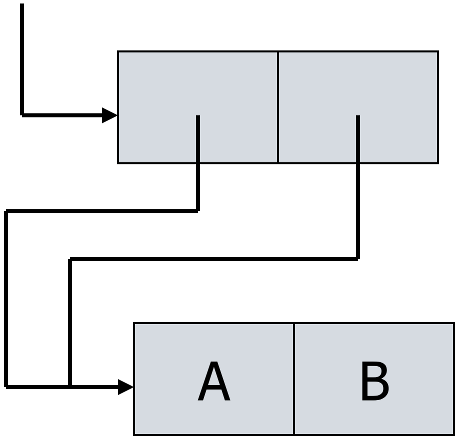
C

Sharing

(a)



(b)



- Both structures could be printed as ((A.B) . (A.B))
- Which is result of evaluating

(cons (cons 'A 'B) (cons 'A 'B)) ?

Note: Scheme actually prints using combination of list and dotted pairs

Garbage Collection

- Garbage:

  At a given point in the execution of a program **`P`**, a memory location **`m`** is *garbage* if no continued execution of **`P`** from this point can access location **`m`**.

- Garbage Collection:
  - Detect garbage during program execution
  - GC invoked when more memory is needed
  - Decision made by run-time system, not program

Examples

(car (cons (e_1) (e_2)))

Cells created in evaluation of e_2 may be garbage, unless shared by e_1 or other parts of program

((lambda (x) (car (cons (... x...) (... x ...)))
'(Big Mess))

The car and cdr of this cons cell may point to overlapping structures.

Mark-and-Sweep Algorithm

- Assume tag bits associated with data
- Need list of heap locations named by program
- Algorithm:
  - Set all tag bits to 0.
  - Start from each location used directly in the program. Follow all links, changing tag bit to 1
  - Place all cells with tag = 0 on free list

Why Garbage Collection in Lisp?

- McCarthy's paper says this is
  - “… more convenient for the programmer than a system in which he has to keep track of and erase unwanted lists."
- Does this reasoning apply equally well to C?
- Is garbage collection "more appropriate" for Lisp than C? Why?

Summary

- Structured Programming
  - Go to considered harmful
- Exceptions
  - “structured” jumps that may return a value
  - dynamic scoping of exception handler
- Continuations
  - Function representing the rest of the program
  - Generalized form of tail recursion
  - Used in Lisp/Scheme compilation, some OS projects, web application development, …
- Heap memory management
  - Definition of garbage
  - Mark-and-sweep garbage collection algorithm