Control in Sequential Languages

Mooly Sagiv

Original slides by John Mitchell

Reading Concepts in Programming Language

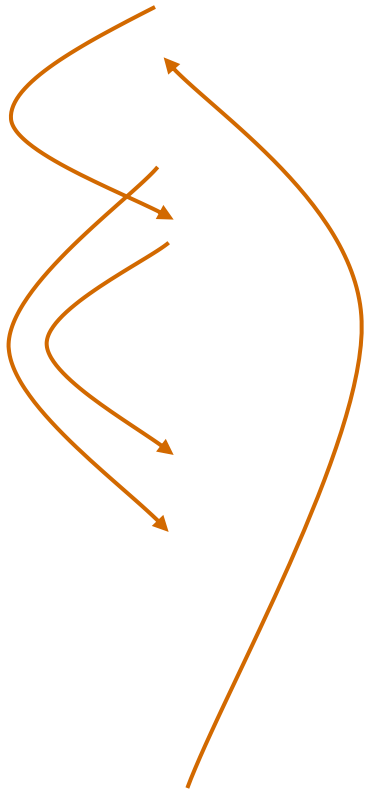
- Chapter 8, Sections 8.1 – 8.3 (only)
- Chapter 3, Sections 3.3, 3.4.2, 3.4.3, 3.4.5, 3.4.8 (only)

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

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

Non-Local goto in C syntax

```
void level_0(void) {
    void level_1(void) {
        void level_2(void) {
            ...
            goto L_1;
            ...
        }
        ...
    L_1:...
        ...
    }
    ...
}
```

Non-local gotos in C

- setjmp remembers the current location and the stack frame
- longjmp jumps to the current location (popping many activation records)

Non-Local Transfer of Control in C

```
#include <setjmp.h>

void find_div_7(int n, jmp_buf *jmpbuf_ptr) {
    if (n % 7 == 0) longjmp(*jmpbuf_ptr, n);
    find_div_7(n + 1, jmpbuf_ptr);
}

int main(void) {
    jmp_buf jmpbuf;          /* type defined in setjmp.h */
    int return_value;

    if ((return_value = setjmp(jmpbuf)) == 0) {
        /* setting up the label for longjmp() lands here */
        find_div_7(1, &jmpbuf);
    }
    else {
        /* returning from a call of longjmp() lands here */
        printf("Answer = %d\n", return_value);
    }
    return 0;
}
```

Historical Debate

- Dijkstra, goto Statement Considered Harmful
  - Letter to Editor, *CACM*, March 1968
- Knuth, Structured Prog. with goto 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
}
```

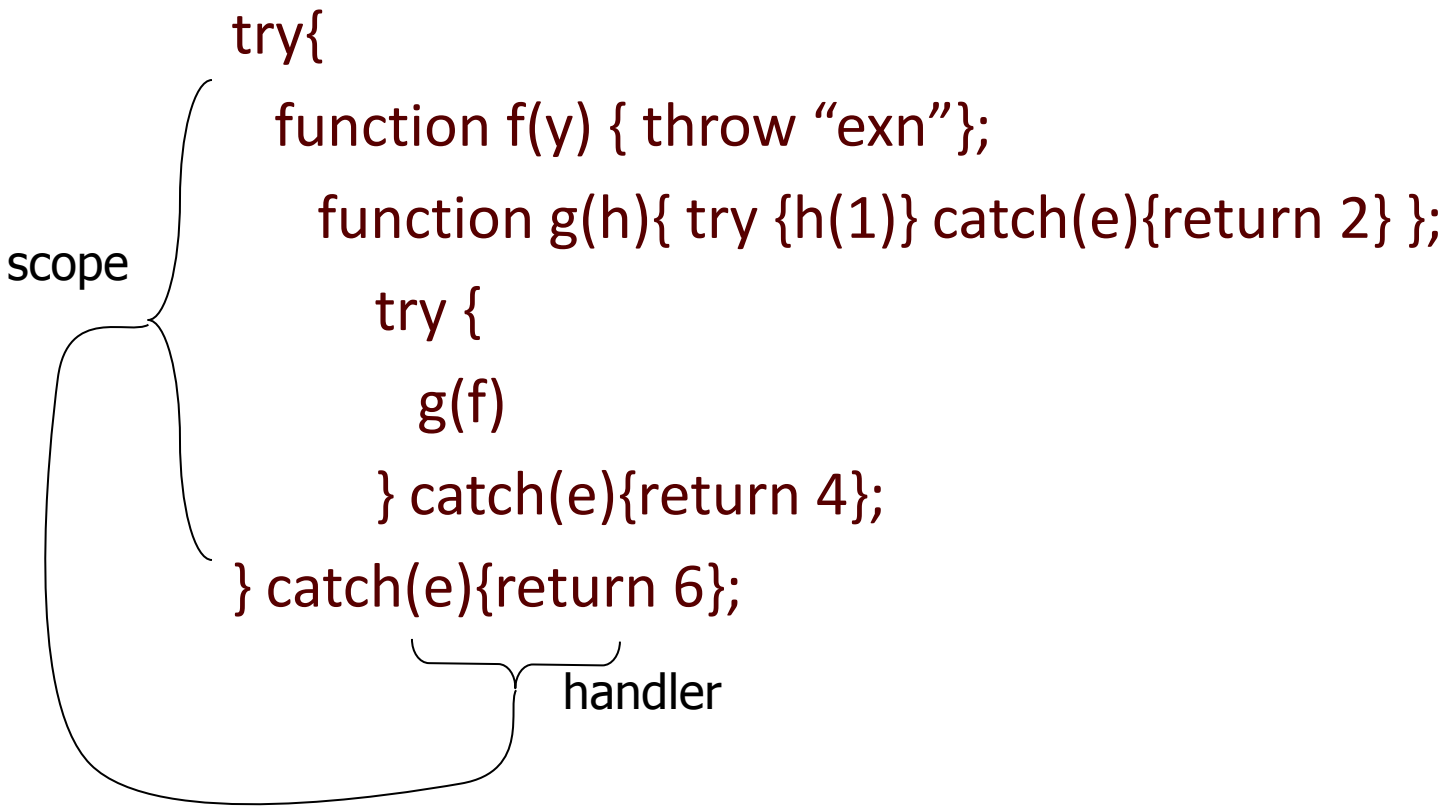
JavaScript Example

```
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
}
```

Dynamic Scope of Handler

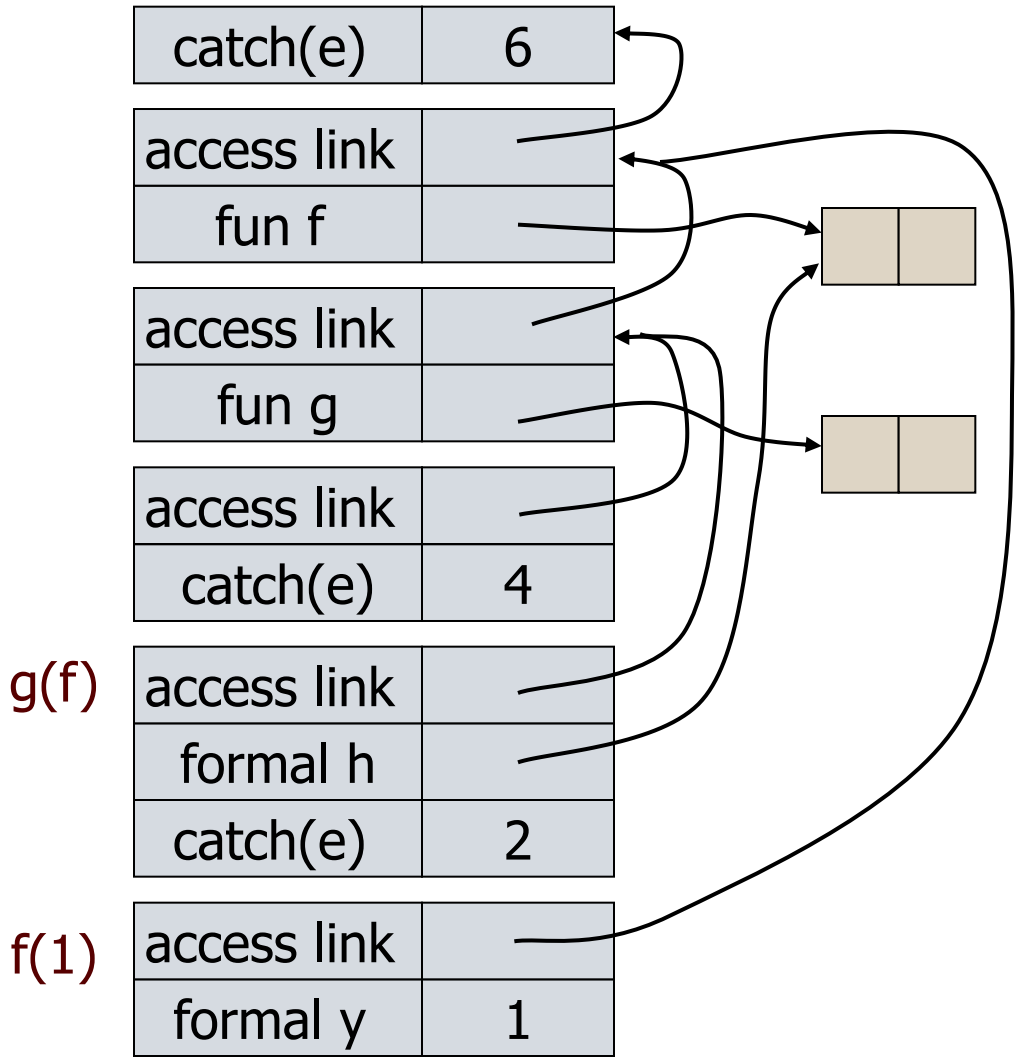


Which catch catches the throw?

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){return 4};
} catch(e){return 6};
```

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



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

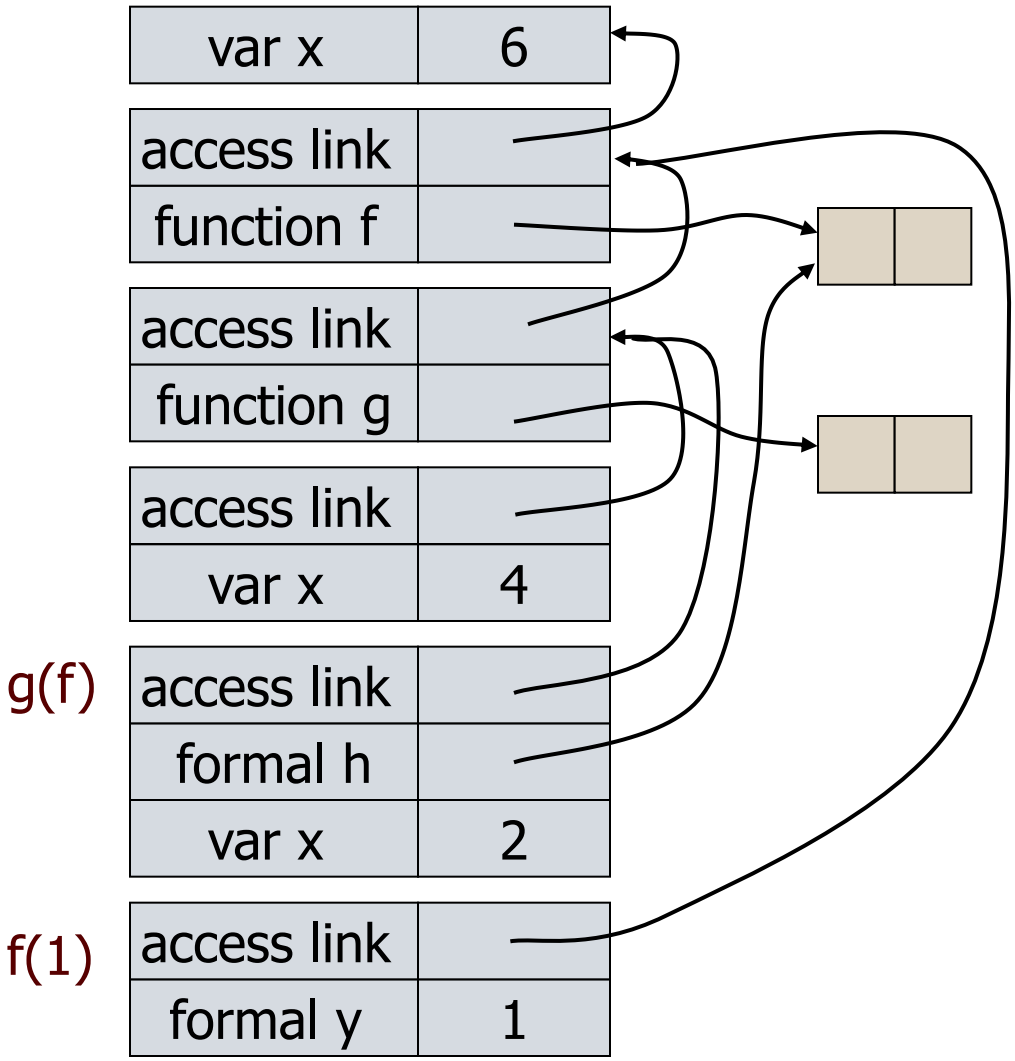
```
exception X;
(let fun f(y) = raise X
    and g(h) = h(1)
                handle X => 2
in
    g(f) handle X => 4
end) handle X => 6;
```

```
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);
```

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);
```

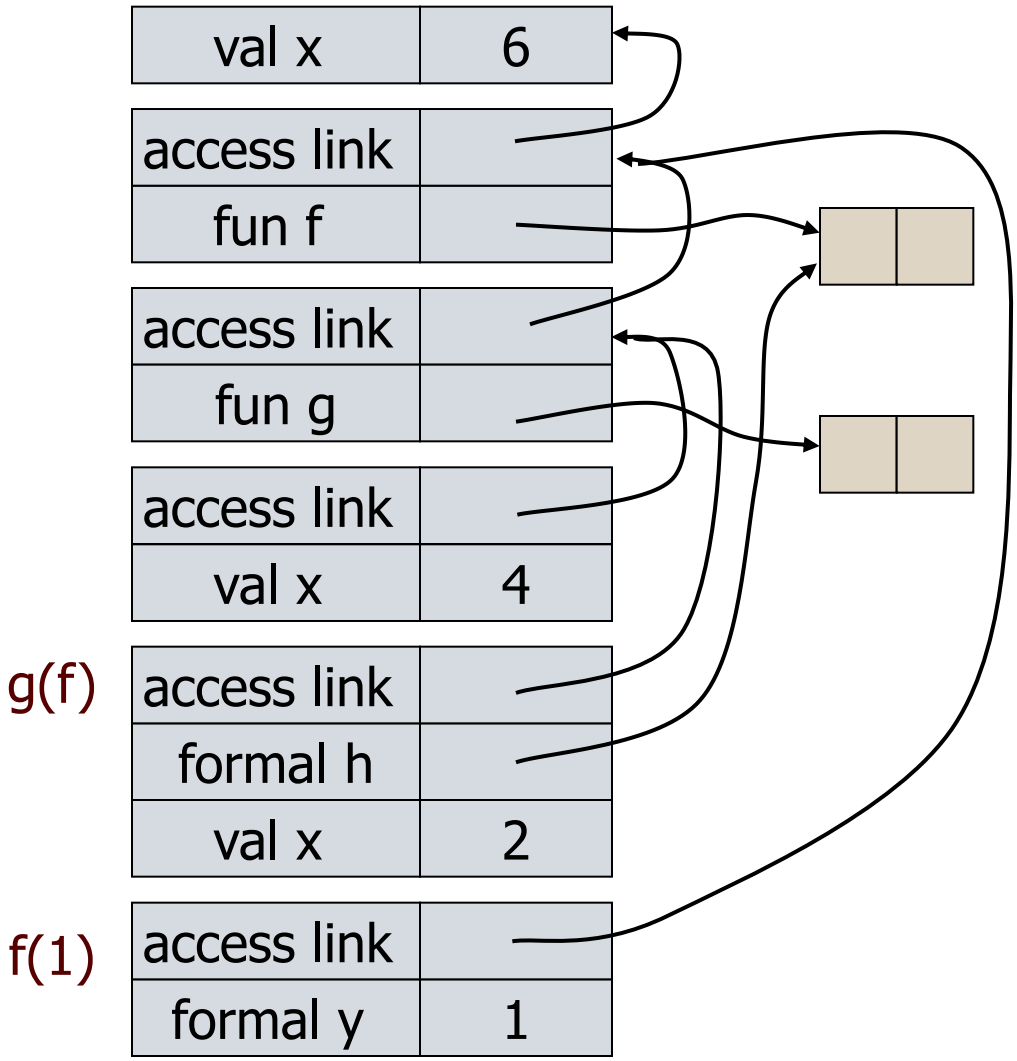
Static scope: find first x, following access links from the reference to X.



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.



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

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
  - Denotational Semantics
  - Compiler optimization: make control flow explicit
  - Operating system scheduling, multiprogramming
  - Web site design, other applications

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);
```

Example: Tail Recursive Factorial

- Standard recursive function

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

- Tail recursive

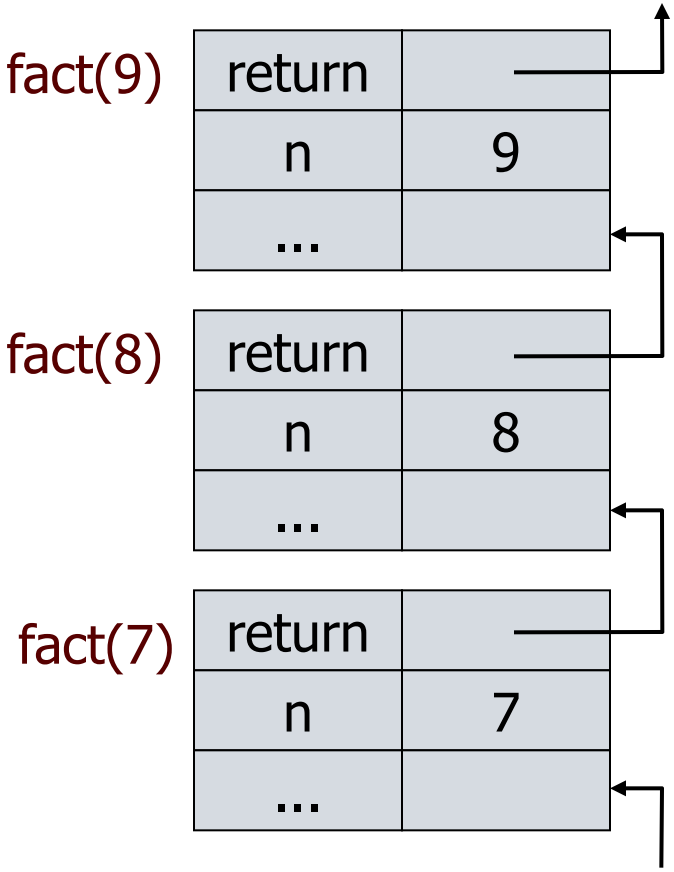
  f(n,k) = if n=0 then k else f(n-1, n*k)

  fact(n) = f(n,1)

- How could we derive this?
  - Transform to continuation-passing form (automatic)
  - 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
$\lambda x.\ 9*x$

Multiplies by 8 and returns
Continuation of fact(7) is
$\lambda y.\ (\lambda x.\ 9*x)\ (8*y)$

Multiplies by 7 and returns
Continuation of fact(6) is
$\lambda z.\ \ (\lambda y.\ (\lambda x.\ 9*x)\ (8*y))\ (7*z)$

continuation fact(n-1) = continuation fact(n) $\circ\ \lambda x.\ n*x$

Derivation of tail recursive form

- Standard function

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

- Continuation form

  fact(n, k) = if n=0 then k(1)

  else fact(n-1, λx.k (n*x))

  continuation

  fact(n, λx.x) computes n!

- Example computation

  fact(3,λx.x) = fact(2, λy.((λx.x) (3*y)))

  = fact(1, λx.((λy.3*y)(2*x)))

  = λx.((λy.3*y)(2*x)) 1 = 6

Tail Recursive Form

- Optimization of continuations

  fact(n, k) = if n=0 then k(1)
  else fact(n-1, λx.k (n*x))

  ⇓

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

  Each continuation is effectively λx.(a*x) for some a

- Example computation

  fact(3,1) = fact(2, 3) was fact(2, λy.3*y)

  = fact(1, 6) was fact(1, λx.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 “goto”
  - 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 [SOSP'91]

- 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 Web programming

- Use continuation-passing style to allow multiple returns

```
function doXHR(url, succeed, fail) {
    var xhr = new XMLHttpRequest(); // or ActiveX equivalent
    xhr.open("GET", url, true);
    xhr.send(null);
    xhr.onreadystatechange = function() {
        if (xhr.readyState == 4) {
            if (xhr.status == 200)
                succeed(xhr.responseText);
            else
                fail(xhr);
} }; }
```

- See http://marijn.haverbeke.nl/cps/

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

Theme Song: Charlie on the MTA

- Let me tell you the story
  Of a man named Charlie
  On a tragic and fateful day
  He put ten cents in his pocket,
  Kissed his wife and family
  Went to ride on the MTA
- Charlie handed in his dime
  At the Kendall Square Station
  And he changed for Jamaica Plain
  When he got there the conductor told him,
  "One more nickel."
  Charlie could not get off that train.
- Chorus:

  Did he ever return,
  No he never returned
  And his fate is still unlearn'd
  He may ride forever
  'neath the streets of Boston
  He's the man who never returned.

Summary

- Structured Programming
  - Goto 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, …