UNIVERSITETET
I OSLO

Control in Sequential Languages

Gerardo Schneider
gerardo@ifi.uio.no

Department of Informatics – University of Oslo

Based on John C. Mitchell's slides (Stanford U.)
(with contributions by Arild Torjusen)

INF 3110 – 2008

ML lectures

1. 08.09: The Algol Family and ML (Mitchell's chap. 5 + more)
2. 15.09: More on ML & types (chap. 5, 6, more)
3. 06.10: More on Types: Type Inference and Polymorphism (chap. 6)
4. **13.10: Control in sequential languages, Exceptions and Continuations (chap. 8)**

INF 3110 - 2008

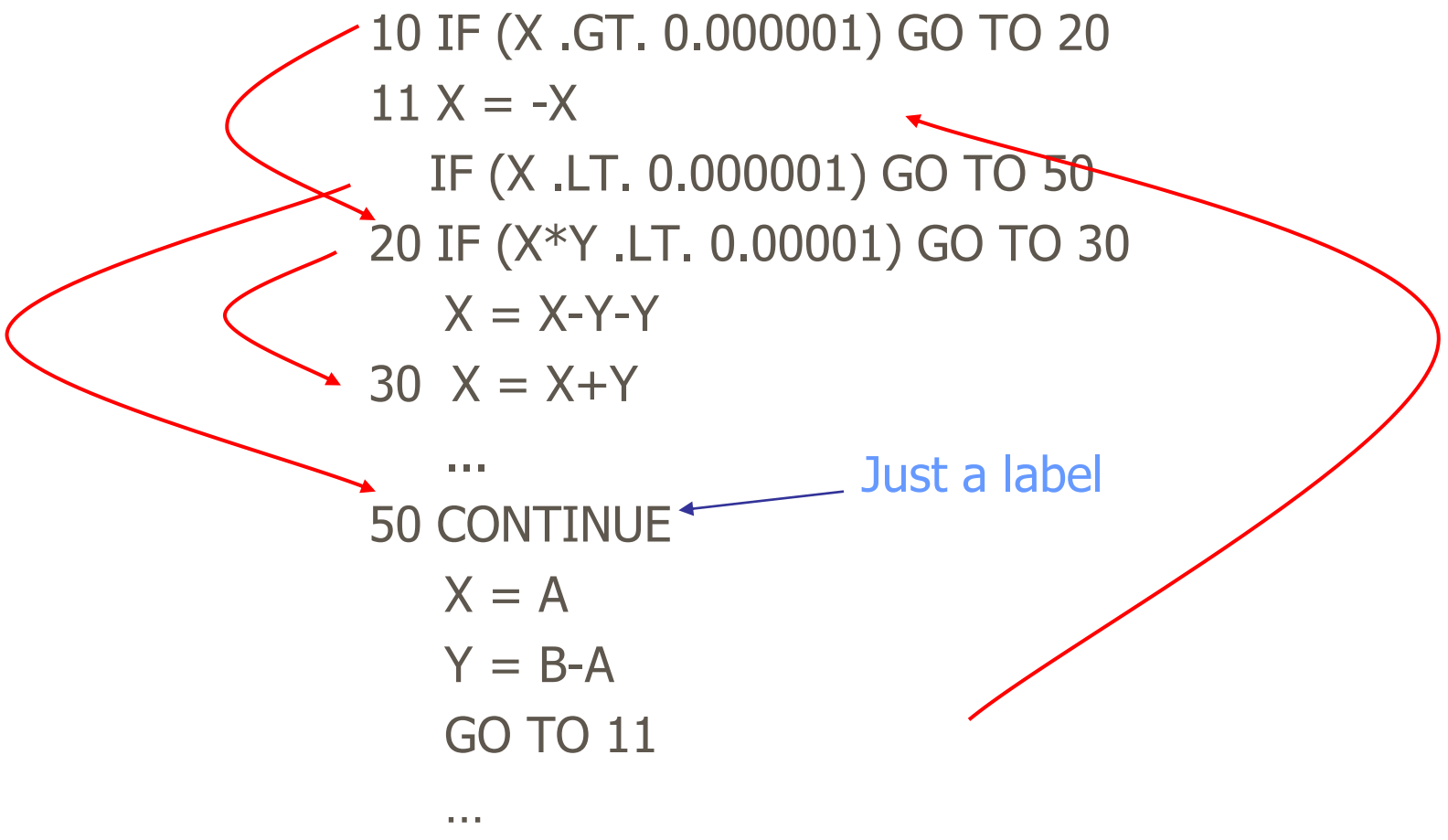
Outline

- Structured Programming
  - *go to* considered harmful
- Exceptions
  - "Structured" jumps that may return a value
  - Dynamic scoping of exception handler
- Continuations
- Evaluation order

Control flow in sequential programs

- The execution of a (sequential) program is done by following a certain control flow
- The end-of-line (or semi-colon) terminates a statement
- What is the next instruction to be executed?
  - The flow of control goes top-down in general
  - Jumps (loops, conditionals, etc)
- It is not easy, in general to ”see” whether a given instruction is reachable from another (Program Analysis)

Fortran Control Structure



Similar structure may occur in assembly code

Historical Debate

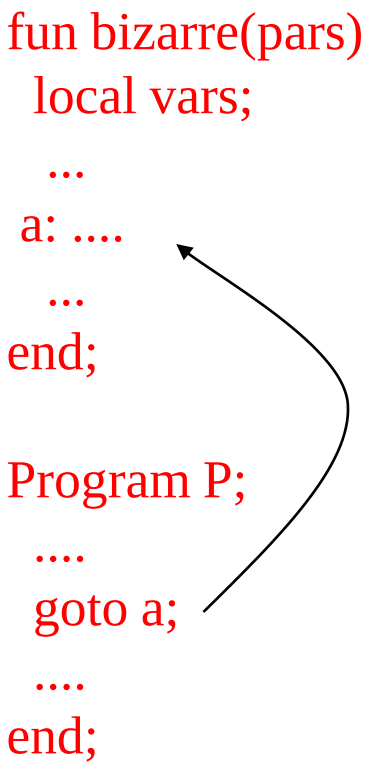
- Dijkstra: "Go To Statement Considered Harmful" (1968)
  - "… the **go to** statement should be abolished from all 'higher level' programming languages…"
- Knuth: "Structured Programming with go to Statements" (1974)
  - You can use goto, but do so in structured way …
- General questions
  - Do syntactic 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 and continuations (?!)

Jumps into Blocks – Why not?

- Label in the body of a function
- Should an activation record be created?
- If not, what about local variables?
  - They are meaningless
- If so, how to set function parameters?
  - There are no parameter values



No clear answers! Better to reject these programs!

Outline

- Structured Programming
  - *go to* considered harmful
- Exceptions
  - "Structured" jumps that may return a value
  - Dynamic scoping of exception handler
- Continuations
- Evaluation order

Exceptions: Structured Exit

- Terminate part of computation
  - Jump *out* of construct, not *into* some part of the program.
  - Pass data as part of jump
  - Return to most recent site set up to handle exception
- Memory management needed
  - Unnecessary activation records may be deallocated
- Two main language constructs
  - Statement or expression to ***raise*** exception (*throw, Java*)
  - Exception ***handler*** (*catch,* Java*)*
- Possible to have more than one handler

Often used for unusual or exceptional condition, but not necessarily

ML Example

```
exception Determinant;  (* declare exception name *)
fun invert (M) =              (* function to invert matrix *)
        ...
        if isZero(Det )
          then raise Determinant  (* exit if Det is zero*)
              else ...
 end;
...
invert (myMatrix) handle Determinant => ... ;
```

Value for expression if determinant of myMatrix is zero

ML Exceptions

- Exceptions are a different kind of entity than types
- Declare exceptions before use
- Exceptions are dynamically scoped
  - Control jumps to the handler most recently established (run-time stack) (more later…)
  - ML is otherwise statically scoped
- Pattern matching is used to determine the appropriate handler (C++/Java uses type matching)

ML Exceptions

- Declaration
  exception ⟨name⟩ of ⟨type⟩
    gives name of exception and type of data passed when raised
    - exception Overflow;
    - exception Signal of int;
- Raise
  raise ⟨name⟩ ⟨parameters⟩
    expression form to raise an exception and pass data
    - raise Overflow;
    - raise Signal(x+4);
- Handler
  ⟨exp1⟩ handle ⟨pattern⟩ => ⟨exp2⟩
    evaluate first expression exp1
    if exception that matches pattern is raised,
      then evaluate second expression exp2 instead
        (General form allows multiple patterns)
- Compare
  try {res:=exp1} catch (OvflException oe) {res:=exp2}

ML Exceptions - example

```
- exception noSuchElement ;
- fun nth (n,nil) = raise noSuchElement
    | nth (0,s::ss) = s
    | nth (n,s::ss) = nth((n-1),ss) ;
  > val nth = fn : int * 'a list -> 'a

 - nth(2,[1,2,3]) ;
   > val it = 3 : int
 - nth(4,[1,2,3]) ;
 > uncaught exception noSuchElement
       raised at: stdIn:10.25-10.38

 - fun safeNth(n,xs) = nth(n,xs) handle noSuchElement => 0 ;
   > val safeNth = fn : int * int list -> int
 - safeNth(4,[1,2,3]) ;      > val it = 0 : int
```

Which Handler is Used?

```
- exception Ovflw;
- fun reciprocal(x) =
      if x<=min  then raise Ovflw  else 1.0/x;
- (reciprocal(x) handle Ovflw=>0.0)  /  (reciprocal(x) handle Ovflw=>1.0);
```

- Dynamic scoping of handlers
  - First call handles exception one way
  - Second call handles exception another
  - General dynamic scoping rule

    Jump to most recently established handler on run-time stack
- Dynamic scoping is not an accident
  - User knows how to handler error
  - Author of library function does not

Handlers with pattern matching

- Handler

⟨exp⟩ handle ⟨pattern1⟩ => ⟨exp1⟩
| ⟨pattern2⟩ => ⟨exp2⟩
...
| ⟨pattern3⟩ => ⟨exp3⟩

evaluate first expression exp
if exception that matches one of the patterns is raised,
then evaluate the corresponding expression

Handlers with pattern matching

```
- exception Signal of int;
- fun f(x) = if x=0   then raise Signal(0)
             else if x=1   then raise Signal(1)
             else if x=10 then raise Signal(x-8)
             else (x-2) mod 4;
```

```
- f(10) handle Signal(0) => 0
              | Signal(1) => 1
              | Signal(x) =>
             x+8;
> val it = 10 : int
```

- The expression to the left of the handler is evaluated
- If it terminates normally the handler is not invoked
- If the handler is invoked, pattern matching works as usual in ML

Exception for Error Condition

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

- This function raises an exception when there is no reasonable value to return

```
- lsub(Leaf(3));
> uncaught exception No_Subtree raised at:...

- lsub(Node (Leaf(3),Leaf(5)));
> val it = Leaf 3 : int tree
```

Exception for Efficiency

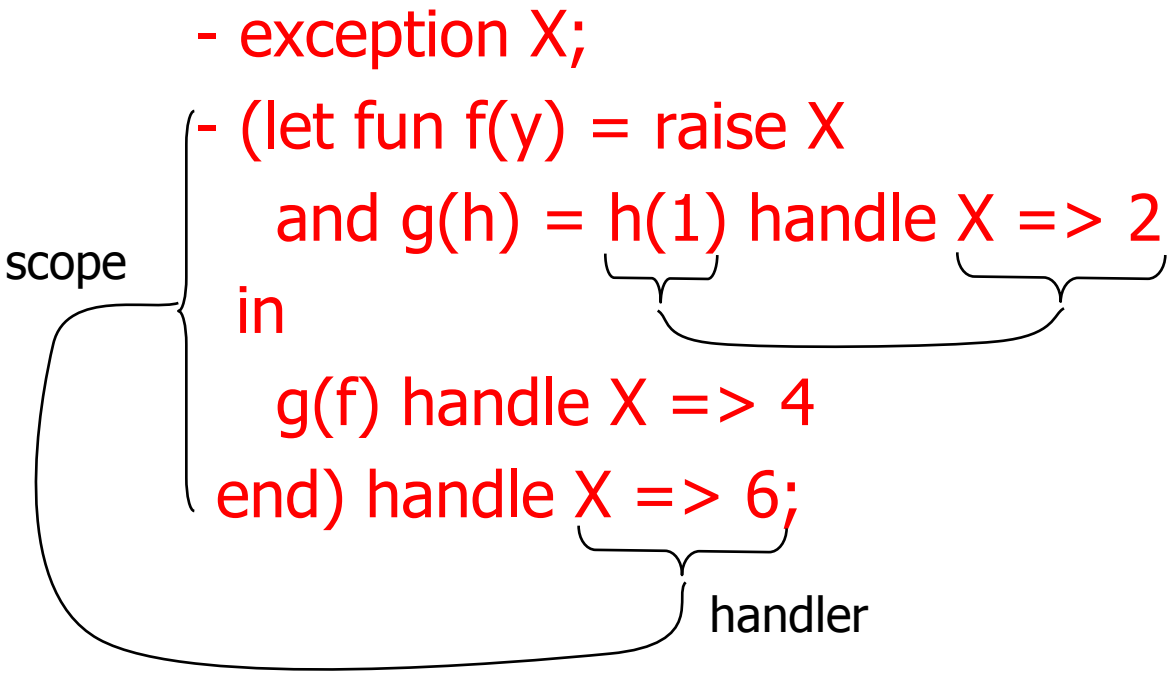
- Function to multiply values of tree leaves

```
- fun prod(LF x) = x: int
  |   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;
```

Dynamic Scope of Handler



What is the value of g(f)?

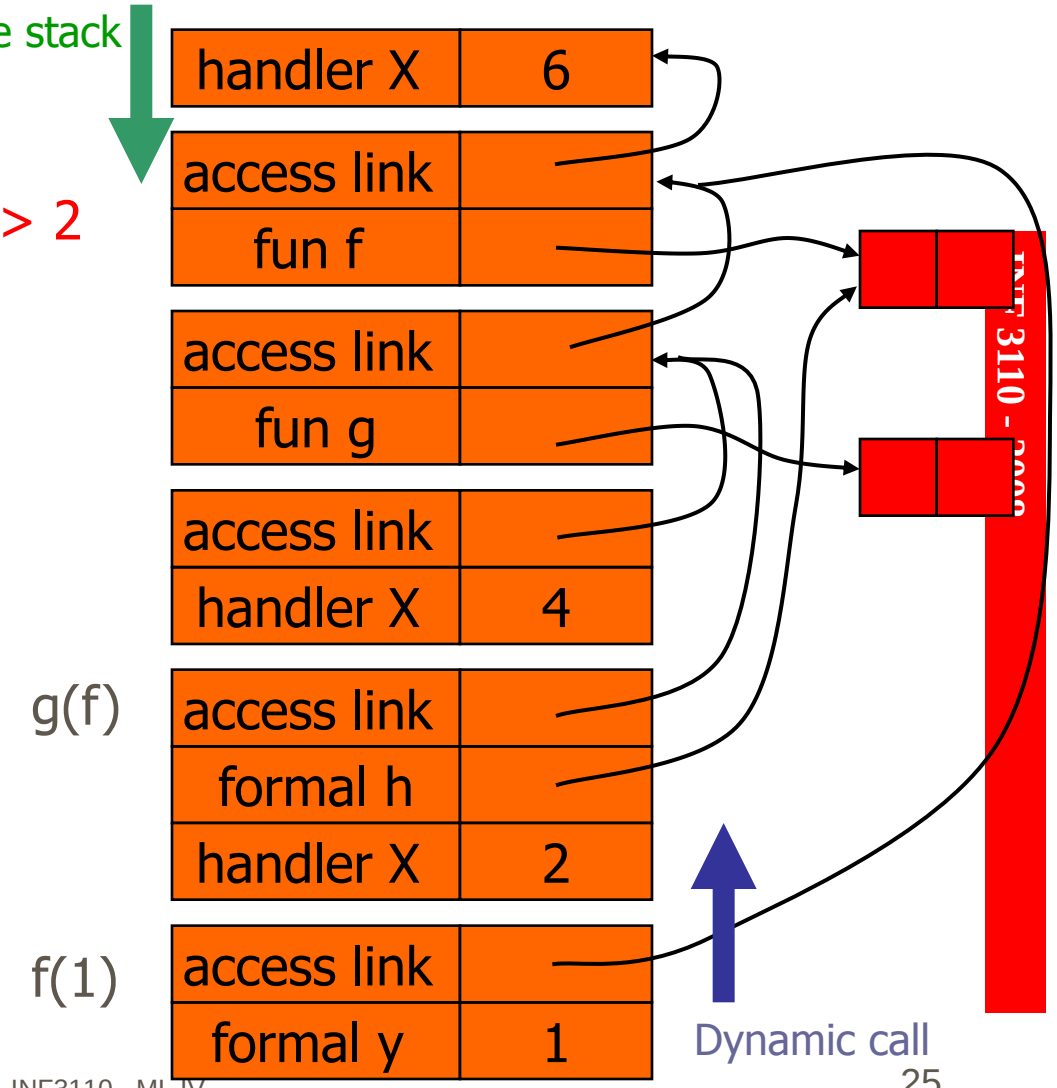
It depends on which handler is used!

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.

Answer: g(f) = **2**



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;
```

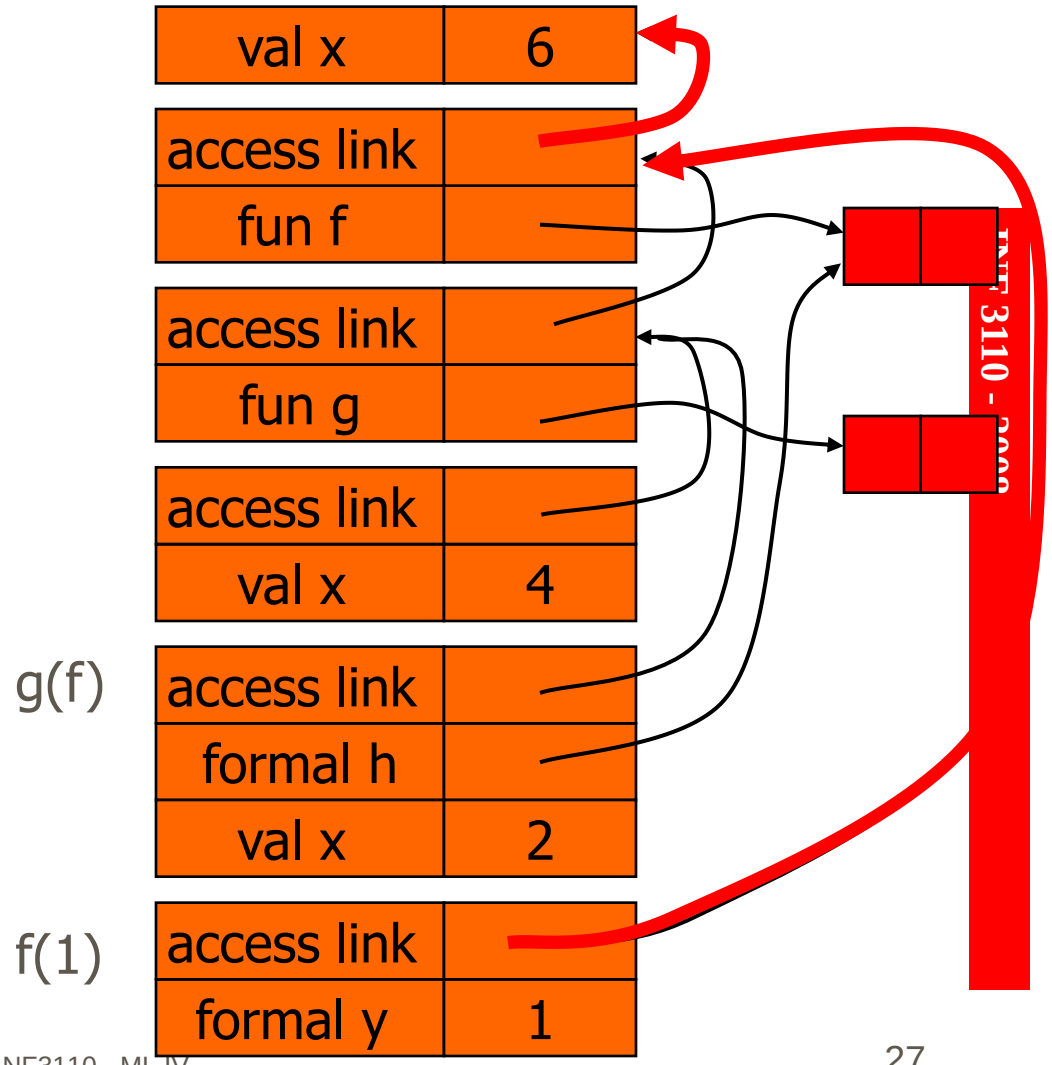
```
val x=6;
(let fun f(y) = x
    and g(h) =
        let val x=2 in h(1)
in
    let val x=4 in g(f)
end);
```

Static Scope of Declarations

```
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.

Answer: g(f) = **6**



Typing of Exceptions

- Typing of raise ⟨exc⟩
  - Recall definition of typing
    - Expression e has type t if normal termination of e produces value of type t
  - Raising exception is not normal termination
    - 1 + raise No_value (the sum will not be performed)
  - Type of raise ⟨exc⟩ is a type variable 'a
- Typing of handle ⟨exc⟩ => ⟨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

```
exception X;
(let
    val x = ref [1,2,3]
 in
    let
        val y = ref [4,5,6]
    in
        ... raise X
     end
end);  handle X => ...
```

[1,2,3] built in the heap, ref x pushed into stack

[4,5,6] built in the heap, ref y pushed into stack

Control is transferred outside the scope

x and y popped off the stack

[1,2,3] and [4,5,6] garbage collected

INF 3110 - 2008

Exceptions and Resource Allocation

```
exception X;
(let
    val x = ref [1,2,3]
 in
    let
        val y = ref [4,5,6]
    in
        ... raise X
     end
end);  handle X => ...
```

- Resources allocated between handler and raise may be "garbage" after exception
- Open files might not be closed

General problem: no obvious solution

Exceptions and Resource Allocation

```
try {
    ...
    fOut = new PrintWriter(new
    FileWriter("OutFile.txt"));
    ...
}
catch (Exception e) {
    ...
}
finally {
        if (fOut != null) {
       fOut.close();
        }  else { ... }
}
```

- Resources allocated between handler and raise may be "garbage" after exception
- Open files might not be closed
- In Java you would use the "finally" construct

Outline

- Structured Programming
  - *go to* considered harmful
- Exceptions
  - "Structured" jumps that may return a value
  - Dynamic scoping of exception handler
- Continuations
- Evaluation order

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. It is a means of capturing the flow of the program and manipulating it.
  - Use it later: "jump" or "exit" by function call
- Useful in
  - Compiler optimization: make control flow explicit
  - Operating system scheduling
  - Function synthesis/design
  - Web applications

Example of Continuation

- Expression

  2*x + 3*y + 1/x + 2/y

- What is the continuation of 1/x?
  - Remaining computation after division

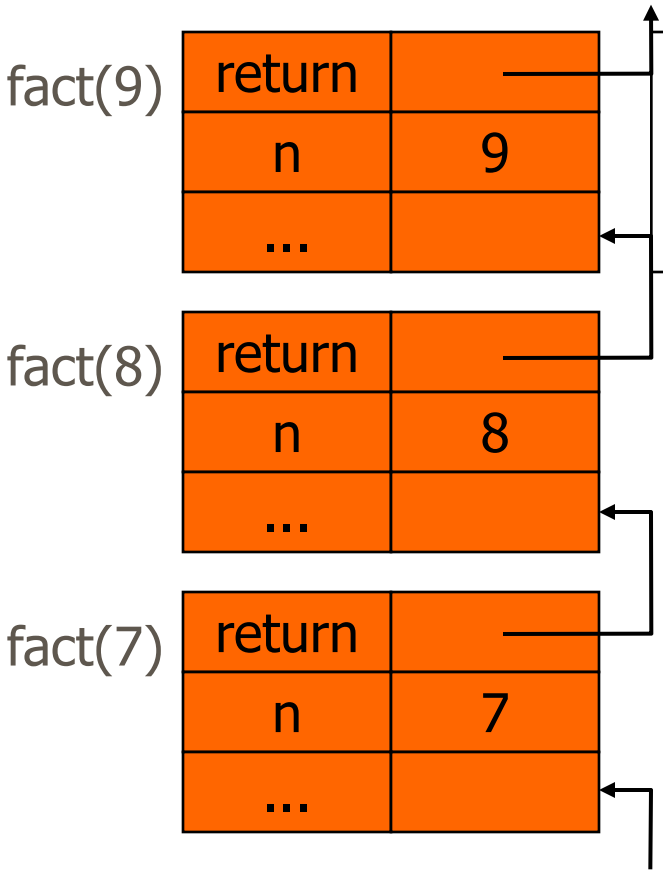
```
let val before = 2*x + 3*y
    fun continue(d) = before + d + 2/y
in
    continue (1/x)
end
```

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 functions 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 $\lambda x.\ 9*x$
- $(\lambda x.\ 9*x)(fact(8)) = 9*fact(8)$

- 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
  $\lambda z.\ \ (\lambda y.\ (\lambda x.\ 9*x)\ (8*y))\ (7*z)$

INF 3110 - 2008

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, $\overbrace{\lambda x.k\ (n*x)}^{\text{continuation}}$)

  fact(n, λx.x) computes n!

- Example computation

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

  = fact(1, λz.((λy.3*y)(2*z)))

  = fact(0, λw. (λz.((λy.3*y)(2*z)) (1*w)))

  = λw. (λz.((λy.3*y)(2*z)) (1*w))) 1

  = λz.((λy.3*y)(2*z)) 1 = (λy.3*y)(2) = 3*2 = 6

INF 3110 - 2008

Tail Recursive Form

- Optimization of continuations

  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

- No automatic tansformation for a given function to tail recursive form, but a function can systematically be transformed to CPS and ... by a clever programmer, to tail recursive from

Other uses for continuations

- Explicit control
  - Normal termination -- call continuation
  - Abnormal termination -- do something else
  - Continuations are more general and flexible than exceptions, but may be more complicated to program
- Compilation techniques
  - Call to continuation is functional form of "go to"
  - Continuation-passing style makes control flow explicit

Web Applications and Services

- Web applications, Web Services, MOM and SOA services
  - Handle long running workflows
  - Progress of subtasks is asynchronous
- Sequential programming is simpler than asynchronous
- Continuations provide
  - An easy way to suspend workflow execution at a wait state
  - Thread of control can be resumed when the next message/event occurs, maybe some long time ahead

Current Java Community effort to support continuations in JVM

Sample projects

- Javaflow a component in the Apache/Jakarta project (http://commons.apache.org/sandbox/javaflow/)
- Cocoon continuations for web apps
- Seaside (smalltalk) continuations for web apps
- RIFE (java based) continuations for web apps
- Borges Ruby port of Seaside ideas
- Modal Web Server Example

  Uses Scheme to build a simple continuation based web server as an example

Reference: http://docs.codehaus.org/display/continuation/Home

“Elevator pitch” for web continuations

- Dijkstra said GOTO was bad idea…
  - In CGI, each link triggers entirely new execution of program
  - Moving from one page to the next is a one-way jump
- Seaside framework presenting the illusion of a continuous interactive session with the user
  - Each page or form acts like a subroutine, returning a value to its caller based on user input
  - Complex, conditional or looping workflows can be described in a single piece of straightforward Smalltalk code as a sequence of calls to individual pages

Reference: http://www.manageability.org/blog/stuff/web-based-continuations/view

Functions and evaluation order

- Another technique for manipulating the order of execution:
  - Place a calculation to be delayed inside a function and pass it to code that will eventually do the calculation
- This is useful if calculation is expensive or might not be needed at all
- Programming constructs: Delay and Force

Evaluation order

- If an expression contains several function calls, we must chose which to evaluate first according to some rules
- There are basically two kinds of evaluation order for functions:
  - **lazy** (call-by-need), outermost reduction , normal order
  - **eager/strict** (call-by-value) , innermost reduction.
    - sq(3+4) → sq 7 → 7*7 → 49 (eager, 3 steps)
    - sq(3+4) → (3+4) * (3+4) → 7 * (3+4) → 7*7 → 49 (lazy, 4 steps)
    - fst(sq(4) , sq(2)) → fst(4*4,sq(2)) → fst(16,sq(2)) → fst(16,2*2) → fst(16,4) → 16 (eager, 5 steps)
    - fst(sq(4) , sq(2)) → sq(4) → 4*4 → 16 (lazy, 3 steps)
    - fst(sq(4) , a_complicated_expression) → sq(4) → 4*4 → 16 (lazy, 3 steps)
- ML uses call-by-value
- Haskell is a lazy language and uses call-by-need

Evaluation order and efficiency

- Which is most efficient?
  - With graph reduction we get more efficient lazy evaluation for functions with repeated arguments:

  - An additional very useful property of lazy evaluation:If an expression has a normal form, then outermost reduction will compute it

- Why not always use "call-by-need"?
  - Needs more bookkeeping
  - We get *non-strict* functions: A function may be defined even if its arguments are not
  - Lazyness leads to infinite data-structures which leads to more complicated mathematical reasoning
  - Lazy evaluation is inefficient in some cases
  - Most lazy languages are purely functional => complicated to combine with commands, e.g. I/O

Lazy evaluation in ML

```
fun f(x,y) =
    ... x ... y
...
f(e1,e2)
```

- Suppose: y is not always needed
- e2 is expensive to calculate.
- I.e. we would like to do something like this:

```
fun f(x,y) =
    ... x ... Force(y)
...
f(e1,Delay(e2))
```

Delay and Force

- Problem: Delay cannot be an ordinary function:
  - To evaluate Delay(e), e must first be evaluated.

- Solution:

```
Delay(e) == fn() => e                    (λ().e)
```

A function which takes no parameters and returns e.

```
Force(e) == e()
```

Application of the function.

```
fun f(x,y) =
    ... x ... e()
...
f(e1, (fn()=>e2) )
```

Evaluation order

- ML uses eager/strict evaluation (call-by-value)
- Haskell uses lazy evaluation with graph reduction
- Haskell has Lazy lists: The elements of a lazy list are not evaluated until their values are required by the rest of the program: thus a lazy list may be infinite
  - In Haskell (a lazy language) all data structures are lazy and infinite lists are common
    - so we could write e.g. [1 ..] or take 5 [1..]
  - In ML this is rare, but we can get lazy lists (or sequences) in ML too
    - The trick is to represent the tail of a list by a function, in order to delay its evaluation

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
- Evaluation order
  - eager (call-by-value) ML
  - lazy (call-by-need) Haskell