Control in Sequential Languages

Reading:
• 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
• 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 \textit{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
throw e  //jump to catch, passing exception object

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

• Optimize using exception
  
  ```haskell
  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

Which catch catches the throw?
Dynamic Scope of Handler

```ml
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?
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

<table>
<thead>
<tr>
<th>Access Link</th>
<th>6</th>
</tr>
</thead>
<tbody>
<tr>
<td>catch(e)</td>
<td></td>
</tr>
<tr>
<td>access link</td>
<td></td>
</tr>
<tr>
<td>fun f</td>
<td></td>
</tr>
<tr>
<td>access link</td>
<td></td>
</tr>
<tr>
<td>fun g</td>
<td></td>
</tr>
<tr>
<td>access link</td>
<td></td>
</tr>
<tr>
<td>catch(e)</td>
<td>4</td>
</tr>
<tr>
<td>access link</td>
<td></td>
</tr>
<tr>
<td>formal h</td>
<td>2</td>
</tr>
<tr>
<td>catch(e)</td>
<td></td>
</tr>
<tr>
<td>access link</td>
<td></td>
</tr>
<tr>
<td>formal y</td>
<td>1</td>
</tr>
</tbody>
</table>

JavaScript version
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.
Compare to static scope of variables

```javascript
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 }
} catch(e) { 6 }

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);
```
Compare to static scope of variables

```ml
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.
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.
Typing of Exceptions  (Haskell)

• Special type IOError of exception
  
  ```haskell
  userError :: String -> IOError
  ```

• Exceptions are raised and caught using
  
  ```haskell
  ioError :: IOError -> IO a
  catch :: IO a -> (IOException -> 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 \texttt{raise \langle exn \rangle}
  – 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: \( 1 + \texttt{raise X} \)

• Typing of \texttt{handle \langle exn \rangle => \langle value \rangle}
  – Converts exception to normal termination
  – Need type agreement
  – Examples
    \begin{align*}
    1 + ((\texttt{raise X}) \texttt{handle X} => e) & \quad \text{Type of } e \text{ must be int} \\
    1 + (e_1 \texttt{handle X} => e_2) & \quad \text{Type of } e_1, e_2 \text{ must be int}
    \end{align*}
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
Example of Continuation Concept

• Expression
  – \(2x + 3y + \frac{1}{x} + \frac{2}{y}\)

• What is continuation of \(\frac{1}{x}\)?
  – Remaining computation after division

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

• Expression
  – $2x + 3y + \frac{1}{x} + \frac{2}{y}$

• What is continuation of $\frac{1}{x}$?
  – Remaining computation after division

let val before = $2x + 3y$
  fun continue(d) = before + d + $\frac{2}{y}$
in
  continue ($\frac{1}{x}$)
end
Example: Tail Recursive Factorial

• Standard recursive function
  \[\text{fact}(n) = \begin{cases} 1 & \text{if } n=0 \\ n \times \text{fact}(n-1) & \text{else} \end{cases}\]

• Tail recursive
  \[\text{f}(n,k) = \begin{cases} k & \text{if } n=0 \\ \text{f}(n-1, n \times k) & \text{else} \end{cases}\]
  \[\text{fact}(n) = \text{f}(n,1)\]

• How could we derive this?
  – Transform to continuation-passing form
  – Optimize continuation function to single integer
Continuation view of factorial

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

fact(9)

<table>
<thead>
<tr>
<th>return</th>
</tr>
</thead>
</table>
| n      | 9
| ...    |

This invocation multiplies by 9 and returns
Continuation of fact(8) is
$$\lambda x. 9 \times x$$

fact(8)

<table>
<thead>
<tr>
<th>return</th>
</tr>
</thead>
</table>
| n      | 8
| ...    |

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

fact(7)

<table>
<thead>
<tr>
<th>return</th>
</tr>
</thead>
</table>
| n      | 7
| ...    |

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

  fact(n, \lambda x.x) computes n!

• Example computation
  fact(3,\lambda x.x) = fact(2, \lambda y.(\lambda x.(3*y)))
  = fact(1, \lambda x.(\lambda y.(3*y)(2*x)))
  = \lambda x.(\lambda y.(3*y)(2*x)) 1 = 6
Tail Recursive Form

• Optimization of continuations
  \[ \text{fact}(n, a) = \begin{cases} a & \text{if } n = 0 \\ \text{fact}(n-1, n \cdot a) & \text{else} \end{cases} \]
  Each continuation is effectively \( \lambda x.(a \cdot x) \) for some \( a \)

• Example computation
  \[ \text{fact}(3, 1) = \text{fact}(2, 3) \]  was  \( \text{fact}(2, \lambda y.3 \cdot y) \)
  \[ = \text{fact}(1, 6) \]  was  \( \text{fact}(1, \lambda x.6 \cdot x) \)
  \[ = 6 \]
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 \( \lambda \)-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”
  \[
  \text{<atom>} ::= \text{<smbl>} | \text{<number>}
  \]
  \[
  \text{<smbl>} ::= \text{<char>} | \text{<smbl><char>} | \text{<smbl><digit>}
  \]
  \[
  \text{<num>} ::= \text{<digit>} | \text{<num><digit>}
  \]

• Dotted pairs
  – Write \((A . B)\) for pair
  – Symbolic expressions, called \(S\)-expressions:
    \[
    \text{<sexp>} ::= \text{<atom>} | (\text{<sexp>}.\text{<sexp>})
    \]

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 \( (\text{function } \text{arg}_1 \ldots \text{arg}_n) \)
  – evaluate each of the arguments
  – pass list of argument values to function
• Special forms do not eval all arguments
  – Example \( (\text{cond } (p_1 \ e_1) \ldots (p_n \ e_n)) \)
    • proceed from left to right
    • find the first \( p_i \) with value true, eval this \( e_i \)
  – Example \( (\text{quote } A) \) 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

\[(\text{cond } (p_1 e_1) (p_2 e_2) \ldots (p_n e_n))\]

• Evaluate conditions \(p_1 \ldots p_n\) 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

\(p_1 \ldots p_i\) false and \(p_{i+1}\) has no value, or

relevant \(p_i\) true and \(e_i\) has no value
Examples

\[(\text{cond } ((< 2 \ 1) \ 2) \ ((< 1 \ 2) \ 1))\] 
has value 1

\[(\text{cond } ((< 2 \ 1) \ 2) \ ((< 3 \ 2) \ 3))\]
has no value

\[(\text{cond } (\text{\textit{diverge}} \ 1) \ (\text{true} \ 0))\]
no value, if expression diverge loops forever

\[(\text{cond } (\text{true} \ 0) \ (\text{\textit{diverge}} \ 1))\]
has value 0
Function Expressions

• Form

  (lambda ( parameters ) ( function_body ))

• Syntax comes from lambda calculus:

  \( \lambda f. \lambda x. f (f x) \)
  
  (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
- Atoms and lists represented by cells

![Diagram showing Lisp Memory Model with cells and addresses]

<table>
<thead>
<tr>
<th>Address</th>
<th>Decrement</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>0</td>
</tr>
<tr>
<td>B</td>
<td>0</td>
</tr>
<tr>
<td>C</td>
<td>0</td>
</tr>
</tbody>
</table>
• Both structures could be printed as \(((A.B) \cdot (A.B))\)
• Which is result of evaluating
  \(\text{cons (cons 'A 'B) (cons 'A 'B)}\)?

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 ( e₁) ( e₂ )))

Cells created in evaluation of e₂ may be garbage, unless shared by e₁ 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