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

Historical Debate

- Dijkstra, Go To Statement Considered Harmful
  - Letter to Editor, CACM, 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

```javascript
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

```javascript
function invert(matrix) {
    if ... throw "Determinant";
    ...
}:
try {
    ... invert(myMatrix);
    ...
} catch (e) {
    ...
    // recover from error
}
```

### C++ Example

```c++
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.

### ML determinant example

```ml
exception Determinant; (* declare exception name *)
fun invert (M) = (* function to invert matrix *)
    ...
    if ...
    then raise Determinant (* exit if Det=0 *)
    else ...
end;
try
    invert (myMatrix) handle Determinant => ...
    catch
Value for expression if determinant of myMatrix is 0
```
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 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
    end)

Dynamic Scope of Handler
JavaScript version
try{
  function f(y) { throw "exn";
    function g(h) { try { h(1) catch(e) {return 2};
    };
      try {
        g(f); } catch(e) {6};
      } catch(e6) {1};
  }
}
Which catch catches the throw?

Dynamic Scope of Handler
Book version (ML)
try{
  function f(y) { throw "exn";
    function g(h) { try { h(1) catch(e) {return 2};
    };
      try {
        g(f); } catch(e) {6};
      } catch(e6) {1};
  }
}
Which handler is used?

Dynamic Scope of Handler
JavaScript version
try{
  function f(y) { throw "exn";
    function g(h) { try { h(1) catch(e) {return 2};
    };
      try {
        g(f); } catch(e) {6};
      } catch(e6) {1};
  }
}
Dynamic scope:
find first handler, going up the dynamic call chain

Dynamic Scope of Handler
Book version (ML)
try{
  function f(y) { throw "exn";
    function g(h) { try { h(1) catch(e) {return 2};
    };
      try {
        g(f); } catch(e) {6};
      } catch(e6) {1};
  }
}
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; }
}
```

### Book version (ML)

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

### Static Scope of Declarations

#### JavaScript version

```
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)

```
val x=6;
(let fun f(y) = x
and g(h) = let val x=2 in h(1)
in
  g(f)
) handle X => 6;
```

### 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 (ve -> 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 + (a1 handle X => a2)`  
        Type of `a1`, `a2` 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

Example of Continuation Concept

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

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

Example: Tail Recursive Factorial

- Standard recursive function
  - \(\text{fact}(n) = \text{if } n=0 \text{ then } 1 \text{ else } n*\text{fact}(n-1)\)
- Tail recursive
  - \(f(n,k) = \text{if } n=0 \text{ then } k \text{ else } f(n-1, n*k)\)
  - \(\text{fact}(n) = f(n,1)\)
- How could we derive this?
  - Transform to continuation-passing form
  - Optimize continuation function to single integer

Example of Continuation Concept

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

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

Continuation view of factorial

- \(\text{fact}(n) = \text{if } n=0 \text{ then } 1 \text{ else } n*\text{fact}(n-1)\)
- This invocation multiplies by 9 and returns
  - Continuation of \(\text{fact}(8)\) is \(\lambda x. 9*x\)

- Multiples by 8 and returns
  - Continuation of \(\text{fact}(7)\) is \(\lambda y. (\lambda x. 9*x) (8*y)\)

- Multiples by 7 and returns
  - Continuation of \(\text{fact}(6)\) is \(\lambda z. (\lambda y. (\lambda x. 9*x) (8*y)) (7*z)\)
Derivation of tail recursive form

- Standard function
  \( \text{fact}(n) = \begin{cases} \text{if } n=0 \text{ then } 1 \text{ else } n \times \text{fact}(n-1) \end{cases} \)
- Continuation form
  \[
  \text{fact}(n, k) = \begin{cases} \text{if } n=0 \text{ then } k(1) \text{ else } \text{fact}(n-1, \lambda x. n \times x) \end{cases}
  \]
  \( \text{fact}(n, \lambda x. x) \) computes \( n! \)
- Example computation
  \[
  \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)) \)
  \( 1 = 6 \)

Tail Recursive Form

- Optimization of continuations
  \[
  \text{fact}(n, a) = \begin{cases} \text{if } n=0 \text{ then } a \text{ else } \text{fact}(n-1, n*a) \end{cases}
  \]
  Each continuation is effectively \( \lambda x. (a*x) \) for some \( a \)
- Example computation
  \[
  \text{fact}(3, 1) = \text{fact}(3, 3) \quad \text{was } \text{fact}(2, \lambda y. 3*y)
  = \text{fact}(1, 6) \quad \text{was } \text{fact}(1, \lambda x. 6*x)
  = 6
  \]

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.”

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.

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 Web programming

- Use continuation-passing style to allow multiple returns
  
  ```javascript
  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/cpu/
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, ...
  - Common Lisp 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> ::= <num> | <smbl>
  <num> ::= <digit> | <num><digit>
  <smbl> ::= <char> | <smbl><char> | <smbl><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
- Handout provides executable alternative, so examples run in Scheme
- 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
  \((\text{lambda} (x) (\text{cond} ((\text{atom} x) x) (T (\text{cons} 'A x)))))\)
  function \(f(x) = \text{if atom}(x) \text{then} x \text{else cons}('A',x)\)
- 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 \((\text{cond } (p_1 \ e_1) \ldots (p_n \ e_n))\)
    \(\text{• proceed from left to right}\)
    \(\text{• find the first } p_i \text{ with value true, eval this } e_i\)
  - Example \((\text{quote } A)\) does not evaluate A

Examples

\((+ 4 5)\)
expression with value 9
\((\text{(} (+ 1 2) (+ 4 5))\)
evaluate 1+2, then 4+5, then 3+9 to get value
\((\text{cons} \ (\text{quote} \ A) \ (\text{quote} \ B))\)
pair of atoms A and B
\((\text{quote} (+ 1 2))\)
evaluates to list \((+ 1 2)\)
\('(+ 1 2)\)
same as \((\text{quote} (+ 1 2))\)

Conditional Expressions in Lisp

- Generalized if-then-else
  \((\text{cond } (p_1 \ e_1) \ldots (p_n \ e_n))\)
  \(\text{• evaluate conditions } p_1 \ldots p_n \text{ left to right}\)
  \(\text{• if } p_i \text{ is first condition true, then evaluate } e_i\)
  \(\text{• Value of } e_i \text{ is value of expression}\)
  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 \(e_j\) has no value

Examples

\((\text{cond } (\text{< } 2 1) 2) (\text{< } 1 2) 1)\)
has value 1
\((\text{cond } (\text{< } 2 1) 2) (\text{< } 3 2) 3)\)
has no value
\((\text{cond } (\text{diverge} \ 1) \ (\text{true} \ 0))\)
no value, if expression diverge loops forever
\((\text{cond } \text{(true} 0) (\text{diverge} \ 1))\)
has value 0

Function Expressions

- Form
  \((\text{lambda } (\text{parameters} ) \ (\text{function body} ))\)
- Syntax comes from lambda calculus:
  \(\lambda f. \lambda x. f \ (f \ x)\)
  \((\text{lambda} \ (f) \ (\text{lambda} \ (x) \ (f \ (f \ x)))))\)
- Defines a function but does not give it a name t
  \(\text{(lambda} \ (f) \ (\text{lambda} \ (x) \ (f \ (f \ x)))))\)
  \(\text{(lambda} \ (x) \ (+ \ 1 \ x)))\)
  \}

Example

\((\text{define} \ \text{twice} \ (\text{lambda} \ (f) \ (\text{lambda} \ (x) \ (f \ (f \ x)))))\)

\((\text{define} \ \text{inc} \ (\text{lambda} \ (x) \ (+ \ 1 \ x)))\)

\((\text{(twice inc)} 2)\)
\(\Rightarrow 4\)
Lisp Memory Model

- Cons cells
- Atoms and lists represented by cells

Sharing

- Both structures could be printed as \((A.B). (A.B)\)
- Which is result of evaluating \((\text{cons} (\text{cons} 'A 'B) (\text{cons} 'A 'B))\)?

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

\[(\text{car} (\text{cons} (e_1) (e_2)))\]

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

\[((\text{lambda} (x) (\text{car} (\text{cons} \ldots x \ldots) \ldots x \ldots)))\]

'\((\text{Big Mess})\)

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