Lecture 17: Control Structures

CSC 131
Fall, 2014

Kim Bruce

Control Structures

- FORTRAN
  - GO TO n
  - GO TO (17, 43, 12, 99), I (also other variants)
  - IF(arith exp) 17, 43, 12
    means go to statement number 17 if arith exp is negative, 43 if zero, and 12 if positive
  - DO label ivble = 1, 20, 2

- Close to machine code

ALGOL 60

- GO TO 99
- IF ... THEN ... ELSE ... (hierarchical)
- for i := 3, 7, 11 step 1 until 16, i/2 while i >= 1, 2 step i until 32 do ..
  - BAROQUE, all expressions re-eval each time through loop:
    - 3, 7, 11, 12, 13, 14, 15, 16, 8, 4, 2, 1, 2, 4, 8, 16, 32.
- switch - like in C/C++/Java.

Goto Statements

- Why need repetition - can do it all with goto's?
- "The static structure of a program should correspond in a simple way with the dynamic structure of the corresponding computation."
  Dijkstra 1968 letter to CACM.
Pascal

- go to
- if .. then .. else
- for, while, repeat (confusion w/positive vs. negative exit)
- labeled case - Tony Hoare
  - clear & efficient
  - construct jump table,
  - optimize depending on size,
  - self-documenting.

More on Case

- Modula 2 improved by adding otherwise clause
- Haskell & ML's pattern matching is compiled into a case statement:
  ```haskell
  fun reverse l = case l of
    nil => nil |
    (h::rest) => (reverse rest)[h];
  ```
- if-then-else as well

Grace's Match - Case

```haskell
match ( x ) // x : 0 | String | Student
  // match against a constant
  case { 0 -> print("Zero") }
  // typematch, binding a variable
  case { s : String -> print(s) }
  // destructuring match, binding variables ...
  case { Student(name, id) -> print (name) }

Scala has similar destructuring match
```

Refresher: Natural Semantics of Commands

```
(e, ev, s) => (v; s')  where ev(x) = loc
(x := e, ev, s) => s'[loc:=v]
(C1, ev, s) => s'  (C2, ev, s') => s''
(C1; C2, ev, s) => s''
(b, ev, s) => (true, s')  (C1, ev, s') => s''
(if b then C1 else C2, ev, s) => s''
```

If every statement returns a value then also return v from semantics
Semantics of While

\[(b, ev, s) \Rightarrow (false, s')\]
\[\text{-------------------------------}\]
\[(while \ b \ do \ C, ev, s) => s'\]
\[(b, ev, s) \Rightarrow (true, s') \quad ((C, ev, s') \Rightarrow s'')\]
\[\text{-------------------------------}\]
\[(while \ b \ do \ C, ev, s'') => s'''\]

\[\text{Notice similarity between}\]
\[\text{while } E \text{ do } C\]
\[\text{and}\]
\[\text{if } E \text{ then begin }\]
\[\quad C;\]
\[\text{while } E \text{ do } C\]
\[\text{end}\]

Iterators

- Abstract over control structures (in Clu)
  
  for \(c : \text{char in string\_chars(s)}\) do ...
  - where
    
    \[\text{string\_chars = iter } (s : \text{string}) \text{ yields } (\text{char});\]
    
    \[\text{index : Int := 1;}\]
    
    \[\text{limit : Int := string\$size } (s);\]
    
    \[\text{while index <= limit do}\]
    
    \[\text{yield } (\text{string\$fetch}(s, \text{index}));\]
    
    \[\text{index := index + 1;}\]
    
    \[\text{end};\]
    
    \[\text{end string\_chars;}\]

Implementing Iterators

- Just another object w/state in o-o language
- What about procedural?
- How can we retain state?
- Specific kind of coroutine.

When Good Programs Go Bad!
Handling Errors

- What happens when something goes wrong, e.g., with read from file.
- In C returns error condition, which is generally ignored.
- In more modern languages, throw exception, which must be handled or crash.

Exceptions

- Designed to handle unexpected errors.
- Exception handlers based on dynamic calls, not static scope.
- Allows program to recover from exceptional conditions, esp. beyond programmers control
- Can be abused!

Example Exceptions

- Arithmetic, array bounds, or I/O faults,
- Failure of preconditions
- Unpredictable conditions
- Tracing program flow in debugger

Exception Handling

- Ada:
  - raise exception_name;
  - handling:
    begin
    C
    exception
    when exp_name1 => C'
    when exp_name2 => C''
    when others => C'
- Java, C++ similar w/ “throw” & “try-catch”
Handling Exceptions

- When throw exception -- where look for handler?
  - Same unit? (Ada/C++/Java)
  - Calling unit? (Clu)
  - If not find, continue up call chain

After Handling ...

- (Ada/Java/ML/Haskell): Return from block
- PL/I: Resumption model: re-execute failed statement.
- Eiffel: Re-execute block where failure occurred
- ML & Java -- exceptions can take parameters

Haskell uses Monads

```haskell
data Exn a = Oops String | Answer a deriving (Show)

instance Monad Exn where
  return a = Answer a -- return :: a -> Exn a

  >>= :: M a -> (a -> M b) -> M b
  (Oops s) >>= f = Oops s
  (Answer a) >>= f = f a

throw :: String -> Exn a
throw = Oops

catch :: Exn a -> (String -> Exn a) -> Exn a

catch (Oops l) h = h l

catch (Answer r) _ = Answer r

See Stone's ExcInterp.hs for interpreter handling exceptions
```

Exceptions in Java

- Objects from subclass of Exception class
  ```java
  try {
      ...
  } catch (ExcType ex) {
      ...
  } catch (ExcType' ex) {...} ...
  ```
- If not caught, must declare. E.g.
  ```java
  public E pop() throws EmptyStackException {
      ... throw new EmptyStackException(); ...
  }
  ```

Pattern matching!!
**RuntimeException**

- If exceptions subclasses of RuntimeException then need not be declared in method headers

- Ex.:
  - NullPointerException,
  - ArrayIndexOutOfBoundsException,
  - IllegalArgumentException,
  - NumberFormatException, and ArithmeticException

- Unfortunately, also includes EmptyStackException

  *Talk later about problems!*

**If Exception Not Handled**

- Pop off activation records while searching for handler.
- What if allocated memory in unit being popped?
- OK if garbage collection, but ...
- Closing files also problems

**Java try-catch-finally**

```
try {
    ...
} catch (ExcType ex) {
    ...
} catch (Exc'Type ex) {
    ...
} finally {... }
```

*No matter how you complete block, will execute finally clause*

**So far ...**

- Structured Programming
  - Goto considered harmful

- Exceptions
  - Structured jumps -- can carry a value
  - dynamic scoping of exception handler

- Continuations ...
Continuations

• Continuation of expression is remaining work to be done after evaluating expression
  - the future
  - Represented as a function, applied to value of exp, which is value computed so far.

• Capture continuation
  - use it later to return to execution.

• Explicitly represented in Scheme, ML

• Have been important in compilers for functional languages, concurrency, web programming