Lecture 26: Control Structures: Iterators & Exceptions

CSC 131
Fall, 2006

Natural Semantics of Commands

$$\frac{(e, ev, s) \Rightarrow (v, s')}{(x := e, ev, s) \Rightarrow s'[loc:=v]} \text{ where } ev(x) = loc$$

$$\frac{(C_1, ev, s) \Rightarrow s' \quad (C_2, ev, s') \Rightarrow s''}{(C_1; C_2, ev, s) \Rightarrow s''}$$

$$\frac{(b, ev, s) \Rightarrow (true, s') \quad (C_1, ev, s') \Rightarrow s''}{(\text{if } b \text{ then } C_1 \text{ else } C_2, ev, s) \Rightarrow s''}$$

Semantics of While

$$\frac{(b, ev, s) \Rightarrow (false, s')}{(\text{while } b \text{ do } C, ev, s) \Rightarrow s'}$$

$$\frac{(b, ev, s) \Rightarrow (true, s') \quad (C, ev, s') \Rightarrow s'' \quad (\text{while } b \text{ do } C, ev, s'') \Rightarrow s'''}{(\text{while } b \text{ do } C, ev, s) \Rightarrow s'''}$$

Notice similarity between

```
while E do C
```

and

```
if E then begin
    C;
    while E do C
end
```

Iterators

- Abstract over control structures

```
for c : char in string_chars(s) do ...
```

- where

```
string_chars = iter (s : string) yields (char);
    index : Int := 1;
    limit : Int := string$size (s);
    while index <= limit do
        yield (string$fetch(s, index));
        index := index + 1;
    end;
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.

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 excp_name1 => C'
   when excp_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): Return from block
- PL/I: Resumption model: re-execute failed statement.
- Eiffel: Reexecute block where failure occurred
- ML & Java -- exceptions can take parameters

ML example

```
datatype 'a stack = EmptyStack |
                    Push of 'a * ('a stack);
exception empty;

fun pop EmptyStack = raise empty
  | pop(Push(n,rest)) = rest;

fun top EmptyStack = raise empty
  | top (Push(n,rest)) = n;

fun IsEmpty EmptyStack = true
  | IsEmpty (Push(n,rest)) = false;
```

ML Match Parens Example

```
exception nomatch;

fun buildstack nil initstack = initstack
  | buildstack ("("::rest) initstack =
                 buildstack rest (Push("(",initstack))
  | buildstack (")"::rest) (Push("(",bottom)) = bottom
  | buildstack (")"::rest) initstack = raise nomatch
  | buildstack (fst::rest) initstack =
                           buildstack rest initstack;

fun balanced string =
            (buildstack (explode string) = EmptyStack)
                            handle nomatch => false;
```

Questionable use of exceptions

Exceptions in SML

Can carry parameters

```
exception error of String;
```

General form:

```
 <exp> handle <pat1> => <exp1>
             | <pat2> => <exp2>
                      ...
             | <patn> => <expn>
```

Exceptions in Java

- Objects from subclass of Exception class

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

- If not caught, must declare. E.g.

```
public E pop() throws EmptyStackException {
   ... throw new EmptyStackException(); ...
}
```

RuntimeException

- Exceptions from subclasses need not be declared in method headers
- Ex.:
  - NullPointerException, ArrayIndexOutOfBoundsException, IllegalArgumentException, NumberFormatException, and ArithmeticException
- Unfortunately, also includes `EmptyStackException`

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 {... }
```

Continuations

- Rest of computation
- Explicitly represented in Scheme
- Have been important in compilers for functional languages
- Other techniques have superseded lately, so will skip.

Manipulating Evaluation Order

- What if actual params are expensive to evaluate, but aren't always used?
- Can suspend evaluation of e by replacing it by Delay(e) = fn () => e.
- Evaluate by Force(e) = e()
- Important to have as macros, not functions!

Call-by-need

- If have to evaluate several times then expensive.
- Cache answer once computed so can be accessed by other occurrences.

Implementing Call-by-need

```
datatype `a delay = Ev of `a |
                    UnEval of unit -> `a;

fun ev(Ev(x)) = x
  | ev(UnEval(f)) = f();

Delay(e) = ref( UnEval(fn () => e))

Force(e) = let
              val v = ev(!d)
           in
              (d := Ev(v);v)
           end;
```

Summary of Statements

- Progression from goto to higher-level abstractions:
  - Expression ⇒ function
  - Statement ⇒ procedure
  - control structure ⇒ iterator
- Modern control: iterators, exceptions, continuations, delay-force