CS558 Programming Languages

Fall 2015

Lecture 5

Structured Control Flow

All modern higher-level imperative languages are designed to support **structured programming**.

Loosely, a structured program is one in which the **syntactic structure** of the program text corresponds to the **flow of control** through the dynamically executing program.

Originally proposed (most famously by Dijkstra) as an improvement on the incomprehensible "spaghetti code" that is easy to produce using the labels and jumps supported directly by hardware.

More specifically, structured programs use a very small collection of (recursively defined) **compound statements** to describe their control flow.

KINDS OF COMPOUND STATEMENTS

- Sequential composition: form a statement from a sequence of statements, e.g.

(Java) `{ x = 2; y = x + 4;}`

(Pascal) `begin x := 2; y := x + 4; end`

- Selection: execute one of several statements, e.g.,

(Java) `if (x < 0) y = x + 1; else z = y + 2;`

- Iteration: repeatedly execute a statement, e.g.,

(Java) `while (x > 10) output(x--);`

(Pascal) `for x := 1 to 12 do output(x*2);`

SELECTION: IF

The basic selection statement is based on boolean values

`if` e `then` s_1 `else` s_2

which translates to

evaluate e *into* t
`cmp` t`,true`
`brneq` l_1
s_1
`br` l_2
l_1: s_2
l_2:

SELECTION: CASE

To test types with more than two values, multi-way selections against constants are appropriate:

```
case e of
  c_1 : s_1
  c_2 : s_2
  ...
  c_n : s_n
  default : s_d
```

The most efficient translation of `case` statements depends on **density** of the value $c_1, c_2, \ldots, c_n$ within the range of possible values for e.

SPARSE CASES

For **sparse** distributions, it's best to translate the `case` just as if it were:

```
t := e;
if t = c1 then
  s1
else if t = c2 then
  s2
else
  ...
else if t = cn then
  sn
else
  sd
```

DENSE CASES

For a **dense** set of labels in the range $[c_1, c_n]$, it's better to use a **jump table**:

```
        evaluate e into t          l1:     s1
        cmp t,c1                           br done
        brlt ld                    l2:     s2
        cmp t,cn                           br done
        brgt ld                            ...
        sub t,c1,t                 ln:     sn
        add table,t,t                      br done
        br *t                      ld:     sd
table:  l1                         done:
        l2
        ...
        ln
```

The best approach for a given `case` may involve a combination of these two techniques. Compilers differ widely in the quality of the code generated for `case`.

ITERATION

The basic loop construct is

```
while e do s
```

corresponding to:

```
top:   evaluate e into t
       cmp t,true
       brneq done
       s
       br top
done:
```

A commonly-supported variant is to move the test to the bottom:

```
repeat s until e
```

which is equivalent to:

```
s;
while not e do s
```

LOOP EXITS

It is sometimes desirable to exit from the middle of a loop:

```
loop
  s1;
  exitif e;
  s2
end
```

is equivalent to:

```
top: s1
     evaluate e into t
     cmp t,true
     breq done
     s2
     br top
done:
```

C/C++/Java have an unconditional form of `exit`, called `break`. They also have a `continue` statement that jumps back to the top of the loop.

USES FOR goto?

An efficient program with goto:

```
    int i;
    for (i = 0; i < n; i++)
      if (a[i] == k)
        goto found;
    n++;
    a[i] = k;
    b[i] = 0;
  found:
    b[i]++;
```

In most languages (e.g., Modula, C/C++) there is **no** equivalently efficient solution without goto.

MULTI-LEVEL `break`

But we **can** do as well in Java, using a named, multi-level `break`:

```
    int i;
  search:
    { for (i = 0; i < n; i++)
        if (a[i] == k)
          break search;
      n++;
      a[i] = k;
      b[i] = 0;
    }
    b[i]++;
```

(This construct was invented by Knuth in the 1960's, but not adopted into a mainstream language for about 30 years!)

COUNTED LOOPS

Since iterating a definite number of times is very common, languages often offer a dedicated statement, with basic form:

`for` i `:=` e_1 `to` e_2 `do` s

Here s is executed repeatedly with i taking on the values e_1, $e_1 + 1$, ..., e_2 in each successive iteration.

The detailed semantics of this statement vary, and can be tricky. Often, s is prohibited from modifying i, which (under certain other conditions) guarantees that the loop will be executed exactly $e_2 - e_1 + 1$ times.

C/C++/Java have a much more general version of `for`, which guarantees much less about the behavior of the loop:

`for (`e_1`;` e_2`;` e_3`)` s`;`

is exactly equivalent to:

e_1`; while (`e_2`) {` s`;` e_3 `}`

THE COME FROM STATEMENT

```
10 J = 1
11 COME FROM 20
12 PRINT J
   STOP
13 COME FROM 10
20 J = J + 2
```

(R. Lawrence Clark, “A linguistic contribution to GOTO-less programming,” *Datamation*, 19(12), 1973, 62-63.)

But is this really a joke?

Even with a `GO TO`, we must examine both the branch **and** the target label to understand the programmer’s intent.

EXCEPTIONS

Programs often need to handle **exceptional** conditions, i.e., deviations from “normal” control flow.

Exceptions may arise from

- failure of built-in or library operations (e.g., division by zero, end of file)
- user-defined events (e.g., key not found in dictionary)

It can be awkward or impossible to deal with these conditions explicitly without distorting normal code.

Most recent languages (Ada, C++, Java, Python, OCaml, etc.) provide a means to **define**, **raise** (or **throw**), and **handle** exceptions.

EXAMPLE: EXCEPTIONS IN SCALA

```
class Help extends Exception  // define a new exception

try {
  ...
  if (gone wrong)
    throw new Help // raise user-defined exception
  ...
  x = a / b  // might raise a built-in exception
  ...
} catch {
  case _: Help => ...report problem...
  case _: ArithmeticException => x = -99  // repair damage
}
```

WHAT TO DO IN AN EXCEPTION?

If there is a statically enclosing handler, the thrown exception behaves much like a `goto`. In previous example:

```
...
if (gone wrong) goto help_label;
...
help_label: ...report problem...
```

But what if there is no handler explicitly wrapped around the exception-throwing point?

• In most languages, uncaught exceptions **propagate** to next **dynamically** enclosing handler. E.g, caller can handle uncaught exceptions raised in callee.

• A few languages support **resumption** of the program at the point where the exception was raised.

• Many languages permit a value to be returned along with the exception itself.

Exception Handling Example

```
case class BadThing(problem:String) extends Exception

def foo() = {
  ...  throw BadThing("my problem") ...
}

def bar() {
  try {
    foo()
  } catch {
    case BadThing(problem) => println("oops:" + problem )
  }
}
```

Exceptions vs. Error Values

An alternative to user-raised exceptions is to return status values, which must be checked on return:

```
def find (k0:String,env:List[(String,Int)]) : Option[Int] =
  env match {
    case Nil => None
    case (k,v)::t => if (k == k0)
                       Some(v)
                     else find(k0,t)
  }

...
find("abc",e) match {
  case Some(v) => ... v ...
  case None => ...perform error recovery...
}
```

EXCEPTION VS. ERROR VALUES (2)

With exceptions, we can defer checking for (rare) error conditions to a more convenient point.

```
class NotFound extends Exception
def find (k0:String,env:List[(String,Int)]) : Int =
  env match {
    case Nil => throw new NotFound
    case (k,v)::t => if (k == k0)
                       v
                     else find(k0,t)
    }

...
try {
  val v = find ("abc",e)
  ... v ...
} catch {
  case _:NotFound => ...perform error recovery...
}
```

IMPLEMENTING EXCEPTIONS (1)

One approach to implementing exceptions is for the runtime system to maintain a **handler stack** with an entry for each handler context currently active.

- Each entry contains a handler code address and a call stack pointer.
- When the scope of a handler is entered (e.g. by evaluating a `try...with` expression), the handler's address is paired with the current call stack pointer and pushed onto the handler stack.
- When an exception occurs, the top of the handler stack is popped, resetting the call stack pointer and passing control to the handler's code. If this handler itself raises an exception, control passes to the next handler on the stack, etc.
- Selective handlers work by simply re-raise any exception they don't want to handle (causing control to pass to the next handler on the stack).

EXCEPTIONS ON PURPOSE

• In this execution model, raising an exception provides a way to return quickly from a deep recursion, with no need to pop stack frames one at a time.

Example:

```
def product(l:List[Int]) : Int = {
  def prod(l:List[Int]) : Int = l match {
    case Nil => 1
    case h::t => if (h==0) throw new Zero else h * prod(t)
  }
  try {
    prod(l)
  } catch {
    case _:Zero => 0
  }
}
```

IMPLEMENTING EXCEPTIONS (2)

The handler-stack implementation makes handling very cheap, but incurs cost each time we enter a new handler scope. If exceptions are very rare, this is a bad tradeoff.

- As an alternative, some runtime systems use a **static table** that maps each code address to the address of the statically enclosing handler (if any).
- If an exception occurs, the table is inspected to find the appropriate handler.
- If there is no handler defined in the current routine, the runtime system looks for a handler that covers the return address (in the caller), and so on up the call-stack.
- The deliberate use of exceptions in the previous example would probably be unwise if this implementation approach is used.