Control Flow

February 16, Lecture 9

Loops and Recursion

- Used to execute same thing repeatedly
- Functional uses mostly recursion
- Imperative uses mostly loops,
  - As with statements they are used for their side-effects

- Enumeration-controlled loop: values over a finite set
- Logically-controlled loop: based on monitoring boolean conditions

Loops and Recursion

- Loop in Fortran

equivalent

```
do 10 i = 1, 10, 2
      ...
10  continue
```

```
     i = 1
10       ...
     i = i + 2
     if i <= 10 goto 10
```

- Index, Step size, Continue is a no-op
- Bounds (were originally required to be positive integers)
- Real bounds?

Loops and Recursion

- Loop in Fortran

```
      do 10 i = 1, 10, 2
         ...
10    continue
```

equivalent

```
      i = 1
10       ...
      i = i + 2
      if i <= 10 goto 10
```

- What happens if index changes by a statement?
  - Bug or hard to read when intentional
- Jumping in and out with goto
- What is the exit value of the index?
  - When exit by goto
  - When exit normally (depends on implementation)

Loops and Recursion

- Loop in Modula-2

```
FOR i := first TO last BY step DO
    ...
END
```

- first, last, step can be arbitrarily complex expressions
- What happens if we change index or bounds?
  - Most languages prohibit it
  - Evaluate the bounds exactly once at the beginning
  - Modula-2 is vague, says just don’t change it
  - Pascal explicitly disallows all “threatening” statements (defined in the outside block and disallows: assignments, calls to functions with it as argument, reading from file, part of compound statements)

Loops and Recursion

- Empty loops
- Most languages test in the beginning and make behavior intuitive

```
FOR i := first TO last BY step DO
    ...
END
```

slightly more efficient

```
     r1 := first
     r2 := step
     r3 := last
L1:  if r1 > r3 goto L2
     ...                    -- loop body; use r1 for i
     r1 := r1 + r2
     goto L1
L2:
```

```
     r1 := first
     r2 := step
     r3 := last
     goto L2
L1:  ...                    -- loop body; use r1 for i
     r1 := r1 + r2
L2:  if r1 ≤ r3 goto L1
```

Only if no overflow is guaranteed

Loops and Recursion

- Empty loops
- Most languages test in the beginning and make behavior intuitive

```
        r1 := first
        r2 := step
        r3 := last
L1:     if r1 > r3 goto L2
        . . .                   –– loop body; use r1 for i
        r1 := r1 + r2
        goto L1
L2:
```

- Loop direction (up or down) affects code
- Some languages ask the programmer to specify it
- Other require that step is known at compile time

Loops and Recursion

- Loop direction (up or down) affects code

```
     r1 := first
     r2 := step
     r3 := last
L1:  if r1 > r3 goto L2
     . . .                    -- loop body; use r1 for i
     r1 := r1 + r2
     goto L1
L2:
```

- Some languages ask the programmer to specify it
- Other require that step is known at compile time

Loops and Recursion

- Fortran 77 and 90 use an iteration count variable

```
      r1 := first
      r2 := step
      r3 := max(⌊(last − first + step)/step⌋, 0)        -- iteration count
                -- NB: this calculation may require several instructions.
                -- It is guaranteed to result in a value within the precision
                   of the machine,
                -- but we have to be careful to avoid overflow during its calculation.
      if r3 ≤ 0 goto L2
L1:   ...                          -- loop body; use r1 for i
      r1 := r1 + r2
      r3 := r3 − 1
      if r3 > 0 goto L1
      i := r1
L2:
```

Loops and Recursion

- Index value after loop
- We already said some languages leave it undefined
- Some insist in it being the last value (requires extra machine code)

```
    r1 := 'a'
    r2 := 'z'
    if r1 > r2 goto L3      -- Code improver may remove this test,
                            -- since 'a' and 'z' are constants.
L1: ...                     -- loop body; use r1 for i
    if r1 = r2 goto L2
    r1 := r1 + 1
    -- NB: Pascal step size is always 1 (or −1 if downto)
    goto L1
L2: i := r1
L3:
```

- Undefined because of overflow of integers
- Or because of allowing i to take enumeration types (Pascal)
- Some languages avoid the problem by making the index a local variable
  - Algol, Modula-3, C++
- Jumps into loop are not allowed

Loops and Recursion

- Algol 60 provides a single loop construct that subsumes everything else

$$\begin{aligned}
\textit{for_stmt} &\longrightarrow \texttt{for}\ \textit{id}\ \texttt{:=}\ \textit{for_list}\ \texttt{do}\ \textit{stmt} \\
\textit{for_list} &\longrightarrow \textit{enumerator}\ (\ \texttt{,}\ \textit{enumerator}\ )^{*} \\
\textit{enumerator} &\longrightarrow \textit{expr} \\
&\longrightarrow \textit{expr}\ \texttt{step}\ \textit{expr}\ \texttt{until}\ \textit{expr} \\
&\longrightarrow \textit{expr}\ \texttt{while}\ \textit{condition}
\end{aligned}$$

- Enumerator can be several things
- Each expression is reevaluated at the top of the loop

```
for i := 1, 3, 5, 7, 9 do ...
for i := 1 step 2 until 10 do ...
for i := 1, i + 2 while i < 10 do ...
```

Loops and Recursion

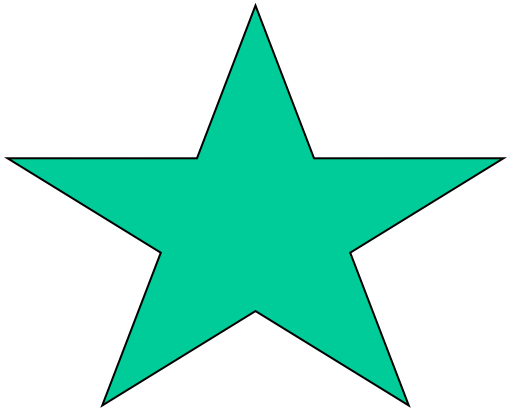
- C's loops are logically controlled
- But they can simulate enumeration controlled
- C's loops is designed in a way that facilitates writing the logically-controlled version of enumeration-controlled loops

```
FOR i := first TO last BY step DO
    ...
END
```

```
for (i = first; i <= last; i += step) {
    ...
}
```

```
i = first;
while (i <= last) {
    ...
    i += step;
}
```

- Checking bounds, overflow, changing indices all become responsibility of programmer



Iterators

- With some exceptions fors use arithmetic values
- We would like to iterate over elements of any other well-defined set
  - Often called containers or collections in object oriented languages

- Languages provide iterators (aka enumerators)
  - Clu, followed by Python, Ruby and C#

- An iterator resembles a subroutine that is permitted to contain **yield** statements each of which produces a loop index calue

Iterators

- Yields: For loops are designed to call an iterator

```
FOR i := first TO last BY step DO
    ...
END
```

would be written as follows in Clu.

```
for i in int$from_to_by(first, last, step) do
    ...
end
```

- When called the iterator calculates the first value index of the loop and it returns it to the main program by a yield
- When executed again the iterator continues where it left off
- At the end the iterator returns empty

Iterators

- Clu pre-order enumerator for binary trees

```
bin_tree = cluster is ..., pre_order, ...          % export list
    node = record [left, right: bin_tree, val: int]
    rep = variant [some: node, empty: null]
    ...
    pre_order = iter(t: cvt) yields(bin_tree)
        tagcase t
            tag empty: return
            tag some(n: node):
                yield(n.val)
                for i: int in pre_order(n.left) do
                    yield(i)
                end
                for i: int in pre_order(n.right) do
                    yield(i)
                end
        end
    end pre_order
    ...
end bin_tree
...
for i: int in bin_tree$pre_order(e) do
    stream$putl(output, int$unparse(i))
end
```

Iterators

- In object oriented languages an iterator is a usual object
- It provides methods for initialization, generation of next value, testing
- Between calls the state is kept by object members
- See Java in book

- C++ uses the overloading mechanism rather than having a special version of for that would interface with iterator objects

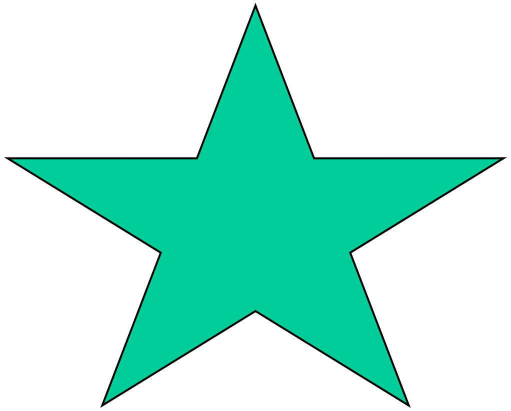
```
tree_node<int> *my_tree = ...
...
for (tree_node<int>::iterator n = my_tree->begin();
                              n != my_tree->end(); ++n) {
    cout << *n << "\n";
}
```

Iterators

- C++ uses the overloading mechanism rather than having a special version of for that would interface with iterator objects

```
tree_node<int> *my_tree = ...
...
for (tree_node<int>::iterator n = my_tree->begin();
                              n != my_tree->end(); ++n) {
    cout << *n << "\n";
}
```

- Dereferencing n using -> and *
- To advance, overloading ++
- The end() method returns a reference to a special iterator that points beyond the end of the set



Iterations in functional languages

- The ability to specify an **inline** function allows writing the body of a loop as a function that takes the index as argument. The function is then passed as argument to the iterator.

```
(define uptoby
  (lambda (low high step f)
    (if (<= low high)
        (begin
         (f low)
         (uptoby (+ low step) high step f))
      '())))
```

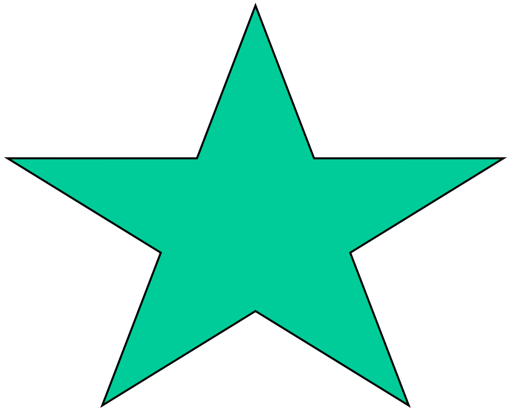
We could then sum the first 50 odd numbers as follows.

```
(let ((sum 0))
  (uptoby 1 100 2
          (lambda (i)
            (set! sum (+ sum i))))
  sum)                                  ⟹ 2500
```

Iterations in functional languages

- Smalltalk
- A square-bracketed block creates a first-class function
- It is then passed as argument to the to: by: do: iterator

```
sum <- 0.
1 to: 100 by: 2 do:
    [:i | sum <- sum + i]
```



Iterating without iterators

- Get away with types and associated functions
- Worse syntax
- No grouping together

```
tree_node *my_tree;
tree_iter ti;
...
for (ti_create(my_tree, &ti); !ti_done(ti); ti_next(&ti)) {
    tree_node *n = ti_val(ti);
    ...
}
ti_delete(&ti);
```

Logically-controlled loops

- Fewer subtleties
- We can imitate them in Fortran 77
- Pre-test and post-test loops

```
do {
    line = read_line(stdin);
} while line[0] != '$';
```

Logically-controlled loops

- Midloop tests (Modula)

```
loop
    statement_list
when condition exit
    statement_list
when condition exit
    ...
end
```

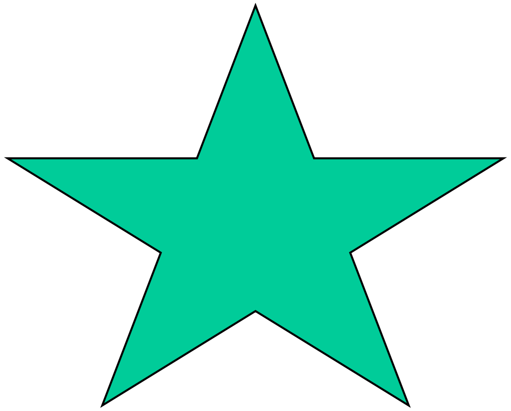
Using this notation, the Pascal construct

```
    while true do begin
        readln(line);
        if all_blanks(line) then goto 100;
        consume_line(line)
    end;
100:
```

can be written as follows in Modula (1).

```
loop
    line := ReadLine;
when AllBlanks(line) exit;
    ConsumeLine(line)
end;
```

- Explicit exit and break statements
- Break with



Recursion

- A natural **iterative** problem

$$\sum_{1 \le i \le 10} f(i)$$

```
typedef int (*int_func) (int);
int summation(int_func f, int low, int high) {
    /* assume low <= high */
    int total = 0;
    int i;
    for (i = low; i <= high; i++) {
        total += f(i);
    }
    return total;
}
```

- A naturally **recursive** problem

$$\begin{array}{c} \mathrm{gcd}(a,b) \\ (\text{positive integers } a, b) \end{array} \equiv \begin{cases} a & \text{if } a = b \\ \mathrm{gcd}(a-b, b) & \text{if } a > b \\ \mathrm{gcd}(a, b-a) & \text{if } b > a \end{cases}$$

```
int gcd(int a, int b) {
    /* assume a, b > 0 */
    if (a == b) return a;
    else if (a > b) return gcd(a-b, b);
    else return gcd(a, b-a);
}
```

Recursion – implementing the other way

- A natural **iterative** problem

$$\sum_{1 \le i \le 10} f(i)$$

```
typedef int (*int_func) (int);
int summation(int_func f, int low, int high) {
    /* assume low <= high */
    if (low == high) return f(low);
    else return f(low) + summation(f, low+1, high);
}
```

- A naturally **recursive** problem

$$\begin{array}{c} \gcd(a, b) \\ \text{(positive integers } a, b) \end{array} \equiv \begin{cases} a & \text{if } a = b \\ \gcd(a-b, b) & \text{if } a > b \\ \gcd(a, b-a) & \text{if } b > a \end{cases}$$

```
int gcd(int a, int b) {
    /* assume a, b > 0 */
    while (a != b) {
        if (a > b) a = a-b;
        else b = b-a;
    }
    return a;
}
```

Efficiency

- It is said that iteration is more efficient that recursion
- That's probably true for **naïve** implementations
- A good compiler will most often generate good code
- Especially true for **tail** recursion, where an additional computation never follows a recursive call
- Recast gcd

```
int gcd(int a, int b) {
    /* assume a, b > 0 */
start:
    if (a == b) return a;
    else if (a > b) {
        a = a-b; goto start;
    } else {
        b = b-a; goto start;
    }
}
```