Lesson 22

• Control Flow
  – Iteration and Iterator
  – Recursion
Overview

• **Expressions evaluation**
  – Evaluation order
  – Assignments

• **Structured and unstructured flow**
  – Goto's
  – Sequencing
  – Selection
  – Iteration and iterators
  – Recursion
Iteration

• An **iterative command** (or *loop*) repeatedly executes a subcommand, which is called the **loop body**.

• Each execution of the loop body is called an **iteration**.

• Classification of iterative commands:
  – **Indefinite iteration**: the number of iterations is not predetermined.
  – **Definite iteration**: the number of iterations is predetermined.

• Note: sequencing, selection and **definite** iteration are not sufficient to make a language Turing complete: either **indefinite** iteration or **recursion** is needed
Iteration

• **Enumeration-controlled loops** (aka *bounded/definite iteration*) repeat a collection of statements a number of times, where in each iteration a *loop index variable* (*counter, control variable*) takes the next value of a set of values specified at the beginning of the loop.

• **Logically-controlled loops** (aka *unbounded/indefinite iteration*) repeat a collection of statements until some Boolean condition changes value in the loop:
  – *Pretest loops* test condition at the begin of each iteration
  – *Posttest loops* test condition at the end of each iteration
  – *Midtest loops* allow structured exits from within loop with exit conditions
Logically-Controlled **Pretest** loops

- *Logically-controlled pretest loops* check the exit condition before the next loop iteration
- Not available in Fortran-77
- Pascal:
  ```pascal
  while <cond> do <stmt>
  ```
  where the condition is a Boolean-typed expression
- C, C++:
  ```c
  while (<expr>) <stmt>
  ```
  where the loop terminates when the condition evaluates to 0, NULL, or false
  - Use `continue` and `break` to jump to next iteration or exit the loop
- Java is similar C++, but condition is restricted to Boolean
Logically-Controlled Posttest Loops

- Logically-controlled posttest loops check the exit condition after each loop iteration
- Not available in Fortran-77
- Pascal:
  
  ```
  repeat <stmt> [; <stmt>]* until <cond>
  ```
  
  where the condition is a Boolean-typed expression and the loop terminates when the condition is true
- C, C++:
  
  ```
  do <stmt> while (<expr>)
  ```
  
  where the loop terminates when the expression evaluates to 0, NULL, or false
- Java is similar to C++, but condition is restricted to Boolean
Logically-Controlled **Midtest** Loops

- Ada supports *logically-controlled midtest loops* check exit conditions anywhere within the loop:

  ```ada
  loop
      <statements>
  exit when <cond>;
      <statements>
  exit when <cond>;
  ...
  end loop
  ```

- Ada also supports labels, allowing exit of outer loops without gotos:

  ```ada
  outer: loop
      ...
      for i in 1..n loop
          ...
          exit outer when a[i]>0;
          ...
      end loop;
  end outer loop;
  ```

- Java allows **labeled breaks** to exit of outer loops
Enumeration-Controlled Loops

General form:

```
for I = start to end by step do
    body
```

- Informal operational semantics...

Some critical issues

- Number of iterations?
- What if \( I, \) start and/or end are modified in body?
- What if step is negative?
- What is the value of I after completion of the iteration?
Enumeration-Controlled Loops

• Some failures on design of enumeration-controlled loops
• Fortran-IV:
  DO 20 i = 1, 10, 2
  ...
  20 CONTINUE
which is defined to be equivalent to
  i = 1
  20  ...
  i = i + 2
  IF i.LE.10 GOTO 20

Problems:
  – Requires positive constant loop bounds (1 and 10) and step size (2)
  – If loop index variable i is modified in the loop body, the number of iterations is changed compared to the iterations set by the loop bounds
  – GOTOs can jump out of the loop and also from outside into the loop
  – The value of counter i after the loop is implementation dependent
  – The body of the loop will be executed at least once (no empty bounds)
Enumeration-Controlled Loops (cont’d)

• Fortran-77:
  – Same syntax as in Fortran-IV, but many dialects support **ENDDO** instead of **CONTINUE** statements
  – Can jump out of the loop, but cannot jump from outside into the loop
  – Assignments to counter \( i \) in loop body are not allowed
  – Number of iterations is determined by
    \[
    \text{max}\left(\left\lfloor \frac{(H - L + S)}{S} \right\rfloor, 0\right)
    \]
    for lower bound \( L \), upper bound \( H \), step size \( S \)
  – Body is not executed when \((H-L+S)/S < 0\)
  – Either integer-valued or real-valued expressions for loop bounds and step sizes
  – Changes to the variables used in the bounds **do not affect** the number of iterations executed
  – Terminal value of loop index variable is the most recent value assigned, which is
    \[
    L + S \times \text{max}\left(\left\lfloor \frac{(H-L+S)}{S} \right\rfloor, 0\right)
    \]
Enumeration-Controlled Loops (cont’d)

• Algol-60 combines logical conditions in *combination loops*:

```plaintext
  for <id> := <forlist> do <stmt>
```

where the syntax of `<forlist>` is

```plaintext
<forlist> ::= <enumerator> [, , , , , ]<enumerator>*
<enumerator> ::= <expr>
  | <expr> step <expr> until <expr>
  | <expr> while <cond>
```

• Not orthogonal: many forms that behave the same:

```plaintext
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 ...
```
Enumeration-Controlled Loops (cont’d)

• Algol-60 combines logical conditions in *combination loops*:

  ```plaintext
  for <id> := <forlist> do <stmt>
  ```

  where the syntax of `<forlist>` is

  ```plaintext
  <forlist> ::= <enumerator> [, <enumerator>]*
  <enumerator> ::= <expr>
      | <expr> step <expr> until <expr>
      | <expr> while <cond>
  ```

• Not orthogonal: many forms that behave the same:

  ```plaintext
  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 ...
  ```
Pascal’s enumeration-controlled loops have simple and elegant design with two forms for *up* and *down*:

```pascal
for <id> := <expr> to <expr> do <stmt>
and
for <id> := <expr> downto <expr> do <stmt>
```

- Can iterate over any discrete type, e.g. integers, chars, elements of a set
- Lower and upper bound expressions are evaluated once to determine the iteration range
- Counter variable cannot be assigned in the loop body
- Final value of loop counter after the loop is undefined
Ada's for loop is much like Pascal's:

```pascal
for <id> in <expr> .. <expr> loop
  <statements>
end loop
```

and

```pascal
for <id> in reverse <expr> .. <expr> loop
  <statements>
end loop
```

- Lower and upper bound expressions are evaluated once to determine the iteration range
- Counter variable has a local scope in the loop body
  - Not accessible outside of the loop
- Counter variable cannot be assigned in the loop body
Enumeration-Controlled Loops (cont’d)

• C and C++ do not have true enumeration-controlled loops, they have combination loops
• A "for" loop is essentially a logically-controlled loop

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

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

• Java’s standard for statement is as in C/C++, but the enhanced for is almost a true enumeration-controlled loop (see later)
Enumeration-Controlled Loops (cont’d)

• Why is C/C++/Java for not enumeration controlled?
  – Assignments to counter \textit{i} and variables in the bounds are allowed, thus
    it is the programmer's responsibility to structure the loop to mimic
    enumeration loops

• Use \texttt{continue} to jump to next iteration
• Use \texttt{break} to exit loop
• C++ and Java also support local scoping for counter variable
  \begin{verbatim}
  for (int i = 1; i <= n; i++) ...
  \end{verbatim}

• In this case the look index variable is not accessible after the loop
Enumeration-Controlled Loops (cont’d)

- Other problems with C/C++ for loops to emulate enumeration-controlled loops are related to the mishandling of bounds and limits of value representations
  - This C program never terminates (do you see why?)
    ```c
    #include <limits.h> // INT_MAX is max int value
    int main()
    {
      int i;
      for (i = 0; i <= INT_MAX; i++)
        printf("Iteration %d\n", i);
    }
    ```
  - This C program does not count from 0.0 to 10.0, why?
    ```c
    float n;
    for (n = 0.0; n <= 10; n += 0.01)
      printf("Iteration %g\n", n);
    ```
Enumeration-Controlled Loops (cont’d)

• How is loop iteration counter overflow handled?
• C, C++, and Java: nope
• Fortran-77
  – Calculate the number of iterations in advance
  – For REAL typed index variables an exception is raised when overflow occurs
• Pascal and Ada
  – Only specify step size 1 and -1 and detection of the end of the iterations is safe
  – Pascal’s final counter value is undefined (may have wrapped)
Iterators

• *Containers (collections)* are aggregates of homogeneous data, which may have various (topo)logical properties
  – Eg: arrays, sets, bags, lists, trees,…

• Common operations on containers requires to iterate on (all of) its elements
  – Eg: search, print, map, …

• *Iterators* provide an abstraction for iterating on containers, through a sequential access to all their elements

• Iterator objects are also called *enumerators* or *generators*
Iterators in Java

• Iterators are supported in the Java Collection Framework: interface `Iterator<T>`
• They exploit generics (as collections do)
• Iterators are usually defined as `nested classes (non-static private member classes)`: each iterator instance is associated with an instance of the collection class
• Collections equipped with iterators have to implement the `Iterable<T>` interface

```java
class BinTree<T> implements Iterable<T> {  
    BinTree<T> left;  
    BinTree<T> right;  
    T val;  
    ...
    // other methods: insert, delete, lookup, ...
    public Iterator<T> iterator() {  
        return new TreeIterator(this);  
    }
}
Iterators in Java (cont’d)

class BinTree<T> implements Iterable<T> {
    ...
    private class TreeIterator implements Iterator<T> {
        private Stack<BinTree<T>> s = new Stack<BinTree<T>>() {
            TreeIterator(BinTree<T> n) {
                if (n.val != null) s.push(n);
            }
            public boolean hasNext() {
                return !s.empty();
            }
            public T next() {
                if (!hasNext()) throw new NoSuchElementException();
                BinTree<T> n = s.pop();
                if (n.right != null) s.push(n.right);
                if (n.left != null) s.push(n.left);
                return n.val;
            }
            public void remove() {
                throw new UnsupportedOperationException();
            }
        }
    }
}
Iterators in Java (cont’d)

• Use of the iterator to print all the nodes of a BinTree:

```java
for (Iterator<Integer> it = myBinTree.iterator();
     it.hasNext();)
{
    Integer i = it.next();
    System.out.println(i);
}
```

• Java provides (since Java 5.0) an enhanced for statement (foreach) which exploits iterators. The above loop can be written:

```java
for (Integer i : myBinTree)
    System.out.println(i);
```

• In the enhanced for, myBinTree must either be an array of integers, or it has to implement Iterable<Integer>

• The enhanced for on arrays is a bounded iteration. On an arbitrary iterator it depends on the way it is implemented.
Iterators in C++

- C++ iterators are associated with a container object and used in loops similar to pointers and pointer arithmetic.
- They exploit the possibility of overloading primitive operations.

```cpp
vector<int> V;
...
for (vector<int>::iterator it = V.begin(); it != V.end(); ++it)
    cout << *n << endl;
```

An in-order tree traversal:

```cpp
tree_node<int> T;
...
for (tree_node<int>::iterator it = T.begin(); it != T.end(); ++it)
    cout << *n << endl;
```
True Iterators

• While Java and C++ use *iterator objects* that hold the state of the iterator, Clu, Python, Ruby, and C# use “true iterators” which are functions that run in “parallel” (in a separate thread) to the loop code to produce elements
  – The *yield* operation in Clu returns control to the loop body
  – The loop returns control to the generator’s last yield operation to allow it to compute the value for the next iteration
  – The loop terminates when the generator function returns
True Iterators (cont’d)

• Generator function for pre-order visit of binary tree in Python
• Since Python is dynamically typed, it works automatically for different types

class BinTree:
    def __init__(self):  # constructor
        self.data = self.lchild = self.rchild = None
    ...
    # other methods: insert, delete, lookup, ...
    def preorder(self):
        if self.data != None:
            yield self.data
        if self.lchild != None:
            for d in self.lchild.preorder():
                yield d
        if self.rchild != None:
            for d in self.rchild.preorder():
                yield d
Iterators in some functional languages

• Exploiting “in line” definitions of functions, the **body** of the iteration can be defined as a function having as argument the loop index

• Then the body is passed as last argument to the **iterator** which is a function realising the loop

• Simple iterator in Scheme and sum of 50 odd numbers:

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

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

• Recursion: subroutines that call themselves directly or indirectly (mutual recursion)
• Typically used to solve a problem that is defined in terms of simpler versions, for example:
  – To compute the length of a list, remove the first element, calculate the length of the remaining list in \( n \), and return \( n+1 \)
  – Termination condition: if the list is empty, return 0
• Iteration and recursion are equally powerful in theoretical sense
  – Iteration can be expressed by recursion and vice versa
• Recursion is more elegant to use to solve a problem that is naturally recursively defined, such as a tree traversal algorithm
• Recursion can be less efficient, but most compilers for functional languages are often able to replace it with iterations
Tail-Recursive Functions

• *Tail-recursive functions* are functions in which no operations follow the recursive call(s) in the function, thus the function returns immediately after the recursive call:

```plaintext
tail-recursive          not tail-recursive
int trfun()             int rfun()
{ ...                { ...
    return trfun();    return 1+rfun();
}                      }
```

• A tail-recursive call could *reuse* the subroutine's frame on the runtime stack, since the current subroutine state is no longer needed
  – Simply eliminating the push (and pop) of the next frame will do

• In addition, we can do more for *tail-recursion optimization*: the compiler replaces tail-recursive calls by jumps to the beginning of the function
Tail-Recursion Optimization

• Consider the GCD function:
  ```
  int gcd(int a, int b) {
    if (a==b) return a;
    else if (a>b) return gcd(a-b, b);
    else return gcd(a, b-a);
  }
  ```

• A good compiler will optimize the function into:
  ```
  int gcd(int a, int b) {
    start:
    if (a==b) return a;
    else if (a>b) { a = a-b; goto start; }
    else { b = b-a; goto start; }
  }
  ```

• Which is just as efficient as the iterative version:
  ```
  int gcd(int a, int b) {
    while (a!=b) {
      if (a>b) a = a-b;
      else b = b-a;
    }
    return a;
  }
  ```
Converting Recursive Functions to Tail-Recursive Functions

- Remove the work after the recursive call and include it in some other form as a computation that is passed to the recursive call.
- For example, the non-tail-recursive function computing \( \sum_{n=low}^{high} f(n) \):

\[
\text{(define summation (lambda (f low high) }
  \text{(if (= low high)}
    \text{(f low)}
    \text{(+ (f low) (summation f (+ low 1) high))))))
\]

- can be rewritten into a tail-recursive function:

\[
\text{(define summation (lambda (f low high subtotal) }
  \text{(if (=low high)}
    \text{(+ subtotal (f low))}
    \text{(summation f (+ low 1) high (+ subtotal (f low))))))}
\]
Example

Here is the same example in C:

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

rewritten into the tail-recursive form:

```c
int summation(int_func f, int low, int high, int subtotal)
{ if (low == high)
    return subtotal+f(low)
  else
    return summation(f, low+1, high, subtotal+f(low));
}
```
When Recursion is Bad

• The Fibonacci function implemented as a recursive function is very inefficient as it takes exponential time to compute:

```
(define fib (lambda (n)
    (cond ((= n 0) 1)
          ((= n 1) 1)
          (else (+ (fib (- n 1)) (fib (- n 2)))))))
```

with a tail-recursive helper function, we can run it in O(n) time:

```
(define fib (lambda (n)
    (letrec ((fib-helper (lambda (f1 f2 i)
                (if (= i n)
                    f2
                    (fib-helper f2 (+ f1 f2) (+ i 1))))))
      (fib-helper 0 1 0)))
```
Continuation-passing Style

- Makes **control** explicit in functional programming (including evaluation order of operands/arguments, returning from a function, etc.)
- A **continuation** is a function representing “the rest of the program” taking as argument the current result
- Functions have an additional (last) argument, which is a continuation
- Primitive functions have to be encapsulated in CPS ones

Encapsulation of primitive operators

```
(define (*& x y k)
  (k (* x y)))
```
Making evaluation order explicit

- Function call arguments must be either variables or lambda expressions (not more complex expressions)

**Direct style**: evaluation order is implicit

```
(define (diag x y)
  (sqrt (+ (* x x) (* y y))))
(diag 3 4)
```

**Continuation-passing style**: evaluation order is explicit

```
(define (diag& x y k)
  (*& x x (lambda (x2)
    (*& y y (lambda (y2)
      (+& x2 y2 (lambda (x2py2)
        (sqrt& x2py2 k))))))))
(diag& 3 4 (lambda (v) v)))
```
Non-tail-recursive functions cause continuation in recursive call to grow

**Direct style**: non-tail-recursive factorial

```
(define (factorial n)
  (if (= n 0)
      1
      (* n (factorial (- n 1))))
```

**Continuation-passing style**: non-tail-recursive factorial

```
(define (factorial& n k)
  (=& n 0 (lambda (b)
    (if b
      (k 1)
      (-& n 1 (lambda (nm1)
        (factorial& nm1 (lambda (f)
          (*& n f k))))))))
```
Tail-recursive functions: continuation in recursive call is identical

**Direct style:** tail-recursive factorial

```
(define (factorial n) (f-aux n 1))
(define (f-aux n a)
  (if (= n 0)
      a ; tail-recursive
      (f-aux (- n 1) (* n a))))
```

**Continuation-passing style:** tail-recursive factorial

```
(define (factorial& n k) (f-aux& n 1 k))
(define (f-aux& n a k)
  (= n 0 (lambda (b)
      (if b
        (k a)
        (- n 1 (lambda (nm1)
            (* n a (lambda (nta)
                (f-aux& nm1 nta k))))))))))
```
On continuation-passing style

- If all functions are in CPS, no runtime stack is necessary: all invocations are *tail-calls*
- The continuation can be replaced or modified by a function, implementing almost arbitrary control structures (exceptions, goto’s, ...)
- Continuations used in denotational semantics for goto’s and other control structure (eg: bind a label with a continuation in the environment)

**Continuation-passing style:** returning *error* to the top-level

```
(define (sqrt n k)
  (if (< n 0)
      'error
      (k (safe-sqrt n))))
```

**Direct style:** the callers should propagate the error along the stack