LECTURE 18

Control Flow

CONTROL FLOW

- Sequencing: the execution of statements and evaluation of expressions is usually in the order in which they appear in a program text.
- Selection (or alternation): a run-time condition determines the choice among two or more statements or expressions.
- Iteration: a statement is repeated a number of times or until a run-time condition is met.
- Procedural abstraction: subroutines encapsulate collections of statements and subroutine calls can be treated as single statements.
- Recursion: subroutines which call themselves directly or indirectly to solve a problem, where the problem is typically defined in terms of simpler versions of itself.
- Concurrency: two or more program fragments executed in parallel, either on separate processors or interleaved on a single processor.
- Non-determinacy: the execution order among alternative constructs is deliberately left unspecified, indicating that any alternative will lead to a correct result.

STRUCTURED AND UNSTRUCTURED FLOW

Unstructured Control Flow: the use of goto statements and statement labels to implement control flow.

- Featured in early versions of Fortran, Cobol, and PL/I.
- Supported by most early Algol-family languages.
- Generally considered “bad”.
- Most can be replaced with structures with some exceptions.
- Java, Clu, Eiffel, and Modula have no goto statement.
- Ada, C#, Fortran90, and C++ only allow gotos in a limited context or to maintain compatibility.

STRUCTURED AND UNSTRUCTURED FLOW

```
      sum = 0;
      i = 0;
TEST: if(i > 10) goto END;
      cout << "i is: " << i << endl;
      sum = sum + i;
      i = i+1;
      goto TEST;
END: cout << "sum is " << sum << endl;
```

STRUCTURED AND UNSTRUCTURED FLOW

Structured Control Flow

- Statement Sequencing.
- Selection with “if-then-else” statements and “switch” statements.
- Iteration with “for” and “while” loop statements.
- Subroutine calls (including recursion).
- All of which promotes “structured programming”.
- Mostly pioneered by Algol 60.

STRUCTURED AND UNSTRUCTURED FLOW

```
sum = 0;
i = 0;
while(i <= 10){
    cout << "i is: " << i << endl;
    sum = sum + i;
    i = i+1;
}
cout << "sum is " << sum << endl;
```

STRUCTURED AND UNSTRUCTURED FLOW

The trend of moving from unstructured to structured flow relied on the ability to find structured solutions to these common uses of goto.

- Mid-loop exit and continuation: replaced with “one-and-a-half” loop constructs, and keywords like “break” and “continue”.
- Early returns from subroutines: most languages now allow an early return rather than explicit goto.
- Multi-level returns: nonlocal gotos replaced by “return-from” in Lisp, catch-and-throw/try-and-except blocks.
- Exceptions: strongly related to multi-level returns. Also replaced by catch/throw blocks.

STRUCTURED AND UNSTRUCTURED FLOW

```
loop
      Put(Item => I, Width => 1);
      exit when I = 10;
      Put(", ");
      I := I + 1;
end loop;
```

```
    sum = 0;
    i = 0;
    while(1){
        if(i > 11)
            break;
        cout << "i is: " << i << endl;
        sum = sum + i;
        i = i+1;
    }
    cout << "sum is " << sum << endl;
```

STRUCTURED AND UNSTRUCTURED FLOW

```
(defun foo (n)
  (dotimes (i 10)
    (dotimes (j 10)
      (when (> (* i j) n)
        (return-from foo (list i j))))))
```

SEQUENCING

- Principle means of controlling execution order in imperative programming (which relies on side-effects for state changes).
- A list of statements in a program text is executed in top-down order.
- A *compound statement* is a delimited list of statements.
- A compound statement is called a *block* when preceded by declarations.
  - C, C++, and Java use { and } to delimit a block.
  - Pascal and Modula use begin ... end.
  - Ada uses declare ... begin ... end.
  - Some languages use whitespace (indentation) to denote blocks.

SELECTION

- Implemented as some variant of an if … then … else structure, as introduced by Algol 60.
- If-else selection statements in C and C++
  - `if (<expr>) <stmt> [else if (<expr>) <stmt>] [else <stmt>]`
  - Condition is a bool, integer, or pointer.
  - Grouping with `{` and `}` is required for statement sequences.
  - Syntax ambiguity is resolved with “an else matches the closest if” rule.
- Conditional expressions, e.g. `if` and `cond` in Lisp and `a?b:c` in C/C++.
- Java syntax is like C/C++, but condition must be Boolean.

SELECTION

Case/switch statements allow an expression to be tested against multiple constants to select statement(s) in one of the arms of the case statement:

In C, C++, and Java:

```
switch (<expr>)
{ case <const>: <statements> break;
  case <const>: <statements> break;
  ...
  default: <statements>
}
```

- A break is necessary to transfer control at the end of an arm to the end of the switch statement.
- Most programming languages support a switch-like statement, but do not require the use of a break in each arm.
- A switch statement can be much more efficient compared to nested if-then-else statements.

ITERATION

- *Enumeration-controlled loops* repeat a collection of statements a number of times, where in each iteration a loop index variable takes the next value of a set of values specified at the beginning of the loop.
- *Logically-controlled loops* repeat a collection of statements until some Boolean condition changes value in the loop.
- Pretest loops test condition at the beginning of each iteration.
  - while loop in C/C++
- Posttest loops test condition at the end of each iteration.
  - do-while loop in C/C++
- Midtest loops allow structured exits from within loop with exit conditions.
  - `for (…) {…; if (…) break; …}` in C/C++

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 <= 10 goto 20
```

ENUMERATION-CONTROLLED LOOPS

Algol-60 combines logical conditions in combination loops:

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

where the syntax of <forlist> is

```
<forlist>    → <enumerator> [, enumerator]*
<enumerator> → <expr>
             | <expr> step <expr> until <expr>
             | <expr> while <cond>
```

Many forms that behave the same:

```
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

In Python:

```
for vowel in ['a', 'e', 'i', 'o', 'u']:
    print vowel
```

ENUMERATION-CONTROLLED LOOPS

C, C++, and Java do not have *true* enumeration-controlled loops. A "for" loop is essentially a logically-controlled loop:

```
for (i = 1; i <= n; i++) ...
```

which iterates `i` from `1` to `n` by testing `i <= n` before the start of each iteration and updating `i` by `1` in each iteration.

Why is this not enumeration controlled? Because the looping behavior is still controlled by a Boolean conditional expression.

- Use continue to jump to next iteration.
- Use break to exit loop
- C++ and Java also support local scoping for counter variable.

ENUMERATION-CONTROLLED LOOPS

Problems with C/C++ for-loops emulating enumeration-controlled loops are related to the mishandling of bounds and limits of value representations.

This C program never terminates (do you see why?)

```
#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?

```
int main()
{ float n;
  for (n = 0.0; n <= 10; n += 0.01)
    printf("Iteration %g\n", n);
}
```

ITERATORS

- *Iterators* are used to iterate over elements of containers such as sets and data structures such as lists and trees.
- Iterator objects are also called *enumerators* or *generators*.
- C++ iterators are associated with a container object and used in loops similar to pointers and pointer arithmetic:

```
vector<int> V;
…
for (vector<int>::iterator it = V.begin(); it != V.end(); ++it)
      // access element of vector
```

ITERATORS IN FUNCTIONAL LANGUAGES

While Java and C++ use iterator objects that hold the state of the iterator, Clu, Python, Ruby, and C# also use generators (“true iterators”) which are functions that run in “parallel” to the loop code to produce elements.

Without side effects, all intermediate states must be maintained to generate the elements in each iteration.

LOGICALLY-CONTROLLED PRETEST LOOPS

- Logically-controlled pretest loops check the exit condition before the next loop iteration.
- Not available in Fortran-77.
- Ada has only one kind of logically-controlled loops: midtest loops.
- Pascal:

```
while <cond> do <stmt>
```

  where the condition is a Boolean-typed expression.
- C, C++:

```
while (<expr>) <stmt>
```

  where the loop terminates when the condition expression 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.

- 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 condition expression evaluates to 0, NULL, or false.

LOGICALLY-CONTROLLED MIDTEST LOOPS

- Ada supports logically-controlled midtest loops which check exit conditions anywhere:

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

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

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

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 as n, and return n+1. If the list is empty, return 0.
- Iteration and recursion are equally powerful in the 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:

Tail-Recursive

```
int trfun()
{ …
  return trfun();
}
```

Not Tail-Recursive

```
int rfun()
{ …
  return rfun()+x;
}
```

- A tail-recursive call could reuse the subroutine's frame on the run-time stack, since the current subroutine state is no longer needed.
- Simply eliminating the push (and pop) of the next frame will do.
- The compiler replaces tail-recursive calls by jumps to the beginning of the function.

TAIL-RECURSIVE FUNCTION

- 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);
}
```

- Can be optimized as:

```
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;
}
```

WHEN RECURSION IS INEFFICIENT

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

```
int fib(n) {
  if (n=1) return 1;
  if (n=2) return 1;
  return fib(n-1) + fib(n-2);
}
```