Chapter 6 – Control Flow

All about expressions and the flow of programs.
Control Flow

- **Control flow** is the order in which a program executes.
- For imperative languages (e.g., C, Java), this is fundamental.
- For other programming paradigms (e.g., functional), the compilers/interpreters take care of ordering.
Expression Evaluation

- An expressions consist of a simple object (e.g., a variable), an operator, or a function applied to a collection of objects and/or operators.
- Expression evaluation is a crucial component of functional languages.
  - Functional languages are very “math-like” and in math the most important concept is evaluating expressions.
Operators

- Operators are used in:
  - **Prefix** notation: operators come first
    
    ```
    (* (+ 1 3) 2 )
    ++a
    ```
  - **Infix** notation: operators in middle
    
    ```
    (1+3) *2
    ```
  - **Postfix** notation: operators last
    
    ```
    ((1 3 +) 2 *)
    a++
    ```
Operators-Precedence

- **Precedence** rules specify the order in which operators of different precedence levels are evaluated.
  - e.g. Multiplication before addition.
- Precedence in **boolean** expressions **very** important
- The phrase “if A<B and C<D” can be read as either:
  - \[ \text{if } (A<B) \text{ and } (C<D) \quad // \quad \text{C} \]
  - \[ \text{if } (A< (B \text{ and } C)) <D \quad // \quad \text{Pascal} \]
Operators--Associativity

- **Associativity** rules specify the order in which operators of the same precedence level are evaluated.
  - Usually they are evaluated “left-to-right”.
- In Fortran, ** associates from **right-to-left**.
  
  \[ x ** y = x^y \]

- Thus \(2**3**4\) is read as \(2^{(3^4)}\) rather than \((2^3)^4\).
Assignment

• The basic operation language is assignment.
• An assignment places a value into a specific memory location.

![Diagram]

• As a result, assignments have longevity and can exist beyond their original context.
Context

- To see the difference between context consider the two following statements.

**Imperative Programming**

```c
int sum(int n){
    int val=0;
    for(int i=0,i<=n;i++){
        val+=i;
    }
    return val;
}
```

**Functional Programming**

```c
int sum(int n){
    if (n<=0) then
        return 0;
    else
        return n+sum(n-1);
}
```

- In the imperative code the value of val changes within the context of sum.
- In the functional code the value of n changes but only between contexts of sum.
Variables

- Two ways to model variables:
  - Value model
  - Reference model
Value Model

- Under the value model variables on the left-hand side (called l-values) of equations denote references, and variables on the right-hand side (called r-values) denote values.
  - Pascal and C++ use this model

```
b = 2;
a = b;
mem(1024) = 2;
mem(2036) = 2;
```

```
2  X
b  a
1024  2036

2  2
b  a
1024  2036
```
Reference Model

- Under the **reference model** variables on both the left and right-hand side **are references**.
  - Chu uses this model.
Expressions: Initialization

- Variable initialization can be *implicit* or *explicit*.
- In **Implicit**, variables are initialized as they are used (e.g., Perl).
  
  ```
  $a = 2+3;
  ```

- In **Explicit**, variables are initialized before they are used (e.g., C, Java).
  
  ```
  int a;
  a=2+3;
  ```
Expressions: Orthogonality

- **Orthogonality** means that features can be used in *any combination* and *the meaning is consistent* regardless of the surrounding features.
Expressions: Orthogonality

begin
    a := if b < c then d else e;
    a := begin f(b); g(c) end;
    g(d);
    2 + 3
end

• Algol 68 is said to be expression-oriented: it has no separate notion of statement.
• Arbitrary expressions can appear in contexts.
Expressions: Orthogonality

- C takes an approach intermediate between Pascal and Algol 68.
  - It distinguishes between statements and expressions, but one of the classes of statement is an “expression statement,” which computes the value of an expression and then throws it away.
  - Allows an expression to appear in any context that would require a statement in most other languages.
- Both Algol 68 and C allow assignments within expressions.
Expressions: Orthogonality

- C lacks a separate Boolean type.
  - In any context that would require a Boolean value in other languages, C accepts an integer.
  - It interprets zero as false; any other value is true.
  - The following constructs are all valid, but have different meaning.

```c
if(a=b){ }

if(a==b){ }
```
Expressions: Complication

- Execution ordering within expression is **complicated by side effects** (and code improvements).

```c
b=1;
int inc(int a) {
    b+=1;
    return a+1;
}
c = (3*b) * inc(b)
```

- If `inc(b)` is evaluated before `(3*b)`, the final value of `c` is 12. If the `(3*b)` is evaluated first, then the value of `c` is 6.
Expression: Multiway Assignments

- Right associativity allows $a = b = c$.
- Clu, ML, Perl, Python, and Ruby allows $a, b = c, d$
  - Here the comma in the right-hand side is *not* the sequencing operator of C.
  - Equal to writing $a = b, c = d$;
- Swap can be implemented this way.
  $a, b = b, a;$ (* swap $a$ and $b$ *)
- Moreover, multiway assignment allows functions to return tuples, as well as single values:
  $a, b, c = f u n c t i o n( d, e, f )$;
Expressions: Short-Circuit

- Expressions may be executed using short-circuit evaluation.

```c
p = my_list;
while (p && p->key != val)
    p = p->next;
```

- If `p = null`, then `p->key` is never checked. Thus, it is “short-circuited”.
Expression: Optimization

- Rearranging operators using mathematical identities.

\[
\begin{align*}
  a &= b/c/d \\
  e &= f/d/c \\
  t &= c \times d \\
  a &= b/t \\
  e &= f/t
\end{align*}
\]
Expressions: Short-Circuit

\[
p = \text{my\_list}; \\
\text{while (p && p->key} \neq \text{val)} \\
p = p->\text{next};
\]

\[
p := \text{my\_list}; \\
\text{while (p<>nil) and} \\
(p^.key <> \text{val}) \text{ do} \\
p := p^.next
\]

Since Pascal does not have short circuiting, this will check both. Thus, if \( p=\text{nil} \), then \( p^.key \) will return an error.
Control Flow Mechanisms

- Sequencing
  - Textual order, precedence in Expression
- Selection
- Iteration
- Procedural abstraction
- Recursion
- Concurrency
- Nondeterminacy
Control Flow: Ordering the Execution of a Program

- Constructs for specifying the execution order:
  1. **Sequencing**: the execution of statements and evaluation of expressions is usually in the order in which they appear in a program text
  2. **Selection** (or alternation): a run-time condition determines the choice among two or more statements or expressions
  3. **Iteration**: a statement is repeated a number of times or until a run-time condition is met
  4. **Procedural abstraction**: subroutines encapsulate collections of statements and subroutine calls can be treated as single statements
  5. **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
  6. **Concurrency**: two or more program fragments executed in parallel, either on separate processors or interleaved on a single processor
  7. **Nondeterminacy**: the execution order among alternative constructs is deliberately left unspecified, indicating that any alternative will lead to a correct result
Sequencing

- **Sequencing** is the order in which statements are to be executed.
- For imperative languages, typically things are executed **in the order they appear**!
  - Usually, executed in top-down order.
- This is not necessarily the case for functional languages!
Sequencing

- A compound statement is a delimited list of statements.
  - A compound statement is called a block when it includes variable declarations.
  - C, C++, and Java use { and } to delimit a block.
  - Pascal and Modula use `begin ... End`.
  - Ada uses `declare ... begin ... End`.
Selection

- **Selection** occurs whenever there is a choice between two or more courses of action.
  - e.g. if/then/else & switch/case.
Selection

- If-then-else selection statements in C and C++:
  - `if (<expr>) <stmt> [else <stmt>]`
  - Condition is a bool, integer, or pointer
  - Grouping with `{ and }` is required for statement sequences in the *then clause* and *else clause*
  - 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.
- Java syntax is like C/C++, but condition must be Boolean.
- Ada syntax supports multiple `elsif`'s to define nested conditions:
  - `if <cond> then
    <statements>
  elsif <cond> then
  ... 
  else
  <statements>
  end if


If-Then-Else

- For complex conditionals two ways to evaluate:
  - Evaluate and put into register (works but slow).
  - Use short-circuiting in assembly to have jump codes (fast and awesome).

```plaintext
if ((A > B) && (C > D)) or (E != F)
then {then_clause}
else {else_clause}
```
If-Then-Else

\[
\begin{align*}
\text{r1} & := \text{A} \\
\text{r2} & := \text{B} \\
\text{r1} & := \text{r1} > \text{r2} \\
\text{r2} & := \text{C} \\
\text{r3} & := \text{D} \\
\text{r2} & := \text{r2} > \text{r3} \\
\text{r1} & := \text{r1} \& \text{r2} \\
\text{r2} & := \text{E} \\
\text{r3} & := \text{F} \\
\text{r2} & := \text{r2} \neq \text{r3} \\
\text{r1} & := \text{r1} \mid \text{r2} \\
\text{if } \text{r1} & = 0 \text{ goto } L2 \\
L1: & \text{ then\_clause} \\
\text{goto } L3 \\
L2: & \text{ else\_clause} \\
L3: &
\end{align*}
\]

**Regular**

\[
\begin{align*}
\text{r1} & := \text{A} \\
\text{r2} & := \text{B} \\
\text{if } \text{r1} & \leq \text{r2} \text{ goto } L4 \\
\text{r1} & := \text{C} \\
\text{r2} & := \text{D} \\
\text{if } \text{r1} & > \text{r2} \text{ goto } L1 \\
L4: & \text{ r1} := \text{E} \\
\text{r2} & := \text{F} \\
\text{if } \text{r1} & = \text{r2} \text{ goto } L2 \\
L1: & \text{ then\_clause} \\
\text{goto } L3 \\
L2: & \text{ else\_clause} \\
L3: &
\end{align*}
\]

**Short Circuit**
Switch-Case

- Not only is it more convenient in certain circumstances, but it's more efficient!
  - Because you can implement a case-switch as an indexed table rather than a very long piece of assembly code.
Switch-Case

- Case/switch statements are different from if-then-else statements in that an expression can be tested against multiple constants to select statement(s) in one of the arms of the case statement:
  - 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 much more efficient compared to nested if-then-else statements.
Unstructured Flow: The GOTO statement

- Assembly languages controls flow via conditional and unconditional jumps.
  
  ```
  JMP  30
  ...
  30: ADD r1, #3
  ```

- Higher level languages have similar statement.
  
  ```
  goto stop_point;
  ...
  stop_point:
  cout<<“stopping”;
Unstructured Flow: The GOTO statement

- **GOTO** has long been considered “evil” (since the 70s)
  - Spaghetti code.
  - Difficult to debug.
Structured Flow

- Structured flow (i.e., if-then-else, loops, etc...) provide the same expressive power
- Bohm & Jacopini in 1964 proved that sequencing, selection, and iteration can effectively emulate \textit{gotos}.
- However, sometimes \textit{gotos} are more convenient.
  - Jumping out of deep \texttt{for} loops.
Special Cases--Perl, `last`

```perl
while ($d++){  
  if($d>=37) { $res = "done"; last; }  
  $sum +=$d;  
}
#last jumps to here
```

Special Cases--Perl, next

while ($d<37){
    $d++;  
    if (($d%5)==1) {next};
    $sum +=$d;

#next jumps to here
}

Special Cases--Perl, redo

while ($d++){
    #redo jumps to here
    $r = random($d);
    if($r>100) {redo};
    $sum +=$r*$d;
}

Special Cases

- Break and continue.
- Early subroutine returns.
- Exceptions and Errors.
Iteration and Recursion

- These two control flow mechanisms allow a computer to perform the same set of operations repeatedly.
- **Imperative languages** mainly rely on **iterations**.
- **Functional languages** mainly rely on **recursion**.
Iteration

• Iteration usually takes the form of loops
• Two principal varieties:
  ▫ **Enumeration** controlled loops: iterates through an enumerated set.
  ▫ **Logically** controlled loops: iterates while (or until) a logical statement is true.
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 begin 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++`
Iteration

**Enumeration**

```java
for (int i=0;i<=10;i++){
    ...
}
```

**Logical**

```java
int i = 0;
while (i<=10){
    ...
    i++;
}
```
Iteration: Enumeration-Controlled Loops

- **Fortran** enumeration-controlled loops are comprised of three elements:
  - Index variable
  - Step size and bounds
  - Body of the loop

```fortran
  do 10 i=1, 10, 2
    ...
  10: continue
```
Problems with Fortran

- Loop boundaries must be integer.
- Index variable can change within body of loop.
- GOTO statements may jump in and out of loop.
- The value of \( i \) after termination of the loop is implementation dependent.
- The test of the loop takes place at the end so body is executed at least once.

```fortran
DO 10 i=1,20,2
  ...
  10: continue

Equivalent to

i = 1
10 ... 
i = i + 2
IF i.LE.20 GOTO 10
```
Iteration: 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>
  ```

- Not orthogonal: 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 ...
  ```
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 \( i \leq n \) before the start of each iteration and updating \( i \) by 1 in each iteration.

Why is this not enumeration controlled?

- Assignments to counter \( i \) and variables in the bounds are allowed, thus it is the programmer's responsibility to structure the loop to mimic enumeration loops.

Use `continue` to jump to next iteration.

Use `break` to exit loop.

C++ and Java also support local scoping for counter variable:

```
for (int i = 1; i <= n; i++) ...
```
Iteration: Enumeration-Controlled Loops

- 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
    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
    main()
    {
        float n;
        for (n = 0.0; n <= 10; n += 0.01)
            printf("Iteration %f\n", n);
    }
    ```
Iteration: Empty conditions

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

r1:=first
r2:=step
r3:=last
L1: if r1>r3 goto L2
...
rl:=rl+r2
goto L1

L2:

Only works if first + (\lfloor(last-first)/step\rfloor+1)step is at most the largest integer.

Slower, due to 2 branches.
One conditional and one non-conditional.
Backwards loop

\[
\begin{align*}
  r1 &= \text{first} \\
  r2 &= \text{step} \\
  r3 &= \text{max}\left\lfloor \frac{(\text{last} - \text{first} + \text{step})}{\text{step}} \right\rfloor, 0 \\
  \text{if } r3 &\leq 0 \text{ goto L2} \\
  \text{L1: ...} \\
  r1 &= r1 + r2 \\
  r3 &= r3 - 1 \\
  \text{if } r3 &> 0 \text{ goto L1} \\
  i &= r1 \\
  \text{L2:}
\end{align*}
\]

Treats decrement as increment.
Access to Index Outside the Loop

```pascal
var c: 'a' .. 'z';
FOR c:= 'a' TO 'z' DO
BEGIN
    ....
END;
```

```pascal
r1:= 'a'
r2:= 'z'
if r1>r2 goto L3
L1: ...
if r1=r2 goto L2
r1:=r1 +1
goto L1
L2: c:=r1
L3:
```

Preserves \( c \) after loop.
Iteration: Iterators

- **Iterators** are used to **enumerate the elements of any well-defined set**.
  - Moreover, they **generalize arithmetic sequences**.
  - Elements of containers such as sets and data structures such as lists and trees.
- **Iterator objects** are also called *enumerators* or *generators*.
- In previous examples, iteration was always over the **elements of an arithmetic sequences**.
Iteration: Iterators

- C++ iterators are associated with a container object and used in loops similar to pointers and pointer arithmetic:

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

An in-order tree traversal:

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

- Both Java and C++ allow for **iterators as objects**.

  hasNext(); // return true if next element

  next(); // Returns next element

  remove(); // Gets rid of the last element (optional)
Iteration: Iterators in Functional Languages

• Iterators typically need special loops to produce elements one by one, e.g. in Clu:

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

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

• Without side effects, all intermediate state must be maintained to generate the elements in each iteration.
Iterator Perl

```perl
@colors = ("red", "green", "blue")
foreach $elt (@colors){
    print $elt, ", ";
}
print "are the colors we have\n";

@colors = ("red", "green", "blue")
foreach (@colors){ #use $_
    print $_, ", ";
}
print "are the colors we have\n";
```
Iteration: Logically-controlled Loops

- Three types:
  - **Post-test**: Test at end.
  - **Midtest**: Test in middle.
  - **Pre-test**: Test at beginning.
Iteration: Logically-controlled Loops

<table>
<thead>
<tr>
<th>Post-test</th>
<th>Midtest</th>
<th>Pre-test</th>
</tr>
</thead>
<tbody>
<tr>
<td>repeat</td>
<td>for(;;){</td>
<td>while (i==false)</td>
</tr>
<tr>
<td></td>
<td>...</td>
<td></td>
</tr>
<tr>
<td></td>
<td>until i==true;</td>
<td></td>
</tr>
<tr>
<td></td>
<td>}</td>
<td></td>
</tr>
<tr>
<td></td>
<td>}</td>
<td></td>
</tr>
<tr>
<td></td>
<td>}</td>
<td></td>
</tr>
</tbody>
</table>
Logically-Controlled Pretest loops

- *Logically-controlled pretest loops* check the exit condition before the next loop iteration.
- Not available Fortran-77.
- Ada has only one kind of logically-controlled loops: midtest loops.
- 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 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
```
Logically-Controlled Midtest Loops

• 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 outer loop;
Logically-Controlled Posttest Loops

- *Logically-controlled posttest loops* check the exit condition after each loop iteration.
  - Pascal:
    ```pascal
    repeat <stmt> [; <stmt>]* until <cond>
    ```
    where the condition is a Boolean-typed expression and the loop terminates when the condition is true (post-test).
  - C, C++:
    ```c
    do <stmt> while (<expr>)
    ```
    where the loop terminates when the expression evaluates to 0, NULL, or false (post-test).
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.
• 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.
Recursion

- Recursion requires no special syntax.
- Iteration and recursion are equally powerful in theoretical sense
  - Iteration can be expressed by recursion and vice versa

\[
\gcd(a, b) = \begin{cases} 
  a & \text{if } a = b \\
  \gcd(a - b, b) & \text{if } a > b \\
  \gcd(a, b - a) & \text{if } a < b 
\end{cases}
\]
Recursion

```c
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);
}
```

Iteration

```c
int gcd(int a, int b) {
    while (a != b) {
        if (a > b)
            a = a - b;
        else
            b = b - a;
    }
    return a;
}
```
Tail Recursion

- **Tail recursion** is when no computation occurs after the recursive statement.
  - Tail recursive method has the recursive call as the last statement in the method.
  - Recursive methods that are not tail recursive are called non-tail recursive.
- The advantage of tail recursion is that **space can be reused**.
Tail Recursion

• Is the factorial method a tail recursive method?

```java
int fact(int x){
    if (x==0)
        return 1;
    else
        return x*fact(x-1);
}
```

• When returning back from a recursive call, there is still one pending operation, multiplication.
• Therefore, factorial is a non-tail recursive method.
Tail Recursion

• Is this method tail recursive?

```java
void tail(int i) {
    if (i>0) {
        system.out.print(i+""");
        tail(i-1);
    }
}
```

• It is tail recursive!
Tail Recursion

• Is the following program tail recursive?

```java
void prog(int i) {
    if (i>0) {
        prog(i-1);
        System.out.print(i+"" );
        prog(i-1);
    }
}
```

• No, because there is an earlier recursive call, other than the last one.

• In tail recursion, the recursive call should be the last statement, and there should be no earlier recursive calls whether direct or indirect.
Tail Recursion

Tail Recursive Code

```c
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);
}
```

Call Stack
Tail Recursion

- 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
- 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

Not Tail Recursive Code

```c
int gcd(int a, int b) {
    int x;
    if (a == b)
        x = a;
    else if (a > b)
        x = gcd(a - b, b);
    else
        x = gcd(a, b);
    return x;
}
```

Call Stack

![Call Stack Diagram]
Advantage of Tail Recursive Method

- Tail Recursive methods are easy to convert to iterative.

```java
void tail(int i) {
    if (i>0) {
        System.out.println(i++);
        tail(i-1)
    }
}
```

```java
void iterative(int i) {
    for (;i>0;i--)
        System.out.println(i++);
}
```

- Smart compilers can detect tail recursion and convert it to iterative to optimize code.
- Used to implement loops in languages that do not support loop structures explicitly (e.g. prolog).
Converting Non-tail to Tail Recursive

- A non-tail recursive method can be converted to a tail-recursive method by means of an "auxiliary" parameter used to form the result.
- The technique is usually used in conjunction with an "auxiliary" function.
  - This is simply to keep the syntax clean and to hide the fact that auxiliary parameters are needed.

```c
int fact_aux(int n, int result) {
    if (n == 1)
        return result;
    return fact_aux(n - 1, n * result);
}

int fact(n) {
    return fact_aux(n, 1);
}
```
Tail-Recursion Optimization

- Consider the GCD function:

```c
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:

```c
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; }
}
```
Tail-Recursion Optimization

• Just as efficient as the iterative version:

```c
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:

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

- Nondeterministic constructs make choices between alternative deliberately unspecified.
- This mechanism is specially useful in concurrent programs.
## Nondeterminacy

<table>
<thead>
<tr>
<th>Client</th>
<th>Server</th>
</tr>
</thead>
<tbody>
<tr>
<td>loop</td>
<td>loop</td>
</tr>
<tr>
<td>toss coin</td>
<td>receive Read:</td>
</tr>
<tr>
<td>if heads,</td>
<td>send data</td>
</tr>
<tr>
<td>send read to server</td>
<td>OR</td>
</tr>
<tr>
<td>if tails,</td>
<td>receive write:</td>
</tr>
<tr>
<td>send write to server</td>
<td>send reply</td>
</tr>
</tbody>
</table>