CONTROL FLOW

PRINCIPLES OF PROGRAMMING LANGUAGES

Norbert Zeh
Winter 2018
Dalhousie University
The successful programmer thinks in terms of basic principles of control flow, not in terms of syntax!

The principal categories of control flow mechanisms are:

- Sequencing
- Selection or alternation
- Iteration
- Procedural abstraction (next topic)
- Recursion
- Concurrency
- Exception handling and speculation (next topic)
- Non-determinism
Order of evaluation may influence result of computation.
Order of evaluation may influence result of computation.

**Purely functional languages:**

- Computation *is* expression evaluation.
- The only effect of evaluation is the returned value—no side effects.
- Order of evaluation of subexpressions is irrelevant.
Order of evaluation may influence result of computation.

**Purely functional languages:**

- Computation *is* expression evaluation.
- The only effect of evaluation is the returned value—no side effects.
- Order of evaluation of subexpressions is irrelevant.

**Imperative languages:**

- Computation is a series of changes to the values of variables in memory.
- This is “computation by side effect”.
- The order in which these side effects happen may determine the outcome of the computation.
- There is usually a distinction between an expression and a statement.
Assignment is the simplest (and most fundamental) type of side effect a computation can have.

- Very important in imperative programming languages
- Much less important in declarative programming languages
Assignment is the simplest (and most fundamental) type of side effect a computation can have.

- Very important in imperative programming languages
- Much less important in declarative programming languages

**Syntactic differences** (Important to know, semantically irrelevant):

- **A = 3** FORTRAN, PL/1, SNOBOL4, C, C++, Java
- **A := 3** Pascal, Ada, Icon, ML, Modula-3, ALGOL 68
- **A <- 3** Smalltalk, Mesa, APL
- **A =. 3** J
- **3 -> A** BETA
- **MOVE 3 TO A** COBOL
- **(SETQ A 3)** LISP
Expressions that denote values are referred to as \textit{r-values}.

Expressions that denote memory locations are referred to as \textit{l-values}.
Expressions that denote values are referred to as **r-values**.

Expressions that denote memory locations are referred to as **l-values**.

In most languages, the meaning of a variable name differs depending on the side of an assignment statement it appears on:

\[
\begin{align*}
\text{d} &= \text{a} \\
\text{a} &= \text{b} + \text{c};
\end{align*}
\]
Expressions that denote values are referred to as **r-values**.

Expressions that denote memory locations are referred to as **l-values**.

In most languages, the meaning of a variable name differs depending on the side of an assignment statement it appears on:

- On the right-hand side, it refers to the variable’s value—it is used as an **r-value**.

```
    d = a;  // a’s value

    a = b + c;  // a’s memory location
```
Expressions that denote values are referred to as r-values.

Expressions that denote memory locations are referred to as l-values.

In most languages, the meaning of a variable name differs depending on the side of an assignment statement it appears on:

- On the right-hand side, it refers to the variable’s value—it is used as an r-value.
- On the left-hand side, it refers to the variable’s location in memory—it is used as an l-value.

```
d = a;
```

```
a = b + c;
```

\( a \)’s value

\( a \)’s memory location
Some languages explicitly distinguish between l-values and r-values:

- **BLISS**: \( X := .X + 1 \)
- **ML**: \( X := !X + 1 \)
Some languages explicitly distinguish between l-values and r-values:

- BLISS: \( X := .X + 1 \)
- ML: \( X := !X + 1 \)

In some languages, a function can return an l-value (e.g., ML or C++):

```c
int a[10];

int &f(int i) {
    return a[i % 10];
}

void main() {
    for (int i = 0; i < 100; ++i)
        f(i) = i;
}
```
Value model

Assignment copies the value.
Value model
Assignment copies the value.

Reference model
• A variable is always a reference.
• Assignment makes both variables refer to the same memory location.
VARIABLE MODELS

Value model
Assignment copies the value.

Reference model

• A variable is always a reference.
• Assignment makes both variables refer to the same memory location.

Distinguish between:
• Variables referring to the same object and
• Variables referring to different but identical objects.
## VARIABLE MODELS

<table>
<thead>
<tr>
<th>Value model</th>
<th>Assignment copies the value.</th>
</tr>
</thead>
</table>
| Reference model | • A variable is always a reference.  
• Assignment makes both variables refer to the same memory location. |

Distinguish between:

• Variables referring to the same object and  
• Variables referring to different but identical objects.

**An example:** Java

• Value model for built-in types  
• Reference model for classes
\[ b = 2; \quad c = b; \quad a = b + c; \]

Value model

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>b</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>c</td>
<td>2</td>
<td></td>
</tr>
</tbody>
</table>

Reference model

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>b</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>c</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
int a = 5;
int b = a;
b += 10;
System.out.println("a = " + a);
System.out.println("b = " + b);

Output:
int a = 5;
int b = a;
b += 10;
System.out.println("a = " + a);
System.out.println("b = " + b);

Output:
a = 5
b = 15
int a = 5;
int b = a;
b += 10;
System.out.println("a = " + a);
System.out.println("b = " + b);

Output:

a = 5
b = 15

Obj a = new Obj();
Obj b = a;
b.change();
System.out.println(a == b);

Output:

true

String a = "hi ";
String b = a;
b += "world";
System.out.println("a = " + a);
System.out.println("b = " + b);

Output:

a = hi 

b = hi world

StringBuffer a = new StringBuffer();
StringBuffer b = a;
b.append("This is b's value.");
System.out.println("a = " + a);
System.out.println("b = " + b);

Output:

a = This is b's value
b = This is b's value
int a = 5;
int b = a;
b += 10;
System.out.println("a = " + a);
System.out.println("b = " + b);

**Output:**
a = 5
b = 15

Obj a = new Obj();
Obj b = a;
b.change();
System.out.println(a == b);

**Output:**
true

String a = "hi ";
String b = a;
b += "world";
System.out.println("a = " + a);
System.out.println("b = " + b);

**Output:**
a = hi 
b = hi world

StringBuffer a = new StringBuffer();
StringBuffer b = a;
b.append("This is b's value.");
System.out.println("a = " + a);
System.out.println("b = " + b);

**Output:**
a = This is b's value
b = This is b's value
```java
int a = 5;
int b = a;
b += 10;
System.out.println("a = " + a);
System.out.println("b = " + b);

Output:

a = 5
b = 15

Obj a = new Obj();
Obj b = a;
b.change();
System.out.println(a == b);

Output:

true
```

```java
String a = "hi ";
String b = a;
b += "world";
System.out.println("a = " + a);
System.out.println("b = " + b);

Output:

```
```
```java
int a = 5;
int b = a;
b += 10;
System.out.println("a = "+ a);
System.out.println("b = "+ b);

Output:
a = 5
b = 15

Obj a = new Obj();
Obj b = a;
b.change();
System.out.println(a == b);

Output:
true

String a = "hi ";
String b = a;
b += "world";
System.out.println("a = "+ a);
System.out.println("b = "+ b);

Output:
a = hi
b = hi world
```
int a = 5;
int b = a;
b += 10;
System.out.println("a = " + a);
System.out.println("b = " + b);

Output:
a = 5
b = 15

Obj a = new Obj();
Obj b = a;
b.change();
System.out.println(a == b);

Output:
true

String a = "hi ";
String b = a;
b += "world";
System.out.println("a = " + a);
System.out.println("b = " + b);

Output:
a = hi
b = hi world

StringBuffer a = new StringBuffer();
StringBuffer b = a;
b.append("This is b's value.");
System.out.println("a = " + a);
System.out.println("b = " + b);

Output:
```java
int a = 5;
int b = a;
b += 10;
System.out.println("a = " + a);
System.out.println("b = " + b);

Output:
a = 5
b = 15

Obj a = new Obj();
Obj b = a;
b.change();
System.out.println(a == b);

Output:
true
```

```java
String a = "hi ";
String b = a;
b += "world";
System.out.println("a = " + a);
System.out.println("b = " + b);

Output:
a = hi
b = hi world
```

```java
StringBuffer a = new StringBuffer();
StringBuffer b = a;
b.append("This is b's value.");
System.out.println("a = " + a);
System.out.println("b = " + b);

Output:
a = This is b's value
b = This is b's value
```
It is usually unwise to write expressions where a side effect of evaluating an operand is to change another operand used in the same expression.

Some languages explicitly forbid side effects in expression operands.
It is usually unwise to write expressions where a side effect of evaluating an operand is to change another operand used in the same expression.

Some languages explicitly forbid side effects in expression operands.

Possible problems:
It is usually unwise to write expressions where a side effect of evaluating an operand is to change another operand used in the same expression.

Some languages explicitly forbid side effects in expression operands.

Possible problems:

- Evaluation order is often left to the compiler (i.e., undefined in the language specification). Thus, such side effects may lead to unexpected results.
It is usually unwise to write expressions where a side effect of evaluating an operand is to change another operand used in the same expression.

Some languages explicitly forbid side effects in expression operands.

**Possible problems:**

- Evaluation order is often left to the compiler (i.e., undefined in the language specification). Thus, such side effects may lead to unexpected results.

- Evaluation order impacts register allocation, instruction scheduling, etc. By fixing a particular evaluation ordering, some code improvements may not be possible. This impacts performance.
for(i = m = M = 1; N - ++i; M = m + (m = M));

What does this code compute?
for(i = m = M = 1; N - ++i; M = m + (m = M));

What does this code compute?

The answer depends on the evaluation order of the two subexpressions of $M = m + (m = M)$. 
for(i = m = M = 1; N - ++i; M = m + (m = M));

What does this code compute?

The answer depends on the evaluation order of the two subexpressions of $M = m + (m = M)$.

**Probably intended**

<table>
<thead>
<tr>
<th>N</th>
<th>m</th>
<th>M</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>4</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>5</td>
<td>3</td>
<td>5</td>
</tr>
<tr>
<td>6</td>
<td>5</td>
<td>8</td>
</tr>
</tbody>
</table>

$M = F_N$ (Nth Fibonacci number)
for(i = m = M = 1; N - ++i; M = m + (m = M));

What does this code compute?

The answer depends on the evaluation order of the two subexpressions of
M = m + (m = M).

<table>
<thead>
<tr>
<th>Probably intended</th>
<th>Actual</th>
</tr>
</thead>
<tbody>
<tr>
<td>N  m  M</td>
<td>N  m  M</td>
</tr>
<tr>
<td>2  1  1</td>
<td>2  1  1</td>
</tr>
<tr>
<td>3  1  2</td>
<td>3  1  2</td>
</tr>
<tr>
<td>4  2  3</td>
<td>4  2  4</td>
</tr>
<tr>
<td>5  3  5</td>
<td>5  4  8</td>
</tr>
<tr>
<td>6  5  8</td>
<td>6  8  16</td>
</tr>
</tbody>
</table>

\[ M = F_N \text{ (Nth Fibonacci number)} \]

\[ M = 2^{N-2} \]
(and \ a \ b): If \ a \ is \ false, \ b \ has \ no \ effect \ on \ the \ value \ of \ the \ whole \ expression.
(or \ a \ b): If \ a \ is \ true, \ b \ has \ no \ effect \ on \ the \ value \ of \ the \ whole \ expression.
(and a b): If a is false, b has no effect on the value of the whole expression.
(or a b): If a is true, b has no effect on the value of the whole expression.

Short-circuit evaluation

If the value of the expression does not depend on b, the evaluation of b is skipped.
(and a b): If a is false, b has no effect on the value of the whole expression.
(or a b): If a is true, b has no effect on the value of the whole expression.

**Short-circuit evaluation**

If the value of the expression does not depend on b, the evaluation of b is skipped.

This is useful, both in terms of optimization and semantically.
(and a b): If a is false, b has no effect on the value of the whole expression.
(or a b): If a is true, b has no effect on the value of the whole expression.

**Short-circuit evaluation**

If the value of the expression does not depend on b, the evaluation of b is skipped.

This is useful, both in terms of optimization and semantically.

Some languages provide both regular and short-circuit versions of Boolean operators.

Ada:

- and vs and then
- or vs or else
COMMON IDIOMS ENABLED BY SHORT-CIRCUIT EVALUATION

Checking for NULL pointers in C:

```c
while (p != NULL && p->e != val) {
    p = p->next;
}
```
Checking for NULL pointers in C:

```c
while (p != NULL && p->e != val) {
    p = p->next;
}
```

Exit on failure in Perl:

```perl
open(F, "file") or die;
```
COMMON IDIOMS ENABLED BY SHORT-CIRCUIT EVALUATION

Checking for NULL pointers in C:

```c
while (p != NULL && p->e != val) {
    p = p->next;
}
```

Exit on failure in Perl:

```perl
open(F, "file") or die;
```

Short-circuit and as if-statement in Perl or shell scripts:

```perl
if (x > max) then max = x;
```

becomes

```perl
(x > max) && max = x;
```
In imperative programming languages, sequencing comes naturally, without a need for special syntax to support it.

Mixed imperative/function languages (LISP, Scheme, ...) often provide special constructs for sequencing.
In imperative programming languages, sequencing comes naturally, without a need for special syntax to support it.

Mixed imperative/function languages (LISP, Scheme, ...) often provide special constructs for sequencing.

**Issue:** What’s the value of a sequence of expressions/statements?
In imperative programming languages, sequencing comes naturally, without a need for special syntax to support it.

Mixed imperative/function languages (LISP, Scheme, ...) often provide special constructs for sequencing.

**Issue:** What’s the value of a sequence of expressions/statements?

- The value of the last subexpression (most common)
  
  
  ```
  C: a = 4, b = 5;
  ```
  
  → 5

- The value of the first subexpression
  
  ```
  LISP: (prog1 (setq a 4) (setq b 5))
  ```
  
  → 4

- The value of the second subexpression
  
  ```
  LISP: (prog2 (setq a 4) (setq b 5) (setq c 6))
  ```
  
  → 5
In imperative programming languages, sequencing comes naturally, without a need for special syntax to support it.

Mixed imperative/function languages (LISP, Scheme, ...) often provide special constructs for sequencing.

**Issue:** What’s the value of a sequence of expressions/statements?

- The value of the last subexpression (most common)
  
  C: \( \text{a} = 4, \text{b} = 5; \)
  \[ \text{LISP: (progn (setq a 4) (setq b 5))} \]
  \[ \implies 5 \]
In imperative programming languages, sequencing comes naturally, without a need for special syntax to support it.

Mixed imperative/function languages (LISP, Scheme, ...) often provide special constructs for sequencing.

**Issue:** What’s the value of a sequence of expressions/statements?

- The value of the last subexpression (most common)
  
  C: \[a = 4, b = 5;\]  
  LISP: \((\text{progn} \ (\text{setq} \ a \ 4) \ (\text{setq} \ b \ 5))\) 
  \[\Rightarrow 5\]

- The value of the first subexpression
  
  LISP: \((\text{prog1} \ (\text{setq} \ a \ 4) \ (\text{setq} \ b \ 5))\) 
  \[\Rightarrow 4\]
In imperative programming languages, sequencing comes naturally, without a need for special syntax to support it.

Mixed imperative/function languages (LISP, Scheme, ...) often provide special constructs for sequencing.

**Issue:** What’s the value of a sequence of expressions/statements?

- The value of the last subexpression (most common)
  - C: `a = 4, b = 5;
  - LISP: `(progn (setq a 4) (setq b 5))`  
    
- The value of the first subexpression
  - LISP: `(prog1 (setq a 4) (setq b 5))`  
    
- The value of the second subexpression
  - LISP: `(prog2 (setq a 4) (setq b 5) (setq c 6))`  
    
"→ 5"
Use of `goto` is bad programming practice if the same effect can be achieved using different constructs.
Use of `goto` is bad programming practice if the same effect can be achieved using different constructs.

Sometimes, it is unavoidable:

- Break out of a loop
- Break out of a subroutine
- Break out of a deeply nested context
Use of **goto** is bad programming practice if the same effect can be achieved using different constructs.

Sometimes, it is unavoidable:

- Break out of a loop
- Break out of a subroutine
- Break out of a deeply nested context

Many languages provide alternatives:

- One-and-a-half loop
- **return** statement
- Structured exception handling
Standard if-then-else statement:

if cond then this
  else that
SELECTION (ALTERNATION)

Standard if-then-else statement:

```plaintext
if cond then this
  else that
```

Multi-way if-then-else statement:

```plaintext
if    cond1 then option1
elsif cond2 then option2
elsif cond3 then option3
...
  else default action
```
Standard if-then-else statement:

if cond then this
  else that

Multi-way if-then-else statement:

if cond1 then option1
elsif cond2 then option2
elsif cond3 then option3
...
  else default action

Switch statement:

switch value of
  case pattern1: option1
  case pattern2: option2
  ...
default: default action
Switch statements are a special case of if/then/elsif/else statements.
Switch statements are a special case of if/then/elsif/else statements.

Principal motivation:
Switch statements are a special case of if/then/elsif/else statements.

Principal motivation: Generate more efficient code!
Switch statements are a special case of if/then/elsif/else statements.

**Principal motivation:** Generate more efficient code!

Compiler can use different methods to generate efficient code:

- Sequential testing
- Binary search
- Hash table
- Jump table
if i == 1:
    option1()
elsif i in [2, 7]:
    option2()
elsif i in [3, 4, 5]:
    option3()
elsif i == 10:
    option4()
else:
    default_action()
if i == 1:
    option1()
elsif i in [2, 7]:
    option2()
elsif i in [3, 4, 5]:
    option3()
elsif i == 10:
    option4()
else:
    default_action()

Assume i is stored in register R1.

if R1 != 1 goto L1
    call option1
goto L6
L1: if R1 == 2 goto L2
    if R1 != 7 goto L3
L2: call option2
goto L6
L3: if R1 < 3 goto L4
    if R1 > 5 goto L4
    call option3
goto L6
L4: if R1 != 10 goto L5
    call option4
goto L6
L5: call default_action
L6: ...
case i:
  1:    option1()
  2, 7:  option2()
  3, 4, 5: option3()
  10:   option4()
otherwise: default_action()
IMPLEMENTATION OF SWITCH STATEMENTS: JUMP TABLE

Assume $i$ is stored in register $R1$.

case $i$:
    1:    option1()
    2, 7: option2()
    3, 4, 5: option3()
    10:   option4()
otherwise: default_action()

T: &L1
   &L2
   &L3
   &L3
   &L3
   &L3
   &L5
   &L2
   &L5
   &L5
   &L4

L1: call option1
goto L7
L2: call option2
goto L7
L3: call option3
goto L7
L4: call option4
goto L7
L5: call default_action
goto L7
L6: if R1 < 1 goto L5
    if R1 > 10 goto L5
    R1 := R1 - 1
    R2 := T[R1]
goto *R2
L7: ...
IMPLEMENTATION OF SWITCH STATEMENTS

Jump table:
+ Fast: one table lookup to find the right branch
- Potentially large table: one entry per possible value

Hash table:
+ Fast: one hash table access to find the right branch
- More complicated
  - Elements in a range need to be stored individually; again, possibly a large table

Linear search:
- Potentially slow
+ No storage overhead

Binary search:
- Fast, but slower than table lookup
+ No storage overhead

No single implementation is best in all circumstances. Compilers often use different strategies based on the specific code.
Jump table:

- Fast: one table lookup to find the right branch
- Potentially large table: one entry per possible value
## IMPLEMENTATION OF SWITCH STATEMENTS

<table>
<thead>
<tr>
<th>Jump table:</th>
<th>Hash table:</th>
</tr>
</thead>
<tbody>
<tr>
<td>+ Fast: one table lookup to find the right branch</td>
<td>+ Fast: one hash table access to find the right branch</td>
</tr>
<tr>
<td>– Potentially large table: one entry per possible value</td>
<td>– More complicated</td>
</tr>
<tr>
<td></td>
<td>– Elements in a range need to be stored individually; again, possibly a large table</td>
</tr>
</tbody>
</table>

No single implementation is best in all circumstances. Compilers often use different strategies based on the specific code.
IMPLEMENTATION OF SWITCH STATEMENTS

Jump table:
- Fast: one table lookup to find the right branch
- Potentially large table: one entry per possible value

Hash table:
- Fast: one hash table access to find the right branch
- More complicated
- Elements in a range need to be stored individually; again, possibly a large table

Linear search:
- Potentially slow
- No storage overhead
### IMPLEMENTATION OF SWITCH STATEMENTS

#### Jump table:
- **Fast**: one table lookup to find the right branch
- **Potentially large table**: one entry per possible value

#### Hash table:
- **Fast**: one hash table access to find the right branch
- **More complicated**
- **Elements in a range need to be stored individually; again, possibly a large table**

#### Linear search:
- **Potentially slow**
- **No storage overhead**

#### Binary search:
- **Fast, but slower than table lookup**
- **No storage overhead**
IMPLEMENTATION OF SWITCH STATEMENTS

Jump table:
+ Fast: one table lookup to find the right branch
- Potentially large table: one entry per possible value

Hash table:
+ Fast: one hash table access to find the right branch
- More complicated
- Elements in a range need to be stored individually; again, possibly a large table

Linear search:
- Potentially slow
+ No storage overhead

Binary search:
± Fast, but slower than table lookup
+ No storage overhead

No single implementation is best in all circumstances. Compilers often use different strategies based on the specific code.
Enumeration-controlled loops:

• Example: for-loop
• One iteration per element in finite set
• The number of iterations is known in advance.

Logically controlled loops:

• Example: while-loop
• Executed until a Boolean condition changes
• The number of iterations is not known in advance.
Enumeration-controlled loops:

- Example: for-loop
- One iteration per element in finite set
- The number of iterations is known in advance.

Logically controlled loops:

- Example: while-loop
- Executed until a Boolean condition changes
- The number of iterations is not known in advance.

Some languages do not have loop constructs (Scheme, Haskell, ...). They use tail recursion instead.
Pre-loop test:

```java
while (cond) {
    ...
}
```
LOGICALLY CONTROLLED LOOPS

Pre-loop test:

```java
while (cond) {
  ...
}
```

Post-loop test:

```java
do {
  ...
} while (cond);
```
LOGICALLY CONTROLLED LOOPS

Pre-loop test:
while (cond) {
  ...
}

Post-loop test:
do {
  ...
} while (cond);

Mid-loop test or “one-and-a-half loop”:
loop {
  ...
  if (cond1) exit;
  ...
  if (cond2) exit;
  ...
}
TRADE-OFFS IN ITERATION CONSTRUCTS (1)

Logically controlled loops:

- Flexible
- Expensive

The for-loop in C/C++ is merely syntactic sugar for the init-test-step idiom in implementing enumeration using logically controlled loops!

while (cond) {
    statements;
}

L1: R1 := evaluate cond
if not R1 goto L2
    statements
    goto L1
L2: ...

for (init; cond; step) {
    statements;
}

init
L1: R1 := evaluate cond
    statements
    step
    goto L1
L2: ...
Logically controlled loops:

+ Flexible

```
while (cond) {
    statements;
}
```
Logically controlled loops:

+ Flexible

- Expensive

```c
while (cond) {
    statements;
}
```

```c
L1: R1 := evaluate cond
    if not R1 goto L2
    statements
    goto L1
L2: ...
```
**Logically controlled loops:**

+ Flexible
- Expensive

The for-loop in C/C++ is merely syntactic sugar for the init-test-step idiom in implementing enumeration using logically controlled loops!

```plaintext
while (cond) {
  statements;
}

L1: R1 := evaluate cond
    if not R1 goto L2
    statements
    goto L1
L2: ...
```

```plaintext
for (init; cond; step) {
  statements;
}
```
Logically controlled loops:

+ Flexible
- Expensive

The for-loop in C/C++ is merely syntactic sugar for the init-test-step idiom in implementing enumeration using logically controlled loops!

```c
while (cond) {
    statements;
}

L1: R1 := evaluate cond
    if not R1 goto L2
    statements
    goto L1
L2: ...
```

```c
for (init; cond; step) {
    statements;
}

init
L1: R1 := evaluate cond
    if not R1 goto L2
    statements
    step
    goto L1
L2: ...
```
Potentially much more efficient:

FOR i = start TO end BY step DO
    statements
END

If modifying the loop variable inside the loop is allowed:

R1 := start
R2 := end
R3 := step
L1: if R1 > R2 goto L2
    statements
    R1 = R1 + R3
    goto L1
L2: ...

If modifying the loop variable inside the loop is not allowed:

R1 := floor((end - start) / step) + 1
L1: if not R1 goto L2
    statements
    decrement R1
    goto L1
L2: ...
Potentially much more efficient:

```
FOR i = start TO end BY step DO
    statements
END
```

If modifying the loop variable inside the loop is allowed:

```
R1 := start
R2 := end
R3 := step
L1: if R1 > R2 goto L2
    statements
    R1 = R1 + R3
    goto L1
L2: ...
```
Potentially much more efficient:

\[
\text{FOR } i = \text{start TO end BY step DO}
\]
\[
\text{statements}
\]
\[
\text{END}
\]

If modifying the loop variable inside the loop is allowed:

\[
R1 := \text{start}
\]
\[
R2 := \text{end}
\]
\[
R3 := \text{step}
\]
\[
L1: \text{if } R1 > R2 \text{ goto } L2
\]
\[
\text{statements}
\]
\[
R1 = R1 + R3
\]
\[
\text{goto } L1
\]
\[
L2: \ldots
\]

If modifying the loop variable inside the loop is not allowed:

\[
R1 := \text{floor}((\text{end - start}) / \text{step}) + 1
\]
\[
L1: \text{if not } R1 \text{ goto } L2
\]
\[
\text{statements}
\]
\[
\text{decrement } R1
\]
\[
\text{goto } L1
\]
\[
L2: \ldots
\]
“Break” statement (“last” in Perl):
Exit the nearest enclosing for-, do-, while- or switch-statement.

“Continue” statement (“next” in Perl):
Skip the rest of the current iteration.
“Break” statement (“last” in Perl):
Exit the nearest enclosing for-, do-, while- or switch-statement.

“Continue” statement (“next” in Perl):
Skip the rest of the current iteration.

Both statements may be followed by a label that specifies

- An enclosing loop (continue) or
- Any enclosing statement (break).
“Break” statement (“last“ in Perl):
Exit the nearest enclosing for-, do-, while- or switch-statement.

“Continue” statement (“next” in Perl):
Skip the rest of the current iteration.

Both statements may be followed by a label that specifies
• An enclosing loop (continue) or
• Any enclosing statement (break).

A loop may have a finally part, which is always executed no matter whether the iteration executes normally or is terminated using a continue or break statement.
Often, for-loops are used to iterate over sequences of elements (stored in a data structure, generated by a procedure, ...).
Often, for-loops are used to iterate over sequences of elements (stored in a data structure, generated by a procedure, ...).

Iterators/generators provide a clean idiom for iterating over a sequence without a need to know how the sequence is generated.
Often, for-loops are used to iterate over sequences of elements (stored in a data structure, generated by a procedure, …).

**Iterators/generators** provide a clean idiom for iterating over a sequence without a need to know how the sequence is generated.

**Generators in Python:**

```python
def lexy(length):
    yield ''
    if length > 0:
        for ch in ['a', 'b', 'c', 'd']:
            for w in lexy(length - 1):
                yield ch + w

for w in lexy(3):
    print(w)
```
C++ and Java provide iterator classes that can be used to enumerate the elements of a collection (or programmatically generate a sequence of elements to be traversed).
C++ and Java provide iterator classes that can be used to enumerate the elements of a collection (or programmatically generate a sequence of elements to be traversed).

C++:

```cpp
for (cont::iterator i = cont.begin(); i != cont.end(); ++i) {
    // Use i
}
```
C++ and Java provide iterator classes that can be used to enumerate the elements of a collection (or programmatically generate a sequence of elements to be traversed).

**C++:**

```cpp
for (cont::iterator i = cont.begin(); i != cont.end(); ++i) {
    // Use i
}
```

**Java 1.4** is similar in its use of the `Enumeration` interface:

```java
Enumeration e = cont.elements();
while (e.hasMoreElements()) {
    MyObj o = (MyObj) e.nextElement();
    // Use o
}
```
Many modern languages provide convenient syntax for iterating over sequences generated using iterators. Behind the scenes, this is translated into code that explicitly uses iterator objects.
Many modern languages provide convenient syntax for iterating over sequences generated using iterators. Behind the scenes, this is translated into code that explicitly uses iterator objects.

**Modern Java (post Java 5):**

```java
for (MyObj obj : cont) {
    // Use obj
}
```
Many modern languages provide convenient syntax for iterating over sequences generated using iterators. Behind the scenes, this is translated into code that explicitly uses iterator objects.

**Modern Java (post Java 5):**

```java
for (MyObj obj : cont) {
    // Use obj
}
```

**Modern C++ (post C++11):**

```cpp
for (auto &obj : cont) {
    // Use obj
}
```
ITERATION WITHOUT ITERATORS

In languages without iterators/generators (e.g., C), we can simulate iterators using function calls:

```c
for (it = begin(coll); it != end(coll); it = next(it)) {
    /* Do something with *it */
}
```
Functions being first-class objects allows passing a function to be applied to every element to an “iterator” that traverses the collection.

Haskell:

doubles = map (* 2) [1 ..]
pairs = zip [1 ..] doubles
doubles2 = filter even [1 ..]
Every iterative procedure can be turned into a recursive one:

```java
while (condition) { S1; S2; ... }
```

becomes

```java
procedure P() {
    if (condition) {
        S1; S2; ...; P();
    }
}
```
Every iterative procedure can be turned into a recursive one:

```java
while (condition) { S1; S2; ... }
```

becomes

```java
procedure P() {
    if (condition) {
        S1; S2; ...; P();
    }
}
```

The converse is not true (e.g., Quicksort, Merge Sort, fast matrix multiplication, ...)

Every iterative procedure can be turned into a recursive one:

```java
while (condition) { S1; S2; ... }
```

becomes

```java
procedure P() {
    if (condition) {
        S1; S2; ...; P();
    }
}
```

The converse is not true (e.g., Quicksort, Merge Sort, fast matrix multiplication, ...)

The type of recursive procedure above can be translated back into a loop by the compiler (tail recursion).
Applicative-and-normal-order evaluation

Applicative-order evaluation
Arguments are evaluated before a subroutine call

Normal-order evaluation
• Arguments are passed to the subroutine unevaluated.
• The subroutine evaluates them as needed.
• Useful for infinite or lazy data structures that are computed as needed.
• Example: macros in C/C++

Normal-order evaluation is fine in functional languages but problematic if there are side effects. Why?

Normal-order evaluation is potentially inefficient. Why?

How can we avoid this?
Applicative-order evaluation
Arguments are evaluated before a subroutine call

Normal-order evaluation
• Arguments are passed to the subroutine unevaluated.
• The subroutine evaluates them as needed.
**Applicative-and Normal-Order Evaluation**

**Applicative-order evaluation**
- Arguments are evaluated before a subroutine call

**Normal-order evaluation**
- Arguments are passed to the subroutine unevaluated.
- The subroutine evaluates them as needed.

- Default in most programming languages

Normal-order evaluation is fine in functional languages but problematic if there are side effects. Why?

Normal-order evaluation is potentially inefficient. Why?

How can we avoid this?
Applicative-order evaluation
Arguments are evaluated before a subroutine call

Normal-order evaluation
- Arguments are passed to the subroutine unevaluated.
- The subroutine evaluates them as needed.

- Default in most programming languages
- Useful for infinite or lazy data structures that are computed as needed.
- Example: macros in C/C++
Applicative-order evaluation
Arguments are evaluated before a subroutine call

Normal-order evaluation
- Arguments are passed to the subroutine unevaluated.
- The subroutine evaluates them as needed.

Normal-order evaluation is fine in functional languages but problematic if there are side effects. Why?

- Default in most programming languages
- Useful for infinite or lazy data structures that are computed as needed.
- Example: macros in C/C++
Applicative-order evaluation
Arguments are evaluated before a subroutine call

Normal-order evaluation
- Arguments are passed to the subroutine unevaluated.
- The subroutine evaluates them as needed.

Normal-order evaluation is fine in functional languages but problematic if there are side effects. Why?
Normal-order evaluation is potentially inefficient. Why?
Applicative-order evaluation
Arguments are evaluated before a subroutine call

Normal-order evaluation
- Arguments are passed to the subroutine unevaluated.
- The subroutine evaluates them as needed.

Normal-order evaluation is fine in functional languages but problematic if there are side effects. Why?

Normal-order evaluation is potentially inefficient. Why? How can we avoid this?
Lazy evaluation

- Evaluate expressions when their values are needed.
- Cache results to avoid recomputation.
Lazy evaluation

- Evaluate expressions when their values are needed.
- Cache results to avoid recomputation.

Haskell:

```haskell
naturals :: [Int]
naturals = next 1
  where next i = i : rest
  where rest = next (i+1)

take 10 naturals -- [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```
Lazy evaluation

- Evaluate expressions when their values are needed.
- Cache results to avoid recomputation.

Haskell:

```haskell
naturals :: [Int]
naturals = next 1
    where next i = i : rest
        where rest = next (i+1)

take 10 naturals -- [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

Strict evaluation may be more efficient. Haskell provides means for us to force the strict evaluation of arguments (bang patterns).
By default, Scheme uses strict applicative-order evaluation.

This code runs forever:

```
(define naturals
  (letrec ((next (lambda (n)
                    (cons n (next (+ n 1)))))))
  (next 1)))
```
A lazy version of the same code:

(define naturals
  (letrec ((next (lambda (n)
      (cons n (delay (next (+ n 1)))))))
    (next 1)))

(define head car)

(define (tail stream) (force (cdr stream)))

(head naturals)  ; 1
(head (tail naturals))  ; 2
(head (tail (tail naturals)))  ; 3
delay is a special form or macro that wraps the expression in a function:

```
(define-syntax delay
  (syntax-rules ()
    ((delay exp) (lambda () exp))))
```

What's the problem with this implementation of delay?

It evaluates exp every time. This is inefficient (essentially normal-order evaluation).
delay is a **special form** or **macro** that wraps the expression in a function:

```
(define-syntax delay
  (syntax-rules ()
    ((delay exp) (lambda () exp))))
```

**force** simply evaluates the given function:

```
(define (force delayed-exp)
  (delayed-exp))
```
delay is a **special form** or **macro** that wraps the expression in a function:

```
(define-syntax delay
  (syntax-rules ()
    ((delay exp) (lambda () exp))))
```

**force** simply evaluates the given function:

```
(define (force delayed-exp)
  (delayed-exp))
```

What’s the problem with this implementation of **delay**?
delay is a **special form** or **macro** that wraps the expression in a function:

```
(define-syntax delay
  (syntax-rules ()
    ((delay exp) (lambda () exp))))
```

force simply evaluates the given function:

```
(define (force delayed-exp)
  (delayed-exp))
```

What’s the problem with this implementation of delay?

It evaluates exp every time. This is inefficient (essentially normal-order evaluation).
A better implementation of delay:

```
(define-syntax delay
  (syntax-rules ()
    ((delay exp) (memoize (lambda () exp)))))

(define (memoize f)
  (let ((first? #t)
         (val #f))
    (lambda ()
      (if first?
          (begin (set! first? #f)
                  (set! val (f)))
          val)))
```

This is pretty much what Haskell does.
A better implementation of delay:

```
(define-syntax delay
  (syntax-rules ()
    ((delay exp) (memoize (lambda () exp))))
)

(define (memoize f)
  (let ((first? #t)
         (val #f))
    (lambda ()
      (if first?
        (begin (set! first? #f)
               (set! val (f)))
               val)))
)
```

This is pretty much what Haskell does.
Example:

```c
#define DIVIDES(n, a) (!(n) % a))
```
Example:

```
#define DIVIDES(n, a) (!(n % a))
```

Problems:
Example:

```
#define DIVIDES(n, a) (!(n % a))
```

Problems:

- Cannot be used recursively.
Example:

```c
#define DIVIDES(n, a) (!(n % a))
```

Problems:

- Cannot be used recursively.
- Textual expansion may not mean what’s intended: Evaluate `DIVIDES(x, y+2)` using the above definition and using

```c
#define DIVIDES(n, a) (!(n % a))
```
• **Side effects:** Evaluate \( \text{MAX}(x++ , y++) \) using

```c
#define MAX(a, b) ((a) > (b) ? (a) : (b))
```
• Side effects: Evaluate $\text{MAX}(x++, y++)$ using
  
  ```c
#define MAX(a, b) ((a) > (b) ? (a) : (b))
  ```

• Name clashes with variables: Evaluate $\text{SWAP}(x, t)$ using

  ```c
#define SWAP(a, b) { int t = a; a = b; b = t; }
  ```
• Side effects: Evaluate \texttt{MAX(x++, y++)} using
\begin{verbatim}
#define MAX(a, b) ((a) > (b) ? (a) : (b))
\end{verbatim}

• Name clashes with variables: Evaluate \texttt{SWAP(x, t)} using
\begin{verbatim}
#define SWAP(a, b) { int t = a; a = b; b = t; }
\end{verbatim}

In C++, inline functions are usually a better alternative.
SUMMARY

- Think in terms of control abstractions rather than syntax!
- Expression evaluation order is left to the compiler; avoid side effects.
- Understand what a variable use means (l-value/r-value; value/reference).
- Short-circuiting helps efficiency and allows some elegant idioms.
- Avoid goto.
- switch is often more efficient than multi-way if.
- for-loops can be more efficient than while-loops (not in C, Java, Python, …).
- Iterators/generators provide an abstraction for enumerating the elements of a sequence useful for iteration constructs.
- Recursion is more general/powerful than iteration.
- Applicative-order evaluation is fast, normal-order evaluation is flexible, lazy evaluation offers a trade-off.