CONTROL FLOW

PRINCIPLES OF PROGRAMMING LANGUAGES

Norbert Zeh

Winter 2018

Dalhousie University

LANGUAGE MECHANISMS FOR CONTROL FLOW

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*

EXPRESSION EVALUATION

Order of evaluation may influence result of computation.

EXPRESSION EVALUATION

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

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

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

REFERENCES AND VALUES

Expressions that denote values are referred to as r-values.

Expressions that denote memory locations are referred to as 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:

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

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.

a's value

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

REFERENCES AND 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:

- 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.

a’s value

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

a’s memory location

EXPLICIT DEREFERENCING

Some languages explicitly distinguish between l-values and r-values:

- BLISS: `X := .X + 1`
- ML: `X := !X + 1`

EXPLICIT DEREFERENCING

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++):

```
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.

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.

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

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.

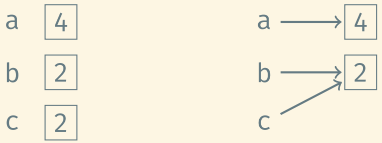
An example: Java

- Value model for built-in types
- Reference model for classes

VALUE MODEL VS REFERENCE MODEL

```
b = 2; c = b; a = b + c;
```

Value model | Reference model



JAVA EXAMPLES

```
int a = 5;
int b = a;
b += 10;
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
```

```
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:

```
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 EXAMPLES

```
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:

JAVA EXAMPLES

```
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
```

JAVA EXAMPLES

```
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 EXAMPLES

```
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
```

EVALUATION ORDER WITHIN EXPRESSIONS

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.

EVALUATION ORDER WITHIN EXPRESSIONS

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 WITHIN EXPRESSIONS

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 WITHIN EXPRESSIONS

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.

AN EXAMPLE WITH SIDE EFFECTS IN C

```
for(i = m = M = 1; N - ++i; M = m + (m = M));
```

What does this code compute?

AN EXAMPLE WITH SIDE EFFECTS IN C

```
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)`.

AN EXAMPLE WITH SIDE EFFECTS IN C

```
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 intented

N	m	M
2	1	1
3	1	2
4	2	3
5	3	5
6	5	8

$\mathtt{M} = F_{\mathtt{N}}$ (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)`.

Probably intented

N	m	M
2	1	1
3	1	2
4	2	3
5	3	5
6	5	8

$M = F_N$ (Nth Fibonacci number)

Actual

N	m	M
2	1	1
3	1	2
4	2	4
5	4	8
6	8	16

$M = 2^{N-2}$

SHORT-CIRCUIT EVALUATION OF BOOLEAN EXPRESSIONS

`(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 OF BOOLEAN EXPRESSIONS

`(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.

SHORT-CIRCUIT EVALUATION OF BOOLEAN EXPRESSIONS

`(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:

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

COMMON IDIOMS ENABLED BY SHORT-CIRCUIT EVALUATION

Checking for NULL pointers in C:

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

Exit on failure in Perl:

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

COMMON IDIOMS ENABLED BY SHORT-CIRCUIT EVALUATION

Checking for NULL pointers in C:

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

Exit on failure in Perl:

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

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

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

becomes

```
(x > max) && max = x;
```

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.

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?

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?

- The value of the last subexpression (most common)

  C: `a = 4, b = 5;` $\Longrightarrow 5$

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;` $\Longrightarrow 5$

  LISP: `(progn (setq a 4) (setq b 5))` $\Longrightarrow 5$

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?

- The value of the last subexpression (most common)

  C: `a = 4, b = 5;` $\Longrightarrow 5$

  LISP: `(progn (setq a 4) (setq b 5))` $\Longrightarrow 5$

- The value of the first subexpression

  LISP: `(prog1 (setq a 4) (setq b 5))` $\Longrightarrow 4$

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;` $\Longrightarrow 5$

  LISP: `(progn (setq a 4) (setq b 5))` $\Longrightarrow 5$

- The value of the first subexpression

  LISP: `(prog1 (setq a 4) (setq b 5))` $\Longrightarrow 4$

- The value of the second subexpression

  LISP: `(prog2 (setq a 4) (setq b 5) (setq c 6)` $\Longrightarrow 5$

GOTO AND ALTERNATIVES

Use of `goto` is bad programming practice if the same effect can be achieved using different constructs.

GOTO AND ALTERNATIVES

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

GOTO AND ALTERNATIVES

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

SELECTION (ALTERNATION)

Standard if-then-else statement:

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

SELECTION (ALTERNATION)

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
```

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

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

SWITCH 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

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

IMPLEMENTATION OF IF STATEMENTS

```
if i == 1:
  option1()
elsif i in [2, 7]:
  option2()
elsif i in [3, 4, 5]:
  option3()
elsif i == 10:
  option4()
else:
  default_action()
```

IMPLEMENTATION OF IF STATEMENTS

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

```
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 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: ...
```

IMPLEMENTATION OF SWITCH STATEMENTS: JUMP TABLE

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

IMPLEMENTATION OF SWITCH STATEMENTS: JUMP TABLE

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

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

```
T: &L1
   &L2
   &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

IMPLEMENTATION OF SWITCH STATEMENTS

Jump table:

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

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

IMPLEMENTATION OF SWITCH STATEMENTS

Jump table:

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

Linear search:

– Potentially slow
+ No storage overhead

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

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:

- $\pm$ 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:

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

LOGICALLY CONTROLLED LOOPS

Pre-loop test:

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

Post-loop test:

```
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

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

TRADE-OFFS IN ITERATION CONSTRUCTS (1)

Logically controlled loops:

+ Flexible
– Expensive

```
while (cond) {
  statements;
}

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

TRADE-OFFS IN ITERATION CONSTRUCTS (1)

Logically controlled loops:

+ Flexible
– Expensive

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

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

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

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

TRADE-OFFS IN ITERATION CONSTRUCTS (1)

Logically controlled loops:

+ Flexible
– Expensive

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

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

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

```
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
```

TRADE-OFFS IN ITERATION CONSTRUCTS (2)

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: ...
```

TRADE-OFFS IN ITERATION CONSTRUCTS (2)

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: ...
```

“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).

LABELLED BREAK AND CONTINUE

“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:

```
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)
```

ITERATOR OBJECTS

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).

ITERATOR OBJECTS

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++:

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

ITERATOR OBJECTS

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++:

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

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

```
Enumeration e = cont.elements();
while (e.hasMoreElements()) {
  MyObj o = (MyObj) e.nextElement();
  // Use o
}
```

TYING ITERATOR OBJECTS TO FOR-LOOPS

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.

TYING ITERATOR OBJECTS TO FOR-LOOPS

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):

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

TYING ITERATOR OBJECTS TO FOR-LOOPS

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):

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

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

```
for (auto &obj : cont) {
  // Use obj
}
```

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

```
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 ..]
```

RECURSION

Every iterative procedure can be turned into a recursive one:

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

becomes

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

Every iterative procedure can be turned into a recursive one:

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

becomes

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

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

RECURSION

Every iterative procedure can be turned into a recursive one:

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

becomes

```
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-order evaluation

Arguments are evaluated before a subroutine call

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

APPLICATIVE AND NORMAL-ORDER EVALUATION

Applicative-order evaluation

Arguments are evaluated before a subroutine call

- Default in most programming languages

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++

APPLICATIVE AND NORMAL-ORDER EVALUATION

Applicative-order evaluation

Arguments are evaluated before a subroutine call

- Default in most programming languages

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?

APPLICATIVE AND NORMAL-ORDER EVALUATION

Applicative-order evaluation

Arguments are evaluated before a subroutine call

- Default in most programming languages

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 evalutaion is potentially inefficient. Why?

APPLICATIVE AND NORMAL-ORDER EVALUATION

Applicative-order evaluation

Arguments are evaluated before a subroutine call

- Default in most programming languages

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 evalutaion 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:

```
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
```

IMPLEMENTATION OF DELAY AND FORCE (1)

`delay` is a special form or macro that wraps the expression in a function:

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

IMPLEMENTATION OF DELAY AND FORCE (1)

`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`?

IMPLEMENTATION OF DELAY AND FORCE (1)

`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)))
```

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.

NORMAL ORDER EVALUATION: C/C++ MACROS (1)

Example:

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

NORMAL ORDER EVALUATION: C/C++ MACROS (1)

Example:

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

Problems:

Example:

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

Problems:

- Cannot be used recursively.

NORMAL ORDER EVALUATION: C/C++ MACROS (1)

Example:

```
#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

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

NORMAL ORDER EVALUATION: C/C++ MACROS (2)

- Side effects: Evaluate `MAX(x++, y++)` using

```
#define MAX(a, b) ((a) > (b) ? (a) : (b))
```

NORMAL ORDER EVALUATION: C/C++ MACROS (2)

- Side effects: Evaluate `MAX(x++, y++)` using

```
#define MAX(a, b) ((a) > (b) ? (a) : (b))
```

- Name clashes with variables: Evaluate `SWAP(x, t)` using

```
#define SWAP(a, b) { int t = a; a = b; b = t; }
```

NORMAL ORDER EVALUATION: C/C++ MACROS (2)

- Side effects: Evaluate `MAX(x++, y++)` using

  ```
  #define MAX(a, b) ((a) > (b) ? (a) : (b))
  ```

- Name clashes with variables: Evaluate `SWAP(x, t)` using

  ```
  #define SWAP(a, b) { int t = a; a = b; b = t; }
  ```

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

- 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.