Control Flow

CSCI 3136

Principles of Programming Languages

Faculty of Computer Science

Dalhousie University

Winter 2012

Reading: Chapters 6

Language Mechanisms for Flow Control

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.

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 (functional or logic) programming languages

`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

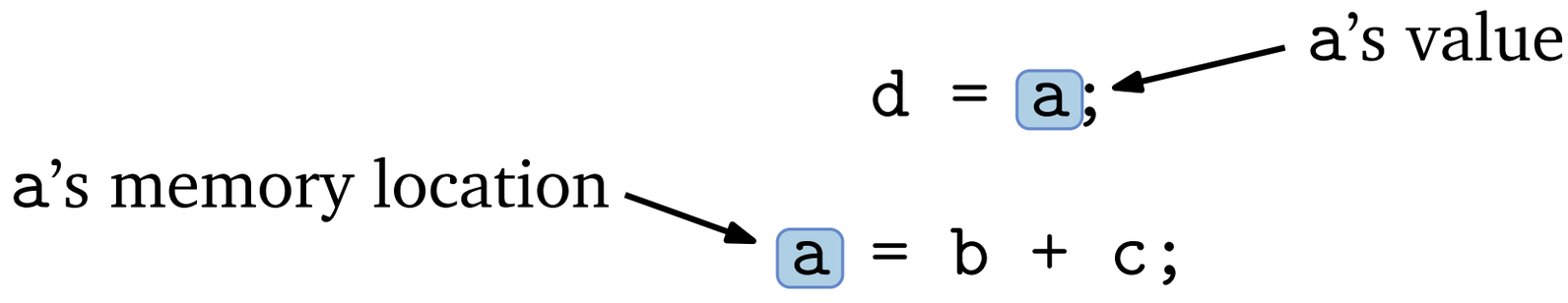
References and Values

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

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

The meaning of a variable name normally 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.



Explicit Deferencing

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

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

Explicit Deferencing

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

int a[ 10 ];
for( int i = 0; i < 100; i++ )
  f( i ) = i;
```

Models of Variables

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.

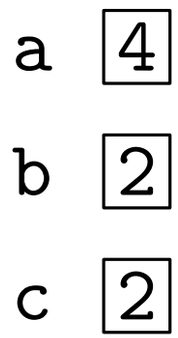
An example: Java

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

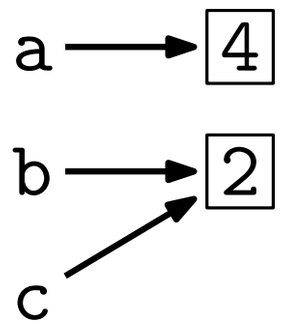
Value Model vs Reference Model

```
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 Examples

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

Output

```
a = 5
b = 15
```

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

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
```

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
```

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

Output

```
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
```

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

Output

```
a = hi
b = hi world
```

```
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
```

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

Output

```
a = hi
b = hi world
```

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

Output

```
true
```

```
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
```

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

Output

```
a = hi
b = hi world
```

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

Output

```
true
```

```
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 Ordering Within Expressions

It is usually unwise to write expressions where a side effect of evaluating one 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, ... 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)`.

Probably intended

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

Actual

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

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 intended

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)

Actual

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

$\mathtt{M} = 2^{\mathtt{N}-2}$

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 intended

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)

Actual

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

$\mathtt{M} = 2^{\mathtt{N}-2}$

Other problems?

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 intended

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)

Actual

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

$\mathtt{M} = 2^{\mathtt{N}-2}$

Other problems?

Try to run this with `N = 1`.

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.

This is a useful optimization. If the evaluation of `b` has side-effects, however, the meaning of the code may be changed.

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 a useful optimization. If the evaluation of `b` has side-effects, however, the meaning of the code may be changed.

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

Ada

- `and` vs `and then`
- `or` vs `or else`

Short-Circuit Evaluation: Examples

C:

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

Perl:

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

Replacing "if" in Perl or shell scripts:

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

becomes

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

Sequencing

In imperative programming languages, sequencing comes naturally, without a need for special syntax to support it.

Mixed imperative/functional 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 (supported in LISP)

  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. Sometimes, however, 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 ... then ...
       else ...
```

Multi-way if-then-else statement

```
if    ... then ...
elsif ... then ...
...
else ...
```

Switch statement

```
switch ... of
  case ...: ...
  case ...: ...
  ...
```

Selection (Alternation)

Standard if-then-else statement

```
if ... then ...
       else ...
```

Multi-way if-then-else statement

```
if    ... then ...
elsif ... then ...
elsif ... then ...
...
else ...
```

Some languages write `elseif`, `elif` or `else if` instead of `elsif`. The latter is problematic. Why?

Switch statement

```
switch ... of
  case ...: ...
  case ...: ...
  ...
```

Switch Statements

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

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 Switch Statements (1)

```
IF i = 1 THEN
   clause_A
ELSIF i IN 2, 7 THEN
   clause_B
ELSIF i IN 3..5 THEN
   clause_C
ELSIF i = 10 THEN
   clause_D
ELSE
   clause_E
END
```

Assume `i` is stored in register r_1

```
    if r1 ≠ 1 goto L1
    clause_A
    goto L6
L1: if r1 = 2 goto L2
    if r1 ≠ 7 goto L3
L2: clause_B
    goto L6
L3: if r1 < 3 goto L4
    if r1 > 5 goto L4
    clause_C
    goto L6
L4: if r1 ≠ 10 goto L5
    clause_D
    goto L6
L5: clause_E
L6: ...
```

Implementation of Switch Statements (2)

```
CASE i OF
   1:     clause_A
| 2, 7:  clause_B
| 3..5:  clause_C
| 10:    clause_D
   ELSE   clause_E
END
```

Implementation of Switch Statements (2)

```
CASE i OF
   1:      clause_A
| 2, 7:    clause_B
| 3..5:    clause_C
| 10:      clause_D
   ELSE    clause_E
END
```

Jump table

Assume `i` is stored in register r_1

T: $\&L_1$
$\&L_2$
$\&L_3$
$\&L_3$
$\&L_3$
$\&L_5$
$\&L_2$
$\&L_5$
$\&L_5$
$\&L_4$

L_1: *clause_A*
goto L_7
L_2: *clause_B*
goto L_7
L_3: *clause_C*
goto L_7
L_4: *clause_D*
goto L_7
L_5: *clause_E*
goto L_7
L_6: if $r_1 < 1$ goto L_5
if $r_1 > 10$ goto L_5
$r_1 := r_1 - 1$
$r_2 := T[r_1]$
goto $*r_2$
L_7: ...

Implementation of Switch Statements (3)

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 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.

Iteration

Enumeration-controlled loops

- Example: `for`-loop
- One iteration per element in a 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.

Iteration

Enumeration-controlled loops

- Example: `for`-loop
- One iteration per element in a 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 (e.g., Scheme). They use tail recursion instead.

Logically Controlled Loops

Pre-loop test

```
while ... do ...
```

Post-loop test

```
repeat ... until ...
do ... while ...
```

Loop is executed at least once.

Mid-loop test or “one-and-a-half loop”

```
loop
   ...
   when ... exit
   ...
   when ... exit
   ...
end
```

Trade-Offs in Iteration Constructs (1)

Logically controlled loops are very
flexible but expensive.

```
WHILE cond do
   statements
END
```

L_1: r_1 := evaluate *cond*
    `if not` r_1 `goto` L_2
    statements
    `goto` L_1
L_2: ...

Trade-Offs in Iteration Constructs (1)

Logically controlled loops are very flexible but expensive.

```
WHILE cond do
   statements
END
```

```
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 common init-step-test idiom in implementing enumeration using logically controlled loops.

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

Trade-Offs in Iteration Constructs (1)

Logically controlled loops are very flexible but expensive.

```
WHILE cond do
   statements
END
```

```
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 common init-step-test 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: ...
```

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:

$r_1 := start$
$r_2 := end$
$r_3 := step$
L_1: `if` $r_1 > r_2$ `goto` L_2
statements
$r_1 := r_1 + r_3$
`goto` L_1
L_2: ...

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

$r_1 := \lfloor (end - start)/step \rfloor + 1$
L_1: `if not` r_1 `goto` L_2
statements
`decrement` r_1
`goto` L_1
L_2: ...

Labelled Break and Continue

"Break" statement ("last" in Perl)

- Exits the nearest enclosing `for`, `do`, `while` or `switch` statement.

"Continue" statement ("next" in Perl)

- Skips 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`).

The loop may have a `finally` part, which is always executed no matter whether the iteration is executed normally or terminated using a `continue` or `break` statement.

Iterators and Generators

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

```
from __future__ import generators

def lexy( len ):
  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
```

Generators in Clu

```
from_to_by = iter( from, to, by : int ) yields ( int )
  i : int := from
  if by > 0 then
    while i <= to do
      yield i
      i +:= by
    end
  else
    while i >= to do
      yield i
      i +:= by
    end
  end
end from_to_by

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

Generators in Icon

Generators in Icon are similar to generators in Clu but are more embedded in the language semantics.

If embedded in an expression, they iterate until the expression succeeds.

Example

```
if ( i := find( "x", s ) ) = find( "x", t ) then {
   ...
}
```

Iterator Objects

C++ and Java provide iterator classes that can be used to enumerate the elements of a collection (or programatically 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 programatically 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 programatically 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

Euclid provides a special semantics of `for` statements, which relies on the use of a generator module that defines a procedure `Next` and two variables `value` and `stop`.

Tying Iterator Objects to for-Loops

Euclid provides a special semantics of `for` statements, which relies on the use of a generator module that defines a procedure `Next` and two variables `value` and `stop`.

Java 5 is similar; its for syntax relies on the "container" class implementing an Iterator interface:

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

Iteration without Iterators

In **C**, we can simulate iterators using function calls:

```
for( it = begin( coll ); it != end( coll ); it = next( it ) ) {
  /* Do something with *it */
}
```

Iteration in Functional Languages

Functions as first-class objects allow passing a function to be applied to every element to an “iterator” that traverses the collection.

Example (Haskell)

```
-- Print the first ten Fibonacci numbers,
-- each multiplied by 2
main = do
  mapM_ (\x -> putStr $ show x ++ " ") fib2
  putStrLn ""
  where
    fib2 = map (* 2)
         $ take 10 fibonacci

-- The infinite sequence of Fibonacci numbers
fibonacci = 1:1:(zipWith (+) fibonacci (tail fibonacci))
```

Recursion

Every iterative procedure can be turned into a recursive one:

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

becomes

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

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, ...)

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, ...)

Why don't functional languages support iteration?

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, ...)

Why don't functional languages support iteration?

Iteration is a strictly imperative feature: it relies on updating the iterator variable.

Implementation of Recursion

A naïve implementation of recursion is often less efficient than a naïve implementation of iteration.

Implementation of Recursion

A naïve implementation of recursion is often less efficient than a naïve implementation of iteration.

An optimizing compiler often converts recursion into iteration when possible:

- **Tail recursion:** There is no work to be done after the recursive call. (Standard method for implementing iteration in functional languages)

```
(define (sum xs)
  (letrec ((sum' (lambda (s xs)
                   (if (null? xs)
                     s
                     (sum' (+ s (car xs)) (cdr xs))))))
    (sum' 0 xs)))
```

- Work to be done after the recursive call may be passed to the recursive call as a continuation.

Recursion: Example (1)

$$\text{Compute} \sum_{1 \le i \le 10} f(i)$$

```
(define (sum f low high)
  (if (= low high)
      (f low)
      (+ (f low) (sum f (+ low 1) high))))
```

Recursion: Example (1)

Compute $\sum_{1 \leq i \leq 10} f(i)$

```
(define (sum f low high)
  (if (= low high)
      (f low)
      (+ (f low) (sum f (+ low 1) high))))
```

Is this implementation tail-recursive?

Recursion: Example (1)

$$\text{Compute} \sum_{1 \le i \le 10} f(i)$$

```
(define (sum f low high)
  (if (= low high)
      (f low)
      (+ (f low) (sum f (+ low 1) high))))
```

Is this implementation tail-recursive?

```
(define (sum f low high)
  (letrec ((sum' (lambda (i subtotal)
                   (if (> i high)
                     subtotal
                     (sum' (+ i 1)
                           (+ subtotal (f i)))))))
    (sum' low 0)))
```

Recursion: Example (2)

Compute the Fibonacci numbers: $F_n = \begin{cases} 1 & n \in \{0, 1\} \\ F_{n-1} + F_{n-2} & \text{otherwise} \end{cases}$

```
(define (fib n)
  (cond ((= n 0) 1)
        ((= n 1) 1)
        (#t (+ (fib (- n 1))
               (fib (- n 2))))))
```

Recursion: Example (2)

Compute the Fibonacci numbers: $F_n = \begin{cases} 1 & n \in \{0, 1\} \\ F_{n-1} + F_{n-2} & \text{otherwise} \end{cases}$

```
(define (fib n)
  (cond ((= n 0) 1)
        ((= n 1) 1)
        (#t (+ (fib (- n 1))
               (fib (- n 2))))))
```

Tail-recursive version:

```
(define (fib n)
  (letrec ((fib' (lambda (f1 f2 i)
                   (if (= i n)
                     f2
                     (fib' f2 (+ f1 f2) (+ i 1))))))
    (fib' 0 1 0)))
```

Recursion: Example (2)

Compute the Fibonacci numbers: $F_n = \begin{cases} 1 & n \in \{0, 1\} \\ F_{n-1} + F_{n-2} & \text{otherwise} \end{cases}$

```
(define (fib n)
  (cond ((= n 0) 1)
        ((= n 1) 1)
        (#t (+ (fib (- n 1))
               (fib (- n 2))))))
```

Tail-recursive version:

```
(define (fib n)
  (letrec ((fib' (lambda (f1 f2 i)
                   (if (= i n)
                     f2
                     (fib' f2 (+ f1 f2) (+ i 1))))))
    (fib' 0 1 0)))
```

Tail-recursive functions imitate iteration.

Recursion: Example (2)

Compute the Fibonacci numbers: $F_n = \begin{cases} 1 & n \in \{0, 1\} \\ F_{n-1} + F_{n-2} & \text{otherwise} \end{cases}$

```
(define (fib n)
  (cond ((= n 0) 1)
        ((= n 1) 1)
        (#t (+ (fib (- n 1))
               (fib (- n 2))))))
```

Tail-recursive version:

```
(define (fib n)
  (letrec ((fib' (lambda (f1 f2 i)
                   (if (= i n)
                     f2
                     (fib' f2 (+ f1 f2) (+ i 1))))))
    (fib' 0 1 0)))
```

Tail-recursive functions imitate iteration.
In the absence of side effects, they use linear space.

Recursion: Example (2)

Compute the Fibonacci numbers: $F_n = \begin{cases} 1 & n \in \{0, 1\} \\ F_{n-1} + F_{n-2} & \text{otherwise} \end{cases}$

```
(define (fib n)
  (cond ((= n 0) 1)
        ((= n 1) 1)
        (#t (+ (fib (- n 1))
               (fib (- n 2))))))
```

Tail-recursive version:

```
(define (fib n)
  (letrec ((fib' (lambda (f1 f2 i)
                   (if (= i n)
                     f2
                     (fib' f2 (+ f1 f2) (+ i 1))))))
    (fib' 0 1 0)))
```

Tail-recursive functions imitate iteration.
In the absence of side effects, they use linear space.

Using garbage collection, the space bound is in fact constant.

Applicative and Normal Order of Evaluation

Applicative-order evaluation

- Arguments are evaluated before a subroutive call.
- Default in most programming languages.

Normal-order evaluation

- Arguments are passed unevaluated to the subroutine. The subroutine evaluates them as needed.
- Useful for infinite or lazy data structures that are computed as needed.
- Examples: Macros in C, Haskell
- Fine in functional languages, problematic if there are side effects. Why?
- Potential for inefficency. Why? How can this be avoided?

Lazy Evaluation in Scheme

By default, Scheme uses ***strict*** evaluation.

This code runs forever

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

Lazy Evaluation in Scheme

By default, Scheme uses ***strict*** evaluation.

Using lazy evaluation, this can be avoided:

```
(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
```

Lazy Evaluation in Haskell

Lazy evaluation is the default in Haskell. If maximum efficiency is needed, strict evaluation of function arguments and data structure members can be forced.

```
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 in Haskell

Lazy evaluation is the default in Haskell. If maximum efficiency is needed, strict evaluation of function arguments and data structure members can be forced.

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

> take 10 naturals
(runs forever)
```

Implementation of DELAY and FORCE (1)

`delay` can be 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))
```

Implementation of DELAY and FORCE (1)

`delay` can be 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` can be 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 `delayed-exp` every time. This is inefficient.

Implementation of DELAY and FORCE (2)

A better implementation:

```
(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)))
```

Implementation of DELAY and FORCE (2)

A better implementation:

```
(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 by default.

Example of Normal-Order Evaluation: C/C++ Macros (1)

Example

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

Example of Normal-Order Evaluation: C/C++ Macros (1)

Example

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

Problems

- Cannot be used recursively

Example of 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: Compare

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

  vs

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

  and evaluate `DIVIDES(x, y + 2)`.

Example of Normal-Order Evaluation: C/C++ Macros (2)

- Side effects: Consider

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

  and evaluate `MAX(x++, y++)`.

Example of Normal-Order Evaluation: C/C++ Macros (2)

- Side effects: Consider

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

  and evaluate `MAX(x++, y++)`.
- Name clashes with variables: Consider

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

  and evaluate `SWAP(x, t)`.

Example of Normal-Order Evaluation: C/C++ Macros (2)

- Side effects: Consider

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

  and evaluate `MAX(x++, y++)`.
- Name clashes with variables: Consider

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

  and evaluate `SWAP(x, t)`.

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