Structured Control Flow

All modern higher-level imperative languages are designed to support **structured programming**.

Loosely, a structured program is one in which the **syntactic structure** of the program text corresponds to the **flow of control** through the dynamically executing program.

Originally proposed (most famously by Dijkstra) as an improvement on the incomprehensible “spaghetti code” that is easy to produce using the labels and jumps supported directly by hardware.

More specifically, structured programs use a very small collection of (recursively defined) **compound statements** to describe their control flow.

Compounds are of three kinds:

- **Sequential composition**: form a statement from a sequence of statements, e.g.
  (Java) \{ \ x = 2; \ y = x + 4; \} 
  (Pascal) `begin x := 2; y := x + 4; end`

- **Selection**: execute one of several statements, e.g.,
  (Java) `if (x < 0) y = x + 1; else z = y + 2;`

- **Iteration**: repeatedly execute a statement, e.g.,
  (Java) `while (x > 10) output (x--);`
  (Pascal) `for x := 1 to 12 do output (x*2);`
Selection

The basic selection statement is based on boolean values

\[ \text{if } e \text{ then } s_1 \text{ else } s_2 \]

which translates to

\[
\begin{align*}
\text{evaluate } e \text{ into } t \\
\text{cmp } t, \text{true} \\
\text{brneq } l_1 \\
\text{ } s_1 \\
\text{br } l_2 \\
\end{align*}
\]

\[ l_1: \quad s_2 \\
 l_2: \quad \]

To test types with more than two values, multi-way selections against constants are appropriate:

\[
\begin{align*}
\text{case } e \text{ of} \\
\quad c_1 : s_1 \\
\quad c_2 : s_2 \\
\quad \ldots \\
\quad c_n : s_n \\
\quad \text{default : } s_d
\end{align*}
\]

The most efficient translation of \texttt{case} statements depends on \textbf{density} of the value \( c_1, c_2, \ldots, c_n \) within the range of possible values for \( e \).
Sparse Cases

For sparse distributions, it’s best to translate the case just as if it were:

\[
\begin{align*}
t & := e; \\
& \text{if } t = c_1 \text{ then } \\
& \hspace{1em} s_1 \\
& \text{else if } t = c_2 \text{ then } \\
& \hspace{1em} s_2 \\
& \text{else} \\
& \hspace{1em} \ldots \\
& \text{else if } t = c_n \text{ then } \\
& \hspace{1em} s_n \\
& \text{else} \\
& \hspace{1em} s_d
\end{align*}
\]
Dense Cases

For a dense set of labels in the range $[c_1, c_n]$, it's better to use a jump table:

```plaintext
evaluate e into t

    cmp  t, c_1
    brlt  l_d
    cmp  t, c_n
    brgt  l_d
    sub  t, c_1, t
    add  table, t, t
    br  *t

table:  l_1
         l_2
         ...
         l_n

l_1:    s_1
        br  done

l_2:    s_2
        br  done
        ...

l_n:    s_n
        br  done

l_d:    s_d
        br done

done:
```

The best approach for a given case may involve a combination of these two techniques. Compilers differ widely in the quality of the code generated for case.
Iteration

The basic loop construct is

\[
\text{while } e \text{ do } s
\]

corresponding to:

\[
\begin{align*}
\text{top:} & \quad \text{evaluate } e \text{ into } t \\
& \quad \text{cmp } t, \text{true} \\
& \quad \text{brneq } \text{done} \\
& \quad s \\
& \quad \text{br } \text{top} \\
\text{done:}
\end{align*}
\]

A commonly-supported variant is to move the test to the bottom:

\[
\text{repeat } s \text{ until } e
\]

which is equivalent to:

\[
\begin{align*}
& \quad s; \\
& \quad \text{while not } e \text{ do } s
\end{align*}
\]
Loop exits

It is sometimes desirable to exit from the middle of a loop:

\[
\text{loop } \\
\quad s_1; \\
\quad \text{exitif } e; \\
\quad s_2 \\
\text{end}
\]

is equivalent to:

\[
\text{top: } s_1 \\
\quad \text{evaluate } e \text{ into } t \\
\quad \text{cmp } t, \text{true} \\
\quad \text{breq done} \\
\quad s_2 \\
\quad \text{br top} \\
\text{done:}
\]

C/C++/Java have an unconditional form of `exit`, called `break`. They also have a `continue` statement that jumps back to the top of the loop.
Uses for goto?

An efficient program with goto:

```java
int i;
for (i = 0; i < n; i++)
    if (a[i] == k)
        goto found;
    n++;
a[i] = k;
b[i] = 0;
found:
b[i]++;
```

In most languages (e.g., Modula, C/C++) there is no equivalently efficient solution without goto.

But we can do as well in Java, using a named, multi-level break:

```java
int i;
search:
{ for (i = 0; i < n; i++)
    if (a[i] == k)
        break search;
    n++;
a[i] = k;
b[i] = 0;
}
b[i]++;
```
The COME FROM statement

```
10 J = 1
11 COME FROM 20
12 PRINT J
   STOP
13 COME FROM 10
20 J = J + 2
```


But is this really a joke?

Even with a GO TO, we must examine both the branch and the target label to understand the programmer’s intent.
Counted loops

Since iterating a definite number of times is very common, languages often offer a dedicated statement, with basic form:

\[
\text{for } i := e_1 \text{ to } e_2 \text{ do } s
\]

Here \( s \) is executed repeatedly with \( i \) taking on the values \( e_1, e_1 + 1, \ldots, e_2 \) in each successive iteration.

The detailed semantics of this statement vary, and can be tricky. Often, \( s \) is prohibited from modifying \( i \), which (under certain other conditions) guarantees that the loop will be executed exactly \( e_2 - e_1 + 1 \) times.

C/C++/Java have a much more general version of \texttt{for}, which guarantees much less about the behavior of the loop:

\[
\text{for } (e_1; e_2; e_3) \ s;
\]

is exactly equivalent to:

\[
e_1;
\text{while } (e_2) \{ \\
\quad s;
\quad e_3 \ }
\]
Fun with C

Problem: Sending characters to an output device as quickly as possible.

Given:

```c
char p[] = "hello world...";
char *m = p;
int n = ... /* length of p */
#define output(c) ... /* do output */
```

Solution 1:

```c
for (i = 0; i < n; i++)
    output(*m++);
```

Faster (maybe):

```c
if (n) do
    output(*m++)
while (--n);
```

(Avoids compare with \(n\) each time.)
More fun

Faster to **unroll** loop, say 4 times:

```c
while (n & 3) {
    output(*m++);
    --n;
}
```

```
n /= 4;
if (n) do { output (*m++);
    output (*m++);
    output (*m++);
    output (*m++);
} while (--n);
```

Or (the Duff Loop):

```
i = (n+3)/4;
if (n) switch (n & 3) {
    case 0: do {output(*m++);
    case 3: output(*m++);
    case 2: output(*m++);
    case 1: output(*m++)
        while (--i);
}
```

“This is the most amazing piece of C I’ve ever seen.” – Ken Thompson

Does this work in Java?
Systematic Removal of Recursion

(Adapted from Sedgewick, *Algorithms*, 2nd ed.. Examples in C.)

Original program:

```c
typedef struct tree *Tree;
struct tree {
    float value;
    Tree left;
    Tree right;
};

float sumtree(Tree t) {
    float sum;
    if (t)
        sum = t->value
            + sumtree(t->left)
            + sumtree(t->right);
    else
        sum = 0.0;
    return sum;
}
```
Step 1:

Change functions to procedures by “globalizing” `sum` variable.

```c
float sum;
void traverse(Tree t) {
  if (t) {
    sum += t->value;
    traverse(t->left);
    traverse(t->right);
  }
}
float sumtree(Tree t) {
  sum = 0.0;
  traverse(t);
  return sum;
}
```
Step 2:

Remove tail-recursion.

```c
float sum;
void traverse(Tree t) {
    top:
    if (t) {
        sum += t->value;
        traverse(t->left);
        t = t->right;
        goto top;
    }
}
float sumtree(Tree t) {
    sum = 0.0;
    traverse(t);
    return sum;
}
```

Step 3:

Use explicit stack to replace non-tail recursion. Simulate behavior of compiler by pushing local variables and return address onto the stack before call and popping them back off the stack after call.

Here there is only one local variable (t) and the return address is always the same, so there’s no need to save it.
Stack empty;
void push(Stack s, Tree t);
Tree pop(Stack s);
int isEmpty(Stack s);

float sum;
void traverse(Tree t) {
    Stack s = empty;
    top:
    if (t) {
        sum += t->value;
        push(s, t);
        t = t->left;
        goto top;
    }
    if (!(isEmpty(s))) {
        t = pop(s);
        goto retaddr;
    }
}
float sumtree(Tree t) {
    sum = 0.0;
    traverse(t);
    return sum;
}
Step 4:

Simplify by:

- Rearranging to avoid the retaddr label.
- Pushing right child instead of parent on stack.
- Replacing first goto with a while loop.
- Inlining traverse routine (now non-recursive) and re-localizing sum variable.

```c
float sumtree(Tree t) {
    Stack s = empty;
    float sum = 0.0;
    top:
    while (t) {
        sum += t->value;
        push(s, t->right);
        t = t->left;
    }
    if (!(isEmpty(s))) {
        t = pop(s);
        goto top;
    }
    return sum;
}
```
Step 5:

Rearrange some more to replace outer *goto* with another *while* loop.

(This is slightly wasteful, since an extra *push, stackempty* check and *pop* are performed on root node.

```c
float sumtree(Tree t) {
    Stack s = empty;
    float sum = 0.0;
    push(s,t);
    while(!(isEmpty(s))) {
        t = pop(s);
        while (t) {
            sum += t->value;
            push(s,t->right);
            t = t->left;
        }
    }
    return sum;
}
```
Step 6:

A more symmetric version can be obtained by pushing and popping the left children.

Compare this to the original recursive program.

```c
float sumtree(Tree t) {
    Stack s = empty;
    float sum = 0.0;
    push(s,t);
    while(!(isEmpty(s))) {
        t = pop(s);
        if (t) {
            sum += t->value;
            push(s,t->right);
            push(s,t->left);
        }
    }
    return sum;
}
```
Step 7:

We can also test for empty subtrees before we push them on the stack rather than after popping them.

```c
float sumtree(Tree t) {
    Stack s = empty;
    float sum = 0.0;
    if (t) {
        push(s, t);
        while(!(isEmpty(s))) {
            t = pop(s);
            sum += t->value;
            if (t->right)
                push(s, t->right);
            if (t->left)
                push(s, t->left);
        }
    }
    return sum;
}
```
Stack Machines

Very simple machine architecture, in which instruction operands are taken from stack and results are put back on stack.

- Natural for handling arithmetic expressions (Reverse Polish Notation).
- Natural for handling recursive functions (arguments, locals, return values, return addresses).
- Allows very compact instruction encoding (e.g., byte code).
- Used in abstract machines and virtual machines (e.g., JVM).

Sample instruction set (from homework):

\[
\begin{align*}
\text{CONST int, ADD, SUB, MUL, DIV, POP} \\
\text{GOTO addr, BRANCHEQ addr, BRANCHLT addr} \\
\text{LOAD var, STORE var, EXTEND var, RETRACT} \\
\text{CALL func}
\end{align*}
\]

Sample code for set \( x = (3 * x) + g(8) \):

\[
\begin{align*}
\text{CONST 3} \\
\text{LOAD x} \\
\text{MUL} \\
\text{CONST 8} \\
\text{CALL g} \\
\text{ADD} \\
\text{STORE x} \\
\text{LOAD x}
\end{align*}
\]
3 + (if x = 2 then y+4 else z)

0:  CONST 3
1:  LOAD x
2:  CONST 2
3:  BRANCHEQ 5
4:  GOTO 9
5:  LOAD y
6:  CONST 4
7:  ADD
8:  GOTO 10
9:  LOAD z
10: ADD

let x = 1
in begin let x = 2 in x;
   x
end

CONST 1
EXTEND x
STORE x
CONST 2
EXTEND x
STORE x
LOAD x
RETRACT
POP
LOAD x
RETRACT