Control structures, first lecture

The birth of control structures:
from “goto” to structured programming

Xavier Leroy
2024-01-25

Collège de France, chair of software sciences
xavier.leroy@college-de-france.fr
First computers, first languages
An architecture described in 1945 by John von Neumann and semi-independently by Alan Turing.

Used by most computers designed since then.
A “PC” register, the program counter, contains the address of the memory word containing the next instruction.

Regular instructions increment the PC → sequential execution.

Branch instructions set the value of the PC → jump to any program point.

Example: output the numbers 1!, 2!, ..., n!, ...

0: set r1, 1  (r1 = the value of n!)
4: set r2, 1  (r2 = the value of n)
8: output r1
12: add r2, r2, 1
16: mul r1, r1, r2
20: branch 8
Conditional branches

Modern processors provide conditional branch instructions that
• set the PC if a condition is true (e.g. if a register contains a non-zero value);
• continue in sequence if the condition is false.

Example: a counted loop that computes $n!$

0: set r2, 1
4: mul r2, r2, r1
8: sub r1, r1, 1
12: brnz r1, 4

(r1 = the value of n)

(r2 = the value of n!)
Neither von Neumann nor Turing consider conditional branches. Instead, they rely on the code being stored in memory and modifiable during the execution of the program!

Example: branch to 0 if \( r1 = 0 \) and to 4 if \( r1 = 1 \).
We assume that the branch \( n \) instruction is encoded as \( 0x76000000 + n \).

184: set r2, 0x76000000
188: mul r3, r1, 4
192: add r2, r2, r3
196: store r2, 200
200: nop  \hspace{1cm} \text{(dynamically-modified instruction)}
An encoding of instructions and their operands as binary numbers (often 32-bit wide).

Example: conditional branch in the Power architecture.

**bc**

Branch Conditional [and Link] [Absolute]

<table>
<thead>
<tr>
<th>Instruction</th>
<th>BO, BI, BD</th>
</tr>
</thead>
<tbody>
<tr>
<td>bc</td>
<td>(AA=0, LK=0)</td>
</tr>
<tr>
<td>bca</td>
<td>(AA=1, LK=0)</td>
</tr>
<tr>
<td>bcl</td>
<td>(AA=0, LK=1)</td>
</tr>
<tr>
<td>bcla</td>
<td>(AA=1, LK=1)</td>
</tr>
</tbody>
</table>

if ¬BO₂ then CTR ← CTR - 1
ctr_ok ← BO₂ || ((CTRₘ:₆₃ ≠ 0) ⊕ BO₃)
cond_ok ← BO₀ || (CRₜₙₙₙₚ₃₂ ≡ BO₁)
if ctr_ok & cond_ok then
  if AA=1 then a ← ⁶₄₀ else a ← CIA
  NIA ← m₀ || (a + EXTS(BD||0b00))ₘ:₆₃
else
  NIA ← m₀ || (CIA + 4)ₘ:₆₃
if LK=1 then LR ← CIA + 4

For us humans: hard to write... and impossible to read!
Program flowcharts

A graphical, often informal representation of programs.

To produce machine language from a flowchart:
place instructions in memory; add branches; encode instructions.
A textual representation of machine language:

- **mnemonics** to name processor instructions;
- **labels** to name program points;
- **comments** to document the code.

Example: a counted loop that computes $n!$

; Compute factorial of $n$.
; $n$ is passed in r1.
; $n!$ is left in r2.
; r1 is clobbered.

    set r2, 1 ; initialize result to 1
again:    mul r2, r2, r1 ; multiply result by $n$
    sub r1, r1, 1 ; decrement $n$
    brnz r1, again ; repeat until $n = 0$

; here, result is $n!$
Macro-assemblers and “autocoders” (IBM, 1955)

Pseudo-instructions that represent often-used instruction sequences. Expanded by the assembler.

Example: a counted loop.

<table>
<thead>
<tr>
<th>Source code</th>
<th>Code after expansion</th>
</tr>
</thead>
<tbody>
<tr>
<td>BEGIN_LOOP(lbl, r1, 0)</td>
<td>set r1, 0</td>
</tr>
<tr>
<td>...</td>
<td>lbl:</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>END_LOOP(lbl, r1, 100)</td>
<td>add r1, r1, 1</td>
</tr>
<tr>
<td></td>
<td>cmp r1, 100</td>
</tr>
<tr>
<td></td>
<td>brlt lbl</td>
</tr>
</tbody>
</table>
The first control structures
The FORTRAN language (1957)

Complex expressions, using familiar algebraic notations, automatically translated to machine instructions (FORmula TRANslator).

**FORTRAN source:**

\[
\begin{align*}
D &= \sqrt{B^2 - 4AC} \\
X1 &= \frac{-B + D}{2A} \\
X2 &= \frac{-B - D}{2A}
\end{align*}
\]

**Assembly code:**

\[
\begin{align*}
mul \ t1, b, b & \quad sub \ x1, d, b \\
mul \ t2, a, c & \quad div \ x1, x1, t3 \\
mul \ t2, t2, 4 & \quad neg \ x2, b \\
sub \ t1, t1, t2 & \quad sub \ x2, x2, d \\
sqrt \ d, t1 & \quad div \ x2, x2, t3 \\
mul \ t3, a, 2 & 
\end{align*}
\]

In addition to expressions: the first control structure!
Control in FORTRAN I

Counted loops:

\begin{verbatim}
    DO 100 I = 1, N
        ...
        ...
    100: ...
\end{verbatim}

Repeatedly executes the lines between `DO` and `100` (excluded) with `I` taking successive values 1, 2, ..., `N`.

Efficient compilation: test at end of loop, loop unrolling, etc.
Labels and GO TO.

Three-way conditional branch \( \text{IF } X \text{ L1, L2, L3} \) (branch whether \( X < 0 \) or \( X = 0 \) or \( X > 0 \)).

\[
\text{IF } X \text{ 701, 702, 702} \\
701: X = -X \\
702: \ldots
\]

Branching to a computed label.

\[
\text{ASSIGN 100 TO DEST} \\
\ldots \\
\text{GO TO DEST}
\]


\[
\text{IF } X \text{ .LT. Y GOTO 100}
\]
Structured commands in Algol (1960)

Expressions are formed from constants and variables. Likewise, Algol promotes the view that commands (statements) are constructed from elementary commands

- assignment: \( x := expr \)
- procedure call: \( proc(arg1, arg2) \)

combining them using control structures

- sequence (“blocks”): \( \text{begin } s_1; s_2; \ldots \text{ end} \)
- conditional: \( \text{if } be \text{ then } s_1 \text{ else } s_2 \)
- counted loop: \( \text{for } i := e_1 \text{ step } e_2 \text{ until } e_3 \text{ do } s \)
- general loop: \( \text{while } be \text{ do } s \)
A change of perspective on programs

At the syntax level: formal grammars such as “Backus-Naur form” (BNF); recursive algorithms for parsing.

Commands:

\[ s ::= x := e \]

  \[ \text{proc}(e_1, \ldots, e_n) \]
  \[ \text{begin } s_1; \ldots; s_n \text{ end} \]
  \[ \text{if } be \text{ then } s_1 \text{ [else } s_2] \]
  \[ \text{while } be \text{ do } s \]
  \[ \text{for } i := e_1 \text{ step } e_2 \text{ until } e_3 \text{ do } s \]

At the semantic level: (intuitive first, later formal)
a realization that the meaning of a command is entirely determined by that of its sub-commands.
Unstructured control in Algol

Algol also supports unstructured control, with jumps to labels that designate commands:

\[
S ::= \ldots
\]

- \( L : s \) command labeled \( L \)
- \( \text{go to } L \mid \text{goto } L \mid \text{go } L \) jump to label \( L \)

A label has a **scope**, which is the block enclosing the definition of the label. We cannot jump into a block from outside the block.

### Jump within ✔

```
begin
  integer i;
L:...
  goto L
end
```

### Jump out ✔

```
begin
  integer i;
L:...
  ... goto L ...
end;
```

### Jump into ✗

```
goto L;
begin
  integer i;
L:...
end
```
Control structures + “goto”

This “Algol style” with structured control and goto quickly became the standard to describe and publish algorithms (since 1960 in Comm. ACM), instead of flowcharts.

ALGORITHM 95
GENERATION OF PARTITIONS IN PART-COUNT FORM
FRANK STOCKMAL
System Development Corp., Santa Monica, Calif.

procedure partgen(c,N,K,G); integer N,K; integer array c; Boolean G;
comment This procedure operates on a given partition of the positive integer N into parts ≤ K, to produce a consequent partition if one exists. Each partition is represented by the integers c[1] thru c[K], where c[j] is the number of parts of the partition equal to the integer j. If entry is made with G = false, procedure ignores the input array c, sets G = true, and produces the first partition of N ones. Upon each successive entry with G = true, a consequent partition is stored in c[1] thru c[K]. For N = KX, the final partition is c[K] = X. For N = KX+r, 1 ≤ r ≤ K-1, final partition is c[K] = X, c[r] = 1. When entry is made with array c = final partition, c is left unchanged and G is reset to false;

begin integer a,i,j;
    if ~ G then go to first;
    j := 2;
    a := C[1];
test: if a < j then go to B;
        c[j] := 1 + c[j];
        c[1] := a - j;
zero: for i := 2 step 1 until j - 1
            do c[i] := 0;
            go to EXIT;
B: if j = K then go to last;
    a := a + j * c[j];
    j := j + 1;
    go to test;
first: G := true;
        c[1] := N;
        j := K + 1;
        go to zero;
last: G := false;
EXIT: end partgen
We find this combination of "goto" and control structures in many imperative and OO languages: Algol 68, Algol W, Pascal, Ada, Simula, PL/I, C, C++, C#, Perl, Go, …

Languages with "goto" and counted loops, such as FORTRAN and BASIC, evolved by adopting Algol-style control structures (if...else... in FORTRAN 77, do while in FORTRAN 90).

Other flavors of conditionals

Cascading Boolean tests:

if \( be_1 \) then \( s_1 \) elsif \( be_2 \) then \( s_2 \) elsif \ldots\) else \( s_n \)

also written \((\text{cond} (be_1 s_1) (be_2 s_2) \ldots (t s_n))\) in Lisp.

Case analysis on a value of integer type or enumerated type:

\[
\begin{align*}
\text{case (grade) of} & \\
'A' & : s_1 \\
'B', 'C' & : s_2 \\
'D' & : s_3 \\
'F' & : s_4 \\
\text{end}
\end{align*}
\]

\[
\begin{align*}
\text{switch (grade) \{} & \\
\text{case 'A': } & s_1; \text{ break;} \\
\text{case 'B':} & \\
\text{case 'C': } & s_2; \text{ break;} \\
\text{case 'D': } & s_3; \text{ break;} \\
\text{case 'F': } & s_4; \text{ break;} \\
\text{\}}
\end{align*}
\]
Other flavors of loops

General loops:

- test at beginning: while be do s
- test at end: do s while be or repeat s until be
- test in the middle: loop ...exit if be...end

Counted loops with early exit conditions: (PL/I, Algol 68)

```
[FOR index] [FROM first] [BY increment] [TO last] [WHILE condition]
DO statements OD
```

Loops that iterate over a collection of values:

```
for item in collection: s            (Python)
for (Type item : collection) { s }   (Java)
```
Early exits from loops

Exit from the enclosing loop:

\begin{itemize}
\item break (C, C++, Java) / exit (Ada) / last (Perl)
\end{itemize}

Abort the current iteration and start the next one:

\begin{itemize}
\item continue (C, C++, Java) / next (Perl)
\end{itemize}

Multi-level exit for nested loops:

\begin{itemize}
\item exit from the N-th enclosing loop \quad break N (Shell)
\item exit from the loop labeled L \quad break L (Java)
\end{itemize}

\begin{verbatim}
xloop: for (int x = 0; x < dimx; x++)
    yloop: for (int y = 0; y < dimy; y++) {
        ... break xloop ... break yloop ...
    }
\end{verbatim}
Structured programming, structured control
The movement for structured programming (1965–1975)

A radical change of perspective on software.

Abandon the view of programs as

flowcharts and machine code

and start viewing programs as

a constructed, structured source text,
directly legible (without a flowchart on the side),
over which we can reason
(informally, then mathematically).

A dispute between two programming styles:

- Programs using many “goto”, obtained by naive transcription of a flowchart.
- Programs using mainly control structures (conditionals, loops), written directly, without a flowchart.

The slogan for this dispute:

*Go to statement considered harmful*

(Title of a short communication by Dijkstra in CACM 1968. The title was chosen not by the author but by the CACM editors.)
Since the summer of 1960, I have been writing programs in outline form, using conventions of indentation to indicate the flow of control. I have never found it necessary to take exception to these conventions by using go statements.

I used the keep these outlines as original documentation of a program, instead of using flow charts ... Then I would code the program in assembly language from the outlines. Everyone liked these outlines better than the flow charts.

(Dewey Val Schorre, 1966)
If you look carefully you will find that surprisingly often a \texttt{go to} statement which looks back really is a concealed \texttt{for} statement. And you will be pleased to find how the clarity of the algorithm improves when you insert the \texttt{for} clause where it belongs.

If the purpose [of a programming course] is to teach Algol programming, the use of flow diagrams will do more harm than good, in my opinion.

(Peter Naur, BIT, 1963).
Go To Statement Considered Harmful

Key Words and Phrases: go to statement, jump instruction, branch instruction, conditional clause, alternative clause, repetitive clause, program intelligibility, program sequencing

CR Categories: 4.22, 5.23, 5.24

EDITOR: For a number of years I have been familiar with the observation that the quality of programmers is a decreasing function of the density of go to statements in the programs they produce. More recently I discovered why the use of the go to statement has such disastrous effects, and I became convinced that the go to statement should be abolished from all "higher level" programming languages (i.e. everything except, perhaps, plain machine code). At that time I did not attach too much importance to this discovery; I now submit my considerations for publication because in very recent discussions in which the subject turned up, I have been urged to do so.


A reflection on the difficulty to know "where we are" in a program execution, and which path was taken from the beginning.
Which “coordinates” to designate a point in the execution?

```java
if (x == 0) {
    ...
} else if (y == 0) {
    ...
        ⇐ YOU ARE HERE
} else {
    ...
    ...
}
```

Sequence and conditionals:
current program point + Boolean values of previous conditionals.
Which “coordinates” to designate a point in the execution?

```java
while (x < y) {
    while (a[x] != 0) {
        if (x == 0) {
            ...
        } else if (y == 0) {
            ...
                ⇐ YOU ARE HERE
        } else {
            ...

    }
}
```

Sequence, conditionals, and loops:
program point + Boolean values + iteration counts for all loops.
Which “coordinates” to designate a point in the execution?

```c
while (x < y) {
    ⇐ 2nd iteration
    while (a[x] != 0) {
        ⇐ 5th iteration
        if (x == 0) {
            L:...
        } else if (y == 0) {
            ...
            ⇐ YOU ARE HERE
        } else {
            ...
            ; if (y < 10) goto L;
            ...
        }
    }
    } else if (y == 0) {
        ...
        ⇐ YOU ARE HERE
    }
    } else {
        ...
        ; if (y < 10) goto L;
        ...
    }
}
```

With unrestricted `goto`: it becomes terribly hard to find a meaningful set of coordinates in which to describe the process progress (Dijkstra).
Food for thought!

The tide of opinions first hit me personally in 1969, when I was teaching an introductory programming course for the first time. I remember feeling frustrated at not seeing how to write programs in the new style; I would run to Bob Floyd’s office asking for help, and he usually showed me what to do.

(D. E. Knuth, 1974)

After Dijkstra’s paper, a great many studies

- to see whether goto-free programming is really the best way to express program structure;
- to find out how to eliminate goto, on a case-by-case basis or systematically by program transformation.

After a historical account of the controversy, Knuth studies, on well-chosen examples,

- the impact of go to elimination on code clarity (especially if Boolean flags must be added);
- the impact of go to elimination on performance;
- which control structures (beyond those of Algol) could help expressing program structure better.
Example: a hash table

Two arrays: \( A[N] \) containing the keys, \( B[N] \) containing the values.

```c
void add(key k, data d)
{
    int i = hash(k);
    while (1) {
        if (A[i] == 0) goto notfound;
        if (A[i] == k) goto found;
        i = i + 1; if (i >= N) i = 0;
    }

    notfound: A[i] = k;
    found: B[i] = d;
}
```

Very concise code! But there are two `goto`...
Example: a hash table

Using a while loop:

```c
void add(key k, data d)
{
    int i = hash(k);
    while (! (A[i] == 0 || A[i] == k)) {
        i = i + 1; if (i >= N) i = 0;
    }
    if (A[i] == 0) A[i] = k;
    B[i] = d;
}
```

The test `A[i] == 0` is duplicated.
Using a loop with early exits (break):

```c
void add(key k, data d)
{
    int i = hash(k);
    while (1) {
        if (A[i] == 0) { A[i] = k; B[i] = d; break; }
        if (A[i] == k) { B[i] = d; break; }
        i = i + 1; if (i >= N) i = 0;
    }
}
```

Minor duplication of the code `B[i] = d`, but no impact on performance.
Example: error handling with resource freeing

A system programming idiom in C.

```c
int retcode = -1;  // error
int fd = open(filename, O_RDONLY);
if (fd == -1) goto err1;
char * buf = malloc(bufsiz);
if (buf == NULL) goto err2;
...
if (something goes wrong) goto err3;
...
retcode = 0;  // success
err3: free(buf);
err2: close(fd);
err1: return retcode;
```

Avoids any duplication of the code that frees resources.
A version without goto that suffers from code duplication:

```c
int fd = open(filename, O_RDONLY);
if (fd == -1) return -1;
char * buf = malloc(bufsiz);
if (buf == NULL) { close(fd); return -1; }
...
if (something goes wrong) {
    free(buf); close(fd); return -1;
}
...
free(buf); close(fd); return 0;
```
Example: error handling with resource freeing

A version based on blocks `do ... while(0)` and on `break`. Would be more robust with a multi-level `break`.

```c
int retcode = -1; // error
do { int fd = open(filename, O_RDONLY);
    if (fd == -1) break;
    do { char * buf = malloc(bufsiz);
        if (buf == NULL) break;
        do { ...
            if (something goes wrong) break;
        } while (0); free(buf);
    } while (0); close(fd);
} while(0); return retcode;
```
Expressiveness of structured control
A technical question that plays a central role in the structured control controversy.

Is it always possible to transform an unstructured program (using goto and if...goto) into an equivalent structured program?

With or without code duplication?

With or without introducing additional variables?
A well-known result

**Theorem**

Any unstructured program (or, equivalently, any flowchart) is equivalent to a structured program comprising a single **while** loop and one extra integer variable.
A simple proof of the well-known result

1: \(s_1;\)
   \(\text{if } (b_1) \text{ goto } 3;\)

2: \(s_2;\)
   \(\text{goto } 4;\)

3: \(s_3;\)

4: \(s_4;\)
   \(\text{if } (b_4) \text{ goto } 1;\)
A simple proof of the well-known result

1: \( s_1; \)
   \[ \text{if} \ (b_1) \ \text{goto} \ 3; \ \text{else} \ \text{goto} \ 2; \]
2: \( s_2; \)
   goto 4;
3: \( s_3; \)
   goto 4;
4: \( s_4; \)
   \[ \text{if} \ (b_4) \ \text{goto} \ 1; \ \text{else} \ \text{goto} \ 0; \]

Reformat the program as **basic blocks**: sequences of assignments terminated by branches, either \text{goto} \ L or \text{if} \ be \ \text{then} \ \text{goto} \ L_1 \ \text{else} \ \text{goto} \ L_2.
A simple proof of the well-known result

1: $s_1$;
   if ($b_1$) goto 3; else goto 2;
2: $s_2$;
   goto 4;
3: $s_3$;
   goto 4;
4: $s_4$;
   if ($b_4$) goto 1; else goto 0;

Replace labels by numbers 0, 1, 2, ..., with 0 being the label for the end of the program.
A simple proof of the well-known result

1: $s_1$;
   if ($b_1$) $pc = 3$; else $pc = 2$;
2: $s_2$;
   $pc = 4$;
3: $s_3$;
   $pc = 4$;
4: $s_4$;
   if ($b_4$) $pc = 1$; else $pc = 0$;

Replace each `goto L` by an assignment $pc = L$ where $pc$ is a new integer variable.
A simple proof of the well-known result

switch (pc) {
    case 1: s_1;
        if (b_1) pc = 3; else pc = 2; break;
    case 2: s_2;
        pc = 4; break;
    case 3: s_3;
        pc = 4; break;
    case 4: s_4;
        if (b_4) pc = 1; else pc = 0; break;
}

Turn each basic block into one case of a `switch` over the value of `pc`.
int pc = 1; while (pc != 0) {
    switch (pc) {
    case 1:  
        s1;
        if (b1) pc = 3; else pc = 2; break;
    case 2: 
        s2;
        pc = 4; break;
    case 3: 
        s3;
        pc = 4; break;
    case 4: 
        s4;
        if (b4) pc = 1; else pc = 0; break;
    }
}

Add a while loop to iterate the switch until pc = 0.
Iterating a transition function

If we view the original flowchart / unstructured program as a finite automaton, the previous proof amounts to constructing its transition function

\[
\text{current state (in } pc\text{)} \rightarrow \text{next state (in } pc\text{)}
\]

as a case analysis on the value of \( pc \)

\[
\text{switch (pc) } \{ \ldots \}
\]

then to iterating this function until the final state is reached

\[
\text{while (pc != 0) } \{ \ldots \}
\]

Variants:

- one integer variable \( pc \) → several Boolean variables
- one switch → a cascade of if...then...else.
The folk lore around this result

(David Harel, *On Folk Theorems*, CACM 23(7), 1980)

The result is often attributed to Böhm and Jacopini, *Flow diagrams, Turing machines, and languages with only two formation rules*, CACM 1966.

However, this paper shows a different result (producing several while loops) with more subtle techniques (local graph rewriting).

---

![Fig. 13. Structure of a type I diagram](image1)

![Fig. 14. Structure of a type II diagram](image2)

![Fig. 15. Structure of a type III diagram](image3)
The simple proof of the well-known result appears in 1967 in a letter by D. C. Cooper to the CACM editors.

It is mentioned later in tens of papers and books, often attributed to Böhm and Jacopini, or without attribution...

Harel traces the result back to Kleene (1936)!
(Every partial recursive function is the “minimization” of a primitive recursive function.)
Reductions between control structures

(S. Rao Kosaraju, *Analysis of Structured Programs*, JCSS 9, 1974.)

Let $L_1$ and $L_2$ be two languages that have the same base commands (assignments, function calls, ...) but differ on their control structures.

$L_1$ is reducible to $L_2$ if for each $L_1$ program, there exists an $L_2$ program that

- has the same base commands (no code duplication);
- uses no additional variables.
Example of reduction: do-while loop

The do-while loop (with test at end of iteration) is not reducible to the while-do loop (with test at beginning). Indeed, to translate

```
do s while be
```

we must either duplicate `s`

```
begin s; while be do s end
```

or introduce a Boolean variable

```
loop := true; while loop do begin s; loop := be end
```

In contrast, do-while is reducible to while-do + break:

```
while true do begin s; if not be then break end
```
Example of reduction: acyclic flowcharts

Cycle-free flowcharts are not reducible to if-then-else.

Either we duplicate $s_2$:

```
if ... 
then if ... then $s_1$ else $s_2$
else if ... then $s_2$ else $s_3$
```
Example of reduction: acyclic flowcharts

Cycle-free flowcharts are not reducible to if-then-else.

Or we add a Boolean variable:

\[
\begin{align*}
\text{do}_s2 & := \text{false} \\
\text{if } & \ldots \\
\text{then if } & \ldots \text{ then } s_1 \text{ else do}_s2 := \text{true} \\
\text{else if } & \ldots \text{ then do}_s2 := \text{true} \text{ else } s_3; \\
\text{if } & \text{do}_s2 \text{ then } s_2
\end{align*}
\]
Example of reduction: acyclic flowcharts

Cycle-free flowcharts are not reducible to if-then-else.

But we can reduce if we have loops with early exits (break):

```
loop
  if ...
  then if ... then begin $s_1$; break end
  else if not ... then begin $s_3$; break end;
  $s_2$; break
endloop
```
Reducible control-flow graphs and structured control

(W. W. Peterson, T. Kasami, N. Tokura. *On the capabilities of while, repeat, and exit statements.* CACM, 16(8), 1973.)

The language of reducible control-flow graphs is reducible to a structured language with conditionals, infinite loops, and multi-level exits (break $N$, continue $N$).
Control-flow graphs (CFG)

A formalization of program flowcharts.

An intermediate representation used in many compilers.

A CFG = a directed graph, with

- nodes = basic blocks
- edges = jumps
  (1 outgoing edge: goto; 2 edges: if-then-else; N edges: switch)
A node $A$ **dominates** a node $B$ if $A$ is necessarily executed before $B$: any path from the root to $B$ goes through $A$.

![Diagram showing dominance relationships]

- Every node dominates itself.
- $A$ dominates $B$, $C$, $D$, $E$.
- $D$ dominates $E$.

This leads to a classification of edges $A \rightarrow B$:

- It’s a **back edge** if $B$ dominates $A$;
- It’s a **forward edge** otherwise.
A CFG is **reducible** if its forward edges form an acyclic graph.
Many equivalent definitions, in particular:
a CFG is reducible if it can reduce to a single node by repeatedly
applying transformations T1 and T2.
Loops in reducible CFGs

In a reducible CFG, each loop $X$ (= strongly-connected set of nodes) is a **natural loop**:

- It has a single entry point $T \in X$, the **loop head**.
- $T$ dominates every node in $X$.
- Every node in $X$ has a path to $T$ that stays within $X$.  

**Natural loop**

**Non-natural loop**
Control structures and reducibility

All the control structures viewed so far:

- conditionals: if-then-else, switch
- loops with test at beginning, at end, in the middle
- early exits: return, break, continue

produce reducible CFGs.
Control structures and reducibility

All the control structures viewed so far:

- conditionals: if-then-else, switch
- loops with test at beginning, at end, in the middle
- early exits: return, break, continue

produce reducible CFGs.

Conversely: every reducible graph is the CFG of a program without goto, using only conditionals, infinite loops, and multi-level break/continue exits.
Constructing the structured program

The historical algorithm by Peterson, Kasami, and Tokura (1973): three passes; outlined in a proof.


Recursion on the dominator tree for the graph.
Ramsey’s algorithm

Each loop head produces an infinite while true loop.

The children in the domination tree that are also successors produce if-then-else conditionals. (A, B, C, D)
Ramsey’s algorithm

(a) Unusual control-flow graph (node E reachable two ways)

(b) Dominator tree

(c) Imposed nesting structure

The other children (E, F) are placed in blocks, nested according to reverse postorder.

Nontrivial edges become `break N` or `continue N`.
Ramsey’s algorithm

(a) Unusual control-flow graph (node $E$ reachable two ways)

(b) Dominator tree

(c) Imposed nesting structure

Fig. 4. Imposing nesting structure for merge nodes and the translations of nodes.

Fig. 5. Schematic translation of Figure 4a

When $\text{doTree } A$ calls $\text{doTree } B$ recursively, node $B$ is also a conditional and is translated into an if—and the translation of $B$'s child $C$ can be placed into the true branch of $B$'s if form. What about $E$?

Looking at the control-flow edges leaving $B$, it appears as if the translation of $E$ should go in the false branch of $B$'s if form. But $\text{doTree}$ doesn't look at control-flow edges; it looks at the dominator tree. Since node $E$ isn't a child of node $B$, $\text{doTree } B$ isn't responsible for $E$'s translation. Instead, $\text{doTree } B$ fills the false branch with a br instruction. The br instruction exits a block···end form, so when $\text{doTree } A$ places $E$’s translation, it must ensure that $E$’s translation follows a block···end form.

Calling $\text{doTree } A$ places the translations of $E$ and $F$ as shown in Figure 5 (on the right). The translation of $E$ is preceded by the translations of nodes $A$, $B$, $C$, and $D$, which are wrapped in block···end. The translations of $C$ and $D$ include the br instructions that are needed to reach node $F$. The translation of $F$ is preceded by the translations of nodes $A$ through $E$, which are wrapped in another block···end. The blocks nest as visualized in Figure 4c, in which each block is represented by a colored region. Given this nesting structure, any preceding translation can reach $E$ or $F$ using a br instruction.

How does $\text{doTree } A$ determine that the translation of $E$ goes on the inside? Nodes $E$ and $F$ are both immediately dominated by $A$, and both are merge nodes. But these properties don't suffice to determine which translation goes inside the other. And order matters! In the control-flow graph (Figure 4a), $E$ branches to $F$, so $E$’s translation has to go on the inside. But what if neither node branches directly to the other? How does $\text{doTree}$ place their translations?

Peterson, Kasami, and Tokura [1973] place translations by using a complicated predicate involving the existence of paths in a mutated control-flow graph. But there is a simpler way to do it. Translations can be placed using a purely local test on the original, unmutated control-flow graph: compare the reverse postorder numbers of the nodes being translated.

```plaintext
block {  // blue area
    block {  // red area
        if (A) { if (B) { C; break 2; } else { break 1; } 
            else { if (D) { break 1; } else { break 2; } 
        } E;
    } F;
}
```
Summary
From 1945 to 1975, programming practices and programming languages shift

- from a “machine” view of control (jumps and labels), with the help of program flowcharts,
- to a structured view of control (conditionals, loops, etc), making the source code the main representation of the program.

Next lecture: how to structure programs at a larger scale: subroutines, procedures, functions, generators, coroutines, ...
References
A short, journalistic story of Algol:

- ALGOL 60 at 60: The greatest computer language you’ve never used and granddaddy of the programming family tree, The Register, 15/05/2020. https://www.theregister.com/2020/05/15/algol_60_at_60/

On structured programming:


On translating CFGs to structured programs:

- N. Ramsey, Beyond Relooper: Recursive Translation of Unstructured Control Flow to Structured Control Flow (Functional Pearl), PACMPL 6, ICFP, 2022.