CAMBRIDGE
Cambridge University Press & Assessment
978-1-009-76751-4 — Control Structures for Programming Languages
Xavier Leroy
Excerpt
More Information

Part I

Control Structures for Imperative Languages

© in this web service Cambridge University Press & Assessment www.cambridge.org

CAMBRIDGE
Cambridge University Press & Assessment
978-1-009-76751-4 — Control Structures for Programming Languages
Xavier Leroy
Excerpt
More Information

1 Early Programming Languages

1.1 Control flow in processors

Most computing devices today are programmable computers with a stored program, meaning that the program that describes the computation is stored in memory along with the data on which it operates. This computer organization is known as "the von Neumann architecture", because it was first described in a technical report by von Neumann (1945). A similar architecture was described by Turing (1946).

Program counter and branches. In a von Neumann architecture, the central processing unit (CPU) has a special register, the *program counter* (PC), which holds the address of the memory word containing the next instruction to be executed. Most instructions (arithmetic and logic operations, memory reads and writes, etc.) increment the PC when they are executed, making it point to the next instruction in memory. This ensures *sequential* execution of the instructions stored in memory. In contrast, *branch instructions* explicitly set the value of the PC, making it possible to jump to any point in the program.

For example, the following machine code for the x86 processor family produces the factorial numbers 1!, 2!, …, n!, … on output port 32. (The actual machine code is the hexadecimal numbers in block 2. A disassembly of the machine code is shown in block 3 to help understand it.)

```
 0:   b8 01 00 00 00          mov  eax, 1  ; eax contains n!
 5:   ba 01 00 00 00          mov  edx, 1  ; edx contains n
10:   e7 40                   out  32, eax
12:   83 c2 01                add  edx, 1
15:   0f af c2                imul eax, edx
18:   eb f6                   jmp  10
```

(1.1)

The PC successively takes the values 0, 5, 10, 12, 15 and 18 as the first five instructions are executed in sequence. The `jmp 10` instruction, encoded by the two bytes `eb f6`, sets the PC back to value 10, creating an infinite loop around the `out`, `add` and `imul` instructions.

Conditional branches. Modern processors offer *conditional branch* instructions that set the PC to a given value if some condition on the registers is true, but continue in sequence if the condition is false.

© in this web service Cambridge University Press & Assessment www.cambridge.org

CAMBRIDGE
Cambridge University Press & Assessment
978-1-009-76751-4 — Control Structures for Programming Languages
Xavier Leroy
Excerpt
More Information

For example, here is a counted loop that, given n in register `edx`, computes $n!$ and outputs it:

```
 0:   b8 01 00 00 00          mov  eax, 1                          (1.2)
 5:   0f af c2                imul eax, edx
 8:   83 ea 01                sub  edx, 1
11:   75 f8                   jnz  5          ; jump to 5 if not 0
13:   e7 40                   out  32, eax
```

The instruction written `jnz` branches if the previous result was not zero, and continues in sequence otherwise. It causes the `imul`/`sub` loop body to be executed n times, until register `edx` drops to zero. Execution then continues with the `out` instruction following the `jnz`.

Historical alternative: self-modifying code. Conditional branch instructions are not part of the original designs of von Neumann and Turing. Instead, both authors rely on the fact that the code is stored in memory, and can therefore be modified during program execution, by writing to the appropriate memory locations. This is called *self-modifying code*.

For example, here is how to jump 100 bytes forward when register `edx` is 1 and continue in sequence when register `edx` is 0, by synthesizing and then executing the appropriate `jmp +100` or `jmp +0` instruction:

```
 0:   6b c2 64                imul eax, edx, 100                   (1.3)
 3:   88 05 01 00 00 00       mov  1(rip), al
 9:   eb ??                   jmp  ??    ; dynamically modified
```

The byte following the `eb` byte is the relative address of the target of the branch. The `mov` instruction writes 0 (if `edx` is 0) or 100 (if `edx` is 1) in this byte. It then increments the PC and executes the `jmp` instruction at 9, which branches to PC 11 (if `edx` is 0) or 111 (if `edx` is 1).

Self-modifying code is difficult to set up and degrades the performance of pipelined processors. This is why, during the 1950s, conditional branch instructions and indexed memory addressing became standard in processor instruction sets, replacing the two most common uses of self-modifying code.

1.2 Machine languages, assembly languages

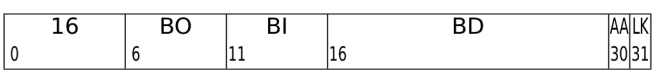
Machine language. Machine language is the programming language that can be executed directly in hardware by the CPU circuits. A machine language program consists of the bit-level encodings of the program's instructions as they are stored in memory during execution. Each instruction set architecture (ISA) defines its own machine language. Many ISAs encode each instruction as a 32-bit word; some use variable-length encodings. The encodings are documented in the ISA reference manual. For example, Figure 1.1 shows the encoding of the conditional branch instruction in the Power ISA. In the 32-bit word that encodes this instruction, some bits are fixed and identify the instruction. Other bits encode the condition

© in this web service Cambridge University Press & Assessment www.cambridge.org

CAMBRIDGE
Cambridge University Press & Assessment
978-1-009-76751-4 — Control Structures for Programming Languages
Xavier Leroy
Excerpt
More Information

Branch Conditional B-form

bc	BO,BI,target_addr	(AA=0 LK=0)
bca	BO,BI,target_addr	(AA=1 LK=0)
bcl	BO,BI,target_addr	(AA=0 LK=1)
bcla	BO,BI,target_addr	(AA=1 LK=1)




```
if (64-bit mode)
   then M ← 0
   else M ← 32
if ¬BO2 then CTR ← CTR - 1
ctr_ok ← BO2 | ((CTRM:63 ≠ 0) ⊕ BO3)
cond_ok ← BO0 | (CRBI+32 ≡ BO1)
if ctr_ok & cond_ok then
   if AA then NIA ←iea EXTS(BD || 0b00)
   else       NIA ←iea CIA + EXTS(BD || 0b00)
if LK then LR ←iea CIA + 4
```

Figure 1.1 The "branch conditional" instruction of the Power instruction set architecture. (Source: OpenPOWER Foundation, *Power Instruction Set Architecture*, version 3.1C, 2024.)

for branching, the target of the branch, and possible side effects on the `LR` and `CTR` registers.

Assembly language. Assembly language is a textual representation of machine language, much easier for humans to write and read, but close enough to machine language that a simple program called an *assembler* can automatically translate from assembly language into executable machine code. The first assembler appeared in 1947 in Kathleen Booth's work on the ARC computer.

Assembly language uses *mnemonics* to name processor instructions (like the `mov`, `imul`, `jmp` mnemonics used in the examples above), and *labels* to name program points and branch targets. It also supports *comments* for documenting the code. Comments are ignored by the assembler, but greatly help humans to understand and maintain the code.

Here is the computation of $n!$ from (1.1), rewritten in x86 assembly language. Comments start with a semicolon character ";"

```
; Compute factorial of n.                                    (1.4)
; n is passed in edx.
; n! is left in eax.
; edx is clobbered.
        mov  eax, 1           ; initialize eax to 1
again:  imul eax, edx         ; multiply eax by n
        dec  edx              ; decrement n
        jnz  again            ; repeat while n is not 0
                              ; here, eax contains n!
```

© in this web service Cambridge University Press & Assessment www.cambridge.org

CAMBRIDGE
Cambridge University Press & Assessment
978-1-009-76751-4 — Control Structures for Programming Languages
Xavier Leroy
Excerpt
More Information

Note the use of a symbolic label, named `again`, to denote the location of the `imul` instruction, making it easy to express jumps to that location, as in the conditional branch `jnz again`.

While much easier to work with than machine language, assembly language remains in one-to-one correspondence with the instruction set architecture: each opcode corresponds to one of the instructions supported by the processor; each line of source text produces one binary-encoded machine instruction.

Macro-assemblers. Macro-assemblers, or autocoders as IBM called them, support additional opcodes beyond those corresponding to the processor instructions, thus providing programmers with a richer assembly language. These extended opcodes, also called macro-instructions, are expanded into canned sequences of processor instructions during translation to machine code.

Macro-instructions are often used to write various types of loops concisely and clearly. For example, a counted loop, with register `ecx` ranging from 0 to 99, can be expressed by two macro-instructions, `begin_loop` and `end_loop`, expanded as shown on the right:

```
; Source code                    ; Code after macro-expansion
                                 mov ecx, 0
begin_loop lbl, ecx, 0     lbl:
...                              ...
...                              ...
end_loop lbl, ecx, 100           inc ecx
                                 cmp ecx, 100
                                 jne lbl
```

(1.5)

`begin_loop` initializes the loop index register and associates a label with the first instruction of the loop body. `end_loop` increments the loop index and re-executes the loop body if it has not reached the final value of the index.

Flowcharts. An important tool for programming in assembly language is the program flowchart: a graphical representation of a program or program fragment, using boxes to represent basic computational steps and arrows to represent the flow of control. For example, Figure 1.2 shows a flowchart for a counted loop that prints five "star" characters.

In the period from 1945 to 1985, from the first modern computers to the widespread adoption of structured programming, flowcharts were the preferred way to design and document programs. Flowcharts omit a number of details such as the exact sequences of instructions that implement the boxes, the order in which they occur in the code, and the labels and branches used to implement the flow of control. This is a strength of flowcharts: they present a higher-level, less cluttered view of the program than the assembly code. This is also a weakness: substantial manual work is required to translate a flowchart into assembly code, and it is difficult to keep the flowchart and the code in sync as the program changes.

© in this web service Cambridge University Press & Assessment www.cambridge.org

CAMBRIDGE
Cambridge University Press & Assessment
978-1-009-76751-4 — Control Structures for Programming Languages
Xavier Leroy
Excerpt
More Information

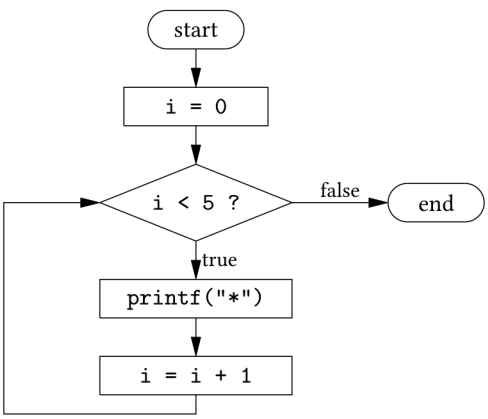


Figure 1.2 A flowchart for a counted loop that prints five "star" characters

1.3 FORTRAN

Expressions. Fortran I, introduced by IBM in 1957, was the first programming language to support general expressions written in algebraic notation (hence its name, which stands for "FORmula TRANslator"). For example, the solutions `X1` and `X2` to the quadratic equation $\mathtt{A}x^2 + \mathtt{B}x + \mathtt{C} = 0$ are concisely expressed in Fortran:

```
D = SQRT(B*B - 4*A*C)                                    (1.6)
X1 = (-B + D) / (2*A)
X2 = (-B - D) / (2*A)
```

It is the job of the Fortran compiler to implement these expressions using elementary processor instructions, using registers or temporary variables to hold intermediate results. For reference, here is the AArch64 assembly code for the calculation above:

```
ldr d0, B              fsub d0, d0, d1        fadd d3, d1, d0    (1.7)
fmul d0, d0, d0        fsqrt d0, d0           fdiv d3, d3, d2
ldr d1, A              ldr d1, B              str d3, X1
fmov d2, #4.0          fneg d1, d1            fsub d3, d1, d0
fmul d1, d1, d2        ldr d2, A              fdiv d3, d3, d2
ldr d2, C              fmov d3, #2.0          str d3, X2
fmul d1, d1, d2        fmul d2, d2, d3
```

DO loops. Fortran I is also notable for the introduction of the first control structure in a programming language: the `DO` counted loop.

```
      DO 100 I = 1, N                                    (1.8)
      ...
      ...
100   ...
```

This code repeatedly executes the lines between the `DO` command and the label 100 (excluded), with variable `I` taking successive values of 1, 2, …, `N`. The loop body is not supposed to change `I`, although some compilers tolerate it.

© in this web service Cambridge University Press & Assessment www.cambridge.org

CAMBRIDGE
Cambridge University Press & Assessment
978-1-009-76751-4 — Control Structures for Programming Languages
Xavier Leroy
Excerpt
More Information

In addition to providing a concise notation for a common programming idiom, the DO notation can be compiled into efficient machine code. The original Fortran I compiler placed the loop test at the end of the loop body and performed limited forms of loop-invariant code motion and addressing strength reduction. Other optimizations that apply well to DO loops include loop unrolling and vectorization.

Control in Fortran. Besides DO loops, Fortran I supports GOTO commands that jump to a label unconditionally. For example, an infinite loop is written as follows:

```
100  ...                                                    (1.9)
     ...
     GOTO 100
```

The GOTO command also supports computed jumps, which are jumps to a label whose value is contained in a variable, as well as the ASSIGN command, which sets a variable to the value of a label. For example,

```
     ASSIGN 100 TO DEST                                     (1.10)
     ...
     GOTO DEST
```

Fortran I supports conditional jumps via an unusual three-way arithmetic branch IF (X) L1, L2, L3, which branches to L1 if X < 0, to L2 if X = 0, and to L3 if X > 0. For example, the following code fragment sets Y to the absolute value of X:

```
     Y = X                                                  (1.11)
     IF (X) 701, 702, 702
701  Y = -Y
702  ...
```

The familiar IF command with a Boolean condition was introduced in Fortran IV (1961), along with Boolean operators and Boolean-valued comparisons .EQ. (equal), .LT. (less than), etc. For example, the computation of the absolute value of X can be written as follows:

```
     Y = X                                                  (1.12)
     IF (X .LT. 0)  Y = -Y
```

IF can only be followed by a single command (except IF and DO), and there is no ELSE clause. More complex conditional commands must be expressed using IF...GOTO and labels:

```
     IF (X .LT. 0)  GOTO 100                                (1.13)
     Y = X
     GOTO 101
100  Y = -X
101
```

Fortran 77 adds block-structured conditional statements IF...ELSE...END IF, so that the previous example can be written more clearly as

```
IF (X .LT. 0) THEN                                          (1.14)
    Y = -X
ELSE
    Y = X
END IF
```

© in this web service Cambridge University Press & Assessment www.cambridge.org

CAMBRIDGE
Cambridge University Press & Assessment
978-1-009-76751-4 — Control Structures for Programming Languages
Xavier Leroy
Excerpt
More Information

Fortran 99 adds block-structured counted loops `DO...END DO` and general loops `DO WHILE...END DO`.

1.4 ALGOL 60

Structured statements. In Fortran and most other programming languages, expressions such as `(1 + X) * Y` are formed by combining constants (here, `1`) and variables (`X`, `Y`) using operators (`+`, `*`). The Algol programming language, introduced in 1960, extends this approach to commands, also called "statements" and written s below. Statements are built from basic commands such as

- assignments: `x := expr`
- procedure calls: `proc(arg1, arg2)`

These basic commands can, then, be combined using *control structures* such as

- sequencing (block): `begin` s_1`;` s_2`; ... end`
- conditional: `if` be `then` s_1 `else` s_2
- counted loop: `for i :=` e_1 `step` e_2 `until` e_3 `do` s
- general loop: `while` be `do` s

Here, e stands for an arithmetic expression and be for a Boolean expression. Blocks `begin ... end` can also declare local variables and local labels. For example, the following block statement sets `m` to the largest value in the array `a[1:n]` and `p` to the position of that value in `a`.

```
begin                                                    (1.15)
  integer i;
  m := min_int; p := 0;
  for i := 1 step 1 until n do
    if a[i] >= m then begin p := i; m := a[i] end
end
```

A fresh perspective on programming languages. Unlike assembly language instructions or Fortran commands, Algol statements are recursive in nature and can be nested to any depth. This has led to new ways of describing programming languages that accommodate this recursive nature and emphasize compositionality.

At the syntactic level, Algol was the first programming language to be defined using a formal grammar, written in Backus–Naur form. Here is the grammar for a simplified Algol-like language:

Statements:

$$
\begin{aligned}
s \quad ::= \; & x := e \\
\mid \; & proc(e_1, \ldots, e_n) \\
\mid \; & \texttt{begin } s_1; \ldots; s_n \texttt{ end} \\
\mid \; & \texttt{if } be \texttt{ then } s_1 \texttt{ [else } s_2\texttt{]} \\
\mid \; & \texttt{while } be \texttt{ do } s
\end{aligned}
$$

© in this web service Cambridge University Press & Assessment www.cambridge.org

CAMBRIDGE
Cambridge University Press & Assessment
978-1-009-76751-4 — Control Structures for Programming Languages
Xavier Leroy
Excerpt
More Information

$$\mid \texttt{for}\ x := e_1\ \texttt{step}\ e_2\ \texttt{until}\ e_3\ \texttt{do}\ s$$

Arithmetic expressions:

$$e \ ::= \ c \mid x \mid e_1 + e_2 \mid \ldots$$

Boolean expressions:

$$be \ ::= \ \texttt{true} \mid \texttt{false} \mid e_1 < e_2 \mid \ldots$$

Consequently, Algol compilers were the first to use recursive or stack-based parsing algorithms.

At the semantic level, Algol popularized the idea that the meaning of a command is determined as a combination of the meanings of its subcommands. This idea, informal at first, was later made mathematically precise in the work on denotational semantics by Strachey, Scott, and others (see Section 6.2).

Unstructured control. In addition to control structures, Algol also supports `goto` jumps to labels L that are attached to statements:

Statements: $s ::= \ldots$

$\mid L : s$	statement labeled L
$\mid \texttt{goto}\ L \mid \texttt{go to}\ L \mid \texttt{go}\ L$	jump to label L

The `goto` spelling is used in the 1963 revised report on Algol, while the 1976 report uses `go to`, and many compilers accept `go` as well.

Each label has a *scope*, which is the nearest block enclosing the definition of the label. A `goto` statement can only jump to a label that is currently in scope, *i.e.* a label defined in the same block as the `goto` statement or in an enclosing block. Consequently, `goto` cannot jump into a block from outside this block: see Figure 1.3.

Jump within the block ✔

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

Jump out of the block ✔

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

Jump into a block ✘

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

Figure 1.3 Three kinds of `goto` jumps. Jumping within the same block or out of the current block is allowed in Algol, but jumping into a sub-block is not.

A notation to describe algorithms. Combining control structures and `goto` jumps in the style of Algol quickly became the preferred way to describe algorithms in textbooks and journals. As early as 1960, the *Communications of the ACM*, a journal that played an important role in the early days of computer science, mandated the use of Algol instead of flowcharts when publishing new algorithms.

As an example of this Algol style, here is binary search in a sorted array `a[1:n]`.

```
begin                                                    (1.16)
    integer low, high, mid;
```

© in this web service Cambridge University Press & Assessment www.cambridge.org

CAMBRIDGE
Cambridge University Press & Assessment
978-1-009-76751-4 — Control Structures for Programming Languages
Xavier Leroy
Excerpt
More Information


```
    low := 1; high := n;
    while low <= high do begin
        mid := low + (high - low) / 2;
        if x < a[mid] then high := mid - 1
        else if x > a[mid] then low := mid + 1
        else goto FOUND
    end;
  NOTFOUND:
    ...
  FOUND:
    ...
end
```

This combination of `goto` with control structures can be found in many imperative and object-oriented languages: Algol 68, Pascal, Ada, Simula, PL/I, C, C++, C#, Perl, Go, … Imperative languages with only structured control and no `goto` statements are relatively few: Modula-2 (1980), Eiffel (1986), Modula-3 (1988), Python (1991), Java (1995), Rust (2012), Swift (2014).

1.5 Conditionals and loops

The control structures introduced by Algol 60 (blocks, conditionals, counted loops, general loops) are found in most, if not all, of the imperative languages that followed. Many variants and extensions of these control structures have been proposed and incorporated into some languages.

Flavors of conditionals. Many languages provide an `elif` or `elseif` construct to facilitate the writing of nested `if…then…else` conditionals:

$$\texttt{if } be_1 \texttt{ then } s_1 \texttt{ elif } be_2 \texttt{ then } s_2 \texttt{ elif } \ldots \texttt{ else } s_n$$

An alternative syntax for cascading Boolean tests is the `cond` construct of Lisp:

$$\texttt{(cond } (be_1\ s_1)\ (be_2\ s_2)\ \ldots\ \texttt{(t } s_n\texttt{))}$$

Case analysis on a value of integer type or enumerated type is provided by the `case` construct of Pascal (left), the `switch` construct of C and C-like languages (center), and the `match` construct of Python (right):

```
case grade of
  'A' : s1
  'B', 'C': s2
  'D' : s3
  'F' : s4
end
```

```
switch (grade) {
  case 'A': s1; break;
  case 'B':
  case 'C': s2; break;
  case 'D': s3; break;
  case 'F': s4; break;
}
```

```
match grade:
  case 'A' : s1
  case 'B' | 'C': s2
  case 'D' : s3
  case 'F' : s4
```

Pattern matching, as found in most functional languages (Haskell, OCaml, SML, …) and some imperative languages (Rust, Swift, C#, …) extends case analysis with the ability to destructure data types and bind values to variables.

© in this web service Cambridge University Press & Assessment www.cambridge.org