Actions: the basic units of imperative programming.

The assignment statement specifies a typical action (and is a distinguishing feature of imperative programming as opposed to functional programming).

For example,
\[ x := 2 + 3 \]

is an assignment specifying the action of computing the value 5 of the expression \( 2 + 3 \) and assigning it to the variable \( x \); the old value of \( x \) is forgotten.

Control Flow: the order in which actions are performed.

The control flow of a program is specified by (control) statements.
A (sequential) computation consists of a sequence of actions. A program is a succinct representation of the computation that occurs when the program runs.

<table>
<thead>
<tr>
<th>Computation</th>
<th>Program</th>
</tr>
</thead>
<tbody>
<tr>
<td>writeln(1, 1*1)</td>
<td>for i:=1 to 3 do</td>
</tr>
<tr>
<td></td>
<td>writeln(i, i*i)</td>
</tr>
<tr>
<td>writeln(1, 2*2)</td>
<td></td>
</tr>
<tr>
<td>writeln(1, 3*3)</td>
<td></td>
</tr>
</tbody>
</table>

The static text of a program is distinct from the dynamic computations that occur when the program runs.

Structured Programming

The structure of the program text should help us understand what the program does. Specifically, a program is structured if the flow of control through the program is evident from the syntactic structure of the program text.
THE GOTO STATEMENT

The goto statement explicitly transfers the flow of control to a labeled statement from elsewhere in a program.

The following shows a FORTRAN program that calculates $1 + 2 + \cdots + N$

```fortran
NUMBER = 0
SUM = 0
10 IF ( NUMBER .LE. N ) THEN
    NUMBER = NUMBER + 1
    SUM = SUM + NUMBER
    GOTO 10
END IF
```

A TYPICAL FORTRAN PROGRAM

```fortran
C A HYPOTHEICAL PROGRAM IN FORTRAN
FUNF(T) = SQRTF(ABSF(T)) + 5.0*T**3
DIMENSION A(11)
1  FORMAT(6F12.4)
READ 1,A
DO 10 J =1,11
  I = 11 - J
  Y = FUNF(A(I+1))
  IF(400.0-Y)4,8,8
4  PRINT 5,I
   5  FORMAT(I10,10H TOO LARGE)
   GOTO 10
8  PRINT 9,I,Y
9  FORMAT(I10,F12.7)
10 CONTINUE
STOP
```
A TYPICAL ALGOL60 PROGRAM

begin integer i, real y, real array a[0:10]
  real procedure f(t); real t; value t;
  f := sqrt(abs(t)) + 5 x t ↑ 3;
  for i := 0 step 1 until 10 do read(a[i]);
  for i := 10 step -1 until 0 do
    begin y := f(a[i]);
       if y > 400 then write(i, 'too large')
       else write(i, y)
    end
end

WHY STRUCTURED PROGRAMMING?

Eventually, one of our aims is to make such well-structured programs that the intellectual effort (...) needed to understand them is proportional to program length (...).
- Dijkstra

Two characteristics of imperative programming:
– programs and computations are not the same thing. Programs are what we write, while computations are the actions that occur when a program runs.
– the values of variables may change as a program runs

Two desirable concepts:
– Structured Statements
– Invariants
**Structured vs. Unstructured Statements**

<table>
<thead>
<tr>
<th>Using goto statements</th>
<th>Using structured statements</th>
</tr>
</thead>
<tbody>
<tr>
<td>read(x);</td>
<td>read(x);</td>
</tr>
<tr>
<td>2: if x=0 then goto 8;</td>
<td>while x ≠ 0 do begin</td>
</tr>
<tr>
<td>writeln(x);</td>
<td>writeln(x);</td>
</tr>
<tr>
<td>4: read(next)</td>
<td>repeat</td>
</tr>
<tr>
<td>if next=x then goto 4;</td>
<td>read(next);</td>
</tr>
<tr>
<td>x:= next;</td>
<td>until next ≠ x;</td>
</tr>
<tr>
<td>goto 2;</td>
<td>x := next;</td>
</tr>
<tr>
<td>8: ;</td>
<td>end;</td>
</tr>
</tbody>
</table>

**Structured vs. Unstructured Statements (cont.)**

```plaintext
if A > B then goto great
else if A < B then goto less;
else
    begin //great
    ...
    end
else if A < B then
    begin //less
    ...
    end
else
    begin //equal
    ...
    end
```
SYNTAX-DIRECTED CONTROL FLOW

Sequencing: Sequential Composition of Statements. Control flows sequentially through a sequence of statements like

\[ \text{temp} := x; \ x := y; \ y := \text{temp} \]

\[
\begin{array}{c}
S_1 \\
S_2 \\
S_k
\end{array}
\]

Selection: Conditional (and Case) Statements

\[
\text{if} \ (\text{expression}) \ \text{then} \ (\text{statement}_1) \ \text{else} \ (\text{statement}_2)
\]

A variant: \[ \text{if} \ (\text{expression}) \ \text{then} \ (\text{statement}) \]
**SYNTAX-DIRECTED CONTROL FLOW**

*Looping: While, Repeat, and For Statements*

- `while` (expression) `do` (statement)
- `repeat` (statement) `until` (expression)
- `for` (name) := (expr) `to` (expr) `do` (statement)
- `for` (name) := (expr) `downto` (expr) `do` (statement)

All the above control flows have the following property: single-entry/single-exit
SYNTAX OF STATEMENTS IN PASCAL

\[
\text{〈statement〉} ::= \text{〈expr〉} := \text{〈expr〉} \\
| \text{〈name〉 (〈expr_list〉 );} \\
| \text{begin 〈statement_list〉 end} \\
| \text{if (〈expr〉) then 〈statement〉} \\
| \text{if (〈expr〉) 〈statement〉 else 〈statement〉} \\
| \text{while (〈expr〉) do 〈statement〉} \\
| \text{repeat 〈statement〉 until (〈expr〉 );} \\
| \text{for 〈name〉 := 〈expr〉 to 〈expr〉 do 〈statement〉} \\
| \text{for 〈name〉 := 〈expr〉 downto 〈expr〉 do 〈statement〉} \\
| \text{case 〈expr〉 of 〈case〉}
\]

\[
\text{〈statement_list〉} ::= \text{〈empty〉} \\
| \text{〈statement〉 ; 〈statement_list〉}
\]

\[
\text{〈case〉} ::= \text{〈constant〉 : 〈statement〉} \\
| \text{〈constant〉 : 〈statement〉 ; 〈case〉}
\]

A STYLE OF NESTING CONDITIONALS

When conditionals are nested, the following style guideline improves readability.

\[
\text{if ... then ...} \\
\text{else if ... then ...} \\
\text{else if ... then ...} \\
\text{else ...}
\]

Example:

\[
\text{if (year \ mod \ 400) = 0 then leap := true} \\
\text{else if (year mod 100) = 0 then leap := false} \\
\text{else if (year mod 4) = 0 then leap := true} \\
\text{else leap := false}
\]
DEFINITE VS. INDEFINITE ITERATIONS

Looping constructs can be divided roughly into two groups:

– **Definite Iteration.** A definite iteration (definite loop) is executed a predetermined number of times.
  Constructs: `for` statements (in most languages).

– **Indefinite Iteration.** The number of executions of an indefinite iteration (indefinite loop) is not known when control reaches the loop; the number is determined by the course of the computation.
  Constructs: `while` and `repeat` statements; `for` statements in C.

DESIGN ISSUES OF `for` STATEMENTS

The design of `for` statements in a language depends on the treatment of the index variable, the step, and the limit.

– Are the step and the limit computed once or are they recomputed each time control flows through the loop?
– Is the limit tested at the beginning or at the end of each pass through the loop?
– Can the value of the index variable be changed within the loop?
– Is the index variable defined upon loop exit?
A Pascal Program to Remove Adjacent Duplicates

```pascal
Program uniq (input, output);
var x, next : integer
begin
  read(x);
  while x <> 0 do begin
    writeln(x);
    repeat read(next);
      until next <> x;
    x := next;
  end;
end.
```

CASE STATEMENTS

A case statement uses the value of an expression to select one of several substatements for execution.
```
case (expression) of
  (constant1) : (statement1);
  (constant2) : (statement2);
  ...
  (constantn) : (statemenentn)
end
```

Most languages agree on the following points:
- Case constants can appear in any order.
- Case constants need not be consecutive.
- Several case constants can select the same substatement.
- Case constants must be distinct.

Further issues: Can there be a default case? Are ranges of case constants allowed?
### An implementation of while statement

```latex
while E do S
```

### IMPLEMENTATION OF CASE STATEMENTS

The implementation of case statements can affect their usage.

The code generated by good compilers depends on the distribution of case constants:

- A small number of cases is implemented using conditionals.
- For a larger number of cases, the compiler uses a "jump table" if, say, at least half the entries will be used.
- If the number of cases is large enough and too many entries in a jump table would remain unused, the compiler uses a hash table.
WHICH IS BETTER?

case E of
  1     : S1;
  11    : S2;
  121   : S3
end

n := E;
if n = 1 then S1
  else if n = 11 then S2
  else if n = 121 then S3

SEQUENCES: SEPARATORS VS. TERMINATORS

Sequences of statements, declarations, or parameters can be classified by asking the following questions:
– Can the sequence be empty?
– If there is a delimiter, does it separate elements or terminate them?

A delimiter separates elements if it appears between them; it terminates elements if it appears after each element.
Fewer programming errors are believed to occur if semicolons terminate statements than if they separate statements.
SEMICOLONS AS SEPARATORS

Pascal uses semicolons primarily to separate statements, as in
\[ \text{begin stmt}_1 \;;\; \text{stmt}_2 \;;\; \text{stmt}_3 \;\text{end} \]

Inserting an empty statement between \text{stmt}_3 and \text{end} makes semicolons look like terminators:
\[ \text{begin stmt}_1 \;;\; \text{stmt}_2 \;;\; \text{stmt}_3 \;;\; \text{end} \]

But empty statements make the placement of semicolons significant; insertion of a semicolon can change the meaning of a program in Pascal.
\[ \text{if expr then} \;;\; \text{stmt} \]
is not the same as
\[ \text{if expr then stmt} \]

Modula-2 avoids the problem by attaching a closing keyword \text{end}:
\[ \text{if expr then stmt end} \]

---

A Grammar For A Fragment of Pascal

\[ S ::= \text{\{empty\}} \]
\[ \quad \text{stmt} \]
\[ \quad \text{begin SL end} \]
\[ \quad \text{if expr then} \; S \]
\[ \quad \text{if expr then} \; S \; \text{else} \; S \]
\[ \quad \text{while expr do} \; S \]

\[ SL ::= SL; \; S \]
\[ \quad S \]
A parse tree for `begin stmt ; stmt ; stmt end`

```
begin stmt ; stmt ; stmt end
begin stmt ; stmt ; stmt end

S
 SL  ;  S
 SL  ;  S
 SL  ;  S
 S
 stmt₁
 stmt₂
 stmt₃
```

A grammar for a fragment of Modula-2

```
S ::= \{empty\}
stmt
   if expr then SL end
   if expr then SL else SL end
   while expr do SL end

SL ::= SL; S
   S
```
AVOIDING DANGLING ELSES

Modula-2 avoids the dangling-else ambiguity because conditionals have a closing keyword end.

But, closing delimiters can lead to a proliferation of keywords.

```
if expr_1 then stmt_1
else if expr_2 then stmt_2
  else if expr_3 then stmt_3
  else stmt_4
end
end
```

Optional elsif parts solve the problem.

```
if expr_1 then stmt_1
elsif expr_2 then stmt_2
elsif expr_3 then stmt_3
else stmt_4
end
```

BREAK AND CONTINUE STATEMENTS

Break and continue statements facilitate the handling of special cases in loops:

- A `break` statement sends control out of the enclosing loop to the statement following the loop. It can be used to jump out of a loop after establishing the conditions upon exit from the loop.
- A `continue` statement repeats the enclosing loop by sending control to the beginning of the loop. It can be used to restart the loop after establishing the loop invariant, the condition that holds upon loop entry.
BREAK AND CONTINUE STATEMENTS (cont.)

One use of break statements is to break out of a loop after handling a special case:

```plaintext
while condition do
    if special case then
        take care of the special case;
        break;
    end if;
    handle the normal cases;
end while
```

A corresponding fragment for continue statements:

```plaintext
while condition do
    if normal case then
        handle the normal case;
        continue;
    end if;
    take care of the special cases;
end while
```

RETURN STATEMENTS

Execution of a statement

```plaintext
return (expression)
```

sends control back from a procedure to a caller, carrying the value of (expression). If the return statement is not in a procedure, then the program halts.

Both return and break statements send control out of an enclosing construct; a return out of an enclosing procedure and a break out of an enclosing loop.
GOTO STATEMENTS

A statement goto $L$ interrupts the normal flow of control from one statement to the next in sequence; control flows instead to the statement labeled $L$ somewhere in the program:

$L: \langle \text{statement} \rangle$

By itself, goto $L$ gives no indication of where label $L$ is to be found. Similarly, $L: \langle \text{statement} \rangle$ does not indicate from where control might come to it.

goto statements can be misused to write unreadable programs.

INVARIANTS

An invariant at some point in a program is an assertion that holds whenever control reaches that point.

Consider the problem of removing adjacent duplicates from a list of integers. For example, given the input 1 1 2 2 2 3 1 4 4, the output should be 1 2 3 1 4.

View 1 1 2 2 2 3 1 4 4 as a sequence of runs 1, 2 2 2, 3, 1. Design the program around invariants:

```
read(x);
while x is not the end marker do begin
  (here, x is the first element of a run)
  writeln(x);
  repeat read(next) until next≠x;
  (here, we have read one element too many)
  x := next;
end;
```
PRE AND POST-CONDITIONS

With single-entry/single-exit constructs, the behavior of a statement can be characterized purely by conditions at the entry and exit to the statement.

A *precondition* is attached just before and a *postcondition* is attached just after a statement; both are assertions. In particular,

– a precondition just before a loop can capture the conditions for executing the loop,
– an assertion just within a loop body (i.e., before the first statement of the loop body) can capture the conditions for staying in the loop, and
– a postcondition just after a loop can capture the conditions upon leaving the loop.

PRE AND POST-CONDITIONS (CONT.)

{ \( x \geq 0 \) and \( y > 0 \) }

*while* \( x \geq y \) *do begin*

{ \( x > 0 \) and \( x \geq y \) }

\( x := x - y \)

{ \( y > 0 \) and \( x \geq 0 \) }

*end*

{ \( y > 0 \) and \( x < y \) }
AN ANNOTATED PROGRAM

{ } ≥ 0 ∧ y ≥ 0 ∧ gcd (x,y) = gcd (m,n) 
while x ≠ 0 and y ≠ 0 do
begin
{ } ≥ 0 ∧ y ≥ 0 ∧ gcd (x,y) = gcd (m,n) 
if x < y
then swap (x,y);
{ } ≥ y ∧ y ≥ 0 ∧ gcd (x,y) = gcd (m,n) 
x := x − y;
{ } ≥ 0 ∧ y ≥ 0 ∧ gcd (x,y) = gcd (m,n) 
end;
{ } = 0 ∧ y ≥ 0 ∧ y = gcd (x,y) = gcd (m,n) ∨
( } ≥ 0 ∧ y = 0 ∧ x = gcd (x,y) = gcd (m,n) )

Note: m and n are two arbitrary non-negative integers, at least one of which is nonzero

EXAMPLE: LINEAR SEARCH

A table supports two operations, insert(x) and find(x). Elements are inserted from left to right, starting at position 1.

The table will be maintained so that the elements of the table are in the subarray A[1..n], for 0 ≤ n, and 0 ≤ n ≤ limit.

Operation find(x) returns 0 if x is not in the table; otherwise, it returns the position in the table at which x was inserted most recently.
ALGORITHM

start with the last inserted element
while elements remain to be examined do begin
if this element is \( x \) then
   return its position
else
   consider the element to its left
end
not found, so return 0

DEVELOPMENT OF A SEARCH PROGRAM

Initial Code Sketch
initialization;
do the search;
{} (x is not in the table) or
{} (the most recent x is \( A[i] \) and \( 0 < i \leq n \) )
if \( x \) is not in the table then
   return 0;
else
   return \( i \);

Simplified Computation of the Result
initialization;
do the search;
{} \( x = A[i] \) and \( x \) is not in \( A[i +1..n] \) and \( 0 \leq i \leq n \)
return \( i \);
DEVELOPMENT OF A SEARCH PROGRAM (cont.)

Making the Sentinel Explicit

\[
\begin{align*}
A[0] & := x; \\
& \text{further initialization} \\
& \{ x = A[0] \text{ and } x \text{ is not in } A[i+1..n] \text{ and } 0 \leq i \leq n \} \\
& \textbf{while} \text{ not yet time to stop and } x \text{ not found at } i \textbf{ do} \\
& \quad i := i - 1; \\
& \{ x = A[i] \text{ and } x \text{ is not in } A[i+1..n] \text{ and } 0 \leq i \leq n \} \\
& \textbf{return} \ i;
\end{align*}
\]

DEVELOPMENT OF A SEARCH PROGRAM (cont.)

Final Developed Program Fragment ...

\[
\begin{align*}
A[0] & := x; \\
& i := n; \\
& \textbf{while} \ x \neq A[i] \textbf{ do} \\
& \quad i := i - 1; \\
& \textbf{return} \ i;
\end{align*}
\]
**PROOF RULES**

\[
\begin{align*}
\{ Q( E/x) \} & \quad x := E \{ Q \} & \text{(Assignment Axiom)} \\
\{ P \} S & \{ Q \} . \{ Q \} S & \{ R \} & \text{(Composition Rule)} \\
\{ P \} S & \{ Q \} & \{ P \} S & \{ R \} & \text{(Conditional Rule)} \\
\{ P \wedge Q \} S & \{ Q \} . \{ P \wedge \neg Q \} S & \{ Q \} & \text{(while Rule)} \\
\{ P \wedge Q \} S & \{ P \} & \text{(Rule of Consequence)}
\end{align*}
\]

**ASSIGNMENTS IN C**

The assignment operator in C is `=.`

C allows assignments to appear within expressions.

An expression `E_1 = E_2` is evaluated by placing the value of `E_2` into the location of `E_1`. The value of `E_1 = E_2` is the value assigned to the left side.

\[
\text{while } ( (c = \text{getchar()} ) \neq \text{EOF} )
\]

\[
\text{putchar(c);}
\]

is semantically equivalent to

\[
\text{while } ( 1 )
\]

\[
\text{c = getchar();}
\]

\[
\text{if ( c == EOF ) break;}
\]

\[
\text{putchar(c);}
\]
CONTROL FLOW IN C

The correspondence between the for statements holds only if S in the C statement does not change the values of i and n.

<table>
<thead>
<tr>
<th>C</th>
<th>Pascal</th>
</tr>
</thead>
<tbody>
<tr>
<td>if ( E ) S</td>
<td>if E then S</td>
</tr>
<tr>
<td>if ( E ) S1 else S2</td>
<td>if E then S1 else S2</td>
</tr>
<tr>
<td>while ( E ) S</td>
<td>while E do S</td>
</tr>
<tr>
<td>do S while ( E ) ;</td>
<td>repeat S until (not E)</td>
</tr>
<tr>
<td>for (i=1;i&lt;=n;i++) S</td>
<td>for i:=1 to n do S</td>
</tr>
<tr>
<td>for (i=n;i&gt;=1;i--) S</td>
<td>for i:=n downto 1 do S</td>
</tr>
</tbody>
</table>

FOR LOOPS IN C

The for statement has the form

\[
\text{for( } E_1; \ E_2; \ E_3 \ \text{) } S
\]

E1 is evaluated just before loop entry, E2 is the condition for staying within the loop, and E3 is evaluated just before every next iteration of the loop.

\[
\text{while ( } x \neq A[i] \ \text{) }\]
\[
--i;
\]

can be rewritten as

\[
\text{for ( } ; \ x \neq A[i]; --i);\]

A missing \(E_2\) is taken to be true; \(\text{for( };\) \(\text{;})\) can thus be read as "forever" because it sets up an infinite loop.
FOR LOOPS IN C (cont.)

\[
S = 0; \\
\text{for}(i = 1; i = n; i++) \quad \text{pitfall!} \\
S = S + i;
\]

void reverse(char S[])
{
    int c, i, j;
    for (i = 0, j = strlen(S); i < j; i++, j--) {
        c = S[i]; S[i] = S[j]; S[j] = c;
    }
}

for(i = 0; j = strlen(S) – 1; i < j; i++, j--)
    c = S[i], S[i] = S[j], S[j] = c;

SYNTAX OF STATEMENTS IN C

\[
S ::= ; \\
| E; \\
| \{ Slist \} \\
| \text{if} ( E ) \ S \\
| \text{if} ( E ) \ S \text{ else } S \\
| \text{while} ( E ) \ S \\
| \text{do} S \text{ while} ( E ); \\
| \text{for} ( Eopt ; Eopt ; Eopt ; ) \ S \\
| \text{switch} ( E ) \ S \\
| \text{case} \ Constant : S \\
| \text{default} : S \\
| \text{break} ; \\
| \text{continue} ; \\
| \text{return} ; \\
| \text{return} E ; \\
| \text{goto} L ; \\
| L : S
\]

Slist ::= <empty> \\
| Slist S

E ::= <empty> \\
| E

Spring, 2001 Statements and Structures 47
BREAK and CONTINUE

```c
for( ; ; c=getchar()) {
    if (c==‘ ‘ || ‘\t’)
        continue;
    if (c !=‘\n’)
        break;
    ++lineno;
}
```