Imperative Programming: Structured Programs

(Based on [Sethi 1996])

Yih-Kuen Tsay

Department of Information Management
National Taiwan University

Imperative Programming

- **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.

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.

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

Computations vs. Programs

- A (sequential) *computation* consists of a sequence of actions.
- A *program* is a succinct representation of the computation that occurs when the program runs.

Computation	Program

writeln(1, 1*1)
writeln(2, 2*2)
writeln(3, 3*3)
	for i:=1 **to** 3 **do** writeln(i, i*i)

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

Structured Statements

With goto's	Structured
read(x);	read(x);
2: **if** $x=0$ **then goto** 8;	**while** $x\neq0$ **do begin**
writeln(x);	writeln(x);
4: read($next$);	**repeat**
if $next=x$ **then goto** 4;	read($next$);
$x := next$;	**until** $next\neq x$;
goto 2;	$x := next$;
8: . . .;	**end**;

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* $\boxed{1\ 1}\ \boxed{2\ 2\ 2}\ \boxed{3}\ \boxed{1}\ \boxed{4\ 4}$.
- Design the program around invariants:

```
read($x$);
**while** $x$ is not the end marker **do begin**
    { $x$ is the first element of a run }
    writeln($x$);
    **repeat** read($next$) **until** $next \neq x$;
    { we have read one element too many }
    $x := next$;
**end**;
```

Syntax-Directed Control Flow

- **Sequencing**: sequential composition of statements
  - Control flows sequentially through a sequence of statements.
  - $temp := x;\ \ x := y;\ \ y := temp$
- **Selection**: conditional (and case) statements
  - **if** $\langle expression \rangle$ **then** $\langle statement_1 \rangle$ **else** $\langle statement_2 \rangle$
  - A variant: **if** $\langle expression \rangle$ **then** $\langle statement \rangle$
- **Looping**: while, repeat, and for statements
  - **while** $\langle expression \rangle$ **do** $\langle statement \rangle$
  - **repeat** $\langle statement \rangle$ **until** $\langle expression \rangle$
  - **for** $\langle name \rangle := \langle expr \rangle$ **to** $\langle expr \rangle$ **do** $\langle statement \rangle$
  - **for** $\langle name \rangle := \langle expr \rangle$ **downto** $\langle expr \rangle$ **do** $\langle statement \rangle$

Principle: *single-entry/single-exit*

Syntax of Statements in Pascal

$$
\begin{array}{lcl}
\langle \mathit{statement} \rangle & ::= & \langle \mathit{expr} \rangle := \langle \mathit{expr} \rangle \\
& | & \langle \mathit{name} \rangle \; (\; \langle \mathit{expr\_list} \rangle \;) \\
& | & \textbf{begin} \; \langle \mathit{statement\_list} \rangle \; \textbf{end} \\
& | & \textbf{if} \; \langle \mathit{expr} \rangle \; \textbf{then} \; \langle \mathit{statement} \rangle \\
& | & \textbf{if} \; \langle \mathit{expr} \rangle \; \textbf{then} \; \langle \mathit{statement} \rangle \; \textbf{else} \; \langle \mathit{statement} \rangle \\
& | & \textbf{while} \; \langle \mathit{expr} \rangle \; \textbf{do} \; \langle \mathit{statement} \rangle \\
& | & \textbf{repeat} \; \langle \mathit{statement} \rangle \; \textbf{until} \; \langle \mathit{expr} \rangle \\
& | & \textbf{for} \; \langle \mathit{name} \rangle := \langle \mathit{expr} \rangle \; \textbf{to} \; \langle \mathit{expr} \rangle \; \textbf{do} \\
& & \langle \mathit{statement} \rangle \\
& | & \textbf{for} \; \langle \mathit{name} \rangle := \langle \mathit{expr} \rangle \; \textbf{downto} \; \langle \mathit{expr} \rangle \; \textbf{do} \\
& & \langle \mathit{statement} \rangle \\
& | & \textbf{case} \; \langle \mathit{expr} \rangle \; \textbf{of} \; \langle \mathit{cases} \rangle
\end{array}
$$

Syntax of Statements in Pascal (cont.)

$$
\begin{array}{lcl}
\langle \textit{statement\_list} \rangle & ::= & \langle \textit{empty} \rangle \\
& | & \langle \textit{statement} \rangle \; ; \; \langle \textit{statement\_list} \rangle
\end{array}
$$

$$
\begin{array}{lcl}
\langle \textit{cases} \rangle & ::= & \langle \textit{constant} \rangle \; : \; \langle \textit{statement} \rangle \\
& | & \langle \textit{constant} \rangle \; : \; \langle \textit{statement} \rangle \; ; \; \langle \textit{cases} \rangle
\end{array}
$$

A Style of Nesting Conditionals

IM NTU

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

```
if      ··· then ···
else if ··· then ···
else if ··· then ···
else    ···
```

Example:

```
if (year mod 400) = 0 then leap := true
else if (year mod 100) = 0 then leap := false
else if (year mod 4) = 0 then leap := true
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?

Case Statements

- A case statement uses the value of an expression to select one of several substatements for execution.

```
**case** ⟨*expression*⟩ **of**
⟨$constant_1$⟩ : ⟨$statement_1$⟩;
⟨$constant_2$⟩ : ⟨$statement_2$⟩;
. . .
⟨$constant_n$⟩ : ⟨$statement_n$⟩;
**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?

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:
  1. A small number of cases is implemented using conditionals.
  2. For a larger number of cases, the compiler uses a "jump table" if, say, at least half the entries will be used.
  3. 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.

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
  begin **stmt**$_1$; **stmt**$_2$; **stmt**$_3$ **end**
- Inserting an empty statement between **stmt**$_3$ and **end** makes semicolons look like terminators:
  begin **stmt**$_1$; **stmt**$_2$; **stmt**$_3$; **end**
- But empty statements make the placement of semicolons significant; insertion of a semicolon can change the meaning of a program in Pascal.
  if **expr** **then** ; **stmt**
  is not the same as
  if **expr** **then** **stmt**
- Modula-2 avoids the problem by attaching a closing keyword **end**:
  if **expr** **then** **stmt** **end**

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
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:

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

Break and Continue Statements (cont.)

- A corresponding fragment for continue statements:

```
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

  $$\textbf{return}\ \langle \textit{expression} \rangle$$

  sends control back from a procedure to a caller, carrying the value of $\langle \textit{expression} \rangle$. If the retrun 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:

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

- By itself, **goto** L gives no indication of where label L is to be found. Similarly, $L : \langle \mathit{statement} \rangle$ does not indicate from where control might come to it.
- Although **goto** statements can be misused to write unreadable programs, there is still a need for them, e.g., in automatically generated programs.

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
    { $y > 0$ and $x \geq y$ }
    $x := x - y$
    { $y > 0$ and $x \geq 0$ }
end
{ $y > 0$ and $x < y$ }
```

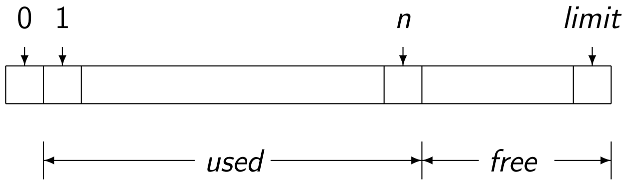
An Annotated Program

```
{$x \geq 0 \land y \geq 0 \land gcd(x, y) = gcd(m, n)$}
while $x \neq 0$ and $y \neq 0$ do
begin
    {$x \geq 0 \land y \geq 0 \land gcd(x, y) = gcd(m, n)$}
    if $x < y$
    then $swap(x, y)$;
    {$x \geq y \land y \geq 0 \land gcd(x, y) = gcd(m, n)$}
    $x := x - y$;
    {$x \geq 0 \land y \geq 0 \land gcd(x, y) = gcd(m, n)$}
end;
{$(x = 0 \land y \geq 0 \land y = gcd(x, y) = gcd(m, n)) \lor$
 $(x \geq 0 \land y = 0 \land x = gcd(x, y) = gcd(m, n))$}
```

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 $0 \leq n$, and $0 \leq n \leq limit$.
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.

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** $i$;
```

Development of a Search Program (cont.)

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 $i$;
```

Making the Sentinel Explicit

```
$A[0] := x$;
further initialization;
{ $x = A[0]$ and $x$ is not in $A[i+1..n]$ and $0 \leq i \leq n$ }
while not yet time to stop and $x$ not found at $i$ do
    $i := i - 1$;
{ $x = A[i]$ and $x$ is not in $A[i+1..n]$ and $0 \leq i \leq n$ }
return $i$;
```

Development of a Search Program (cont.)

Final Developed Program Fragment

```
$A[0] := x$;
$i := n$;
**while** $x \neq A[i]$ **do**
    $i := i - 1$;
**return** $i$;
```

Proof Rules

$$\{Q[E/x]\}\ x := E\ \{Q\} \qquad \text{(Assignment Axiom)}$$

$$\frac{\{P\}\ S_1\ \{Q\},\ \{Q\}\ S_2\ \{R\}}{\{P\}\ S_1;S_2\ \{R\}} \qquad \text{(Composition Rule)}$$

$$\frac{\{P \wedge E\}\ S_1\ \{Q\},\ \{P \wedge \neg E\}\ S_2\ \{Q\}}{\{P\}\ \textbf{if}\ E\ \textbf{then}\ S_1\ \textbf{else}\ S_2\ \{Q\}} \qquad \text{(Conditional Rule)}$$

$$\frac{\{P \wedge E\}\ S\ \{P\}}{\{P\}\ \textbf{while}\ E\ \textbf{do}\ S\ \{P \wedge \neg E\}} \qquad (\textbf{while}\ \text{Rule})$$

$$\frac{P \text{ implies } P',\ \{P'\}\ S\ \{Q'\},\ Q' \text{ implies } Q}{\{P\}\ S\ \{Q\}} \qquad \text{(Rule of Consequence)}$$

Control Flow in C

C	Pascal
if (E) S	**if** E **then** S
if (E) S_1 **else** S_2	**if** E **then** S_1 **else** S_2
while (E) S	**while** E **do** S
do S **while** (E) ;	**repeat** S **until** (not E)
for (i=1; i<=n; i++) S	**for** i:=1 **to** n **do** S
for (i=n; i>=1; i−−) S	**for** i:=n **downto** 1 **do** S

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

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.

```
while ( (c = getchar()) != EOF )
   putchar(c);
```

is semantically equivalent to

```
while ( 1 ) {
   c = getchar();
   if ( c == EOF ) break;
   putchar(c);
}
```

For Loops in C

- The `for` statement has the form

  `for (` E_1`;` E_2`;` E_3 `)` S

  E_1 is evaluated just before loop entry, E_2 is the condition for staying within the loop, and E_3 is evaluated just before every next iteration of the loop.

- The while vs. for statements:

```
while ( x != A[i] )
    --i;
```

  can be rewritten as

```
for ( ; x != A[i]; --i)
      ;
```

- A missing E_2 is taken to be true; `for(;;)` can thus be read as "forever" because it sets up an infinite loop.

Syntax of Statements in C

$$
\begin{array}{lcl}
S & ::= & ; \\
 & \mid & E\ ; \\
 & \mid & \{\ \mathit{Slist}\ \} \\
 & \mid & \text{if}\ (\ E\ )\ S \\
 & \mid & \text{if}\ (\ E\ )\ S\ \text{else}\ S \\
 & \mid & \text{while}\ (\ E\ )\ S \\
 & \mid & \text{do}\ S\ \text{while}\ (\ E\ )\ ; \\
 & \mid & \text{for}\ (\ \mathit{Eopt}\ ;\ \mathit{Eopt}\ ;\ \mathit{Eopt})\ S \\
 & \mid & \text{switch}\ (\ E\ )\ S \\
 & \mid & \text{case}\ \mathit{Constant}\ :\ S \\
 & \mid & \text{default}\ :\ S \\
 & \mid & \text{break}\ ; \\
 & \mid & \text{continue}\ ; \\
 & \mid & \text{return}\ ; \\
 & \mid & \text{return}\ E\ ; \\
 & \mid & \text{goto}\ L\ ; \\
 & \mid & L\ :\ S
\end{array}
$$

Syntax of Statements in C (cont.)

$$\begin{array}{lcl} \mathit{Slist} & ::= & \langle \mathit{empty} \rangle \\ & \mid & \mathit{Slist}\ S \\ \\ \mathit{Eopt} & ::= & \langle \mathit{empty} \rangle \\ & \mid & E \end{array}$$