Imperative Programming Languages

Thomas Sewell
UNSW
Term 3 2024

Imperative Programming

imperō

Definition

Imperative programming is where programs are described as a series of *statements* or commands to manipulate mutable *state* or cause externally observable *effects*.

States may take the form of a *mapping* from variable names to their values, or even a model of a CPU state with a memory model (for example, in an *assembly language*).

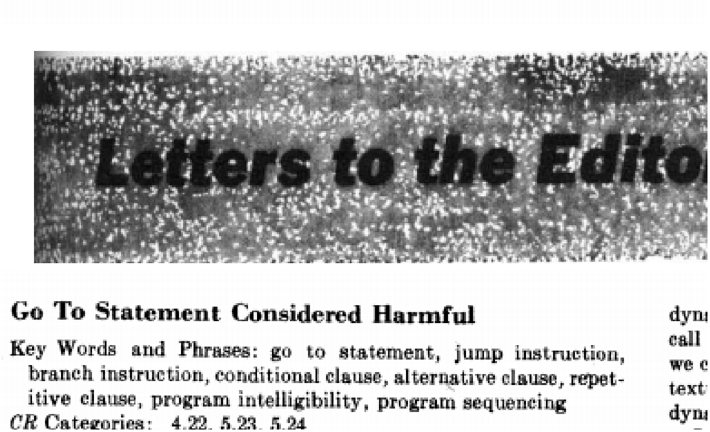
The Old Days

Early microcomputer languages used a line numbering system with GO TO statements used to arrange control flow.

Factorial Example in BASIC (1964)

```
10 N = 4
20 I = 0
30 M = 1
40 IF I > = N THEN GOTO 100
50 I = I + 1
60 M = M * I
70 GOTO 40
100 PRINT M
110 END
```

Dijkstra (1968)


Letters to the Editor

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

The *structured programming* movement brought in *control structures* to mainstream use, such as conditionals and loops.

Factorial Example in Pascal (1970)

```
program factorial;
var n : integer;
    m : integer;
    i : integer;
begin
n := 5;
m := 1;
i := 0;
while (i < n) do
begin
  i := i + 1;
  m := m * i;
end;
println(m);
```

Syntax

We're going to specify a language **TinyImp**, based on structured programming. The syntax consists of statements and expressions.

Grammar

$$
\begin{array}{lcll}
\textbf{Stmt} & ::= & \texttt{skip} & \textit{Do nothing} \\
& \mid & x \;\texttt{:=}\; \textbf{Expr} & \textit{Assignment} \\
& \mid & \texttt{var}\; y \cdot \textbf{Stmt} & \textit{Declaration} \\
& \mid & \texttt{if}\; \textbf{Expr}\; \texttt{then}\; \textbf{Stmt}\; \texttt{else}\; \textbf{Stmt}\; \texttt{fi} & \textit{Conditional} \\
& \mid & \texttt{while}\; \textbf{Expr}\; \texttt{do}\; \textbf{Stmt}\; \texttt{od} & \textit{Loop} \\
& \mid & \textbf{Stmt}\; ;\; \textbf{Stmt} & \textit{Sequencing} \\
\textbf{Expr} & ::= & \langle \textit{Arithmetic expressions} \rangle &
\end{array}
$$

We already know how to make unambiguous abstract syntax, so we will use concrete syntax in the rules for readability.

Examples

Example (Factorial and Fibonacci)

$$
\begin{array}{l}
\textbf{var } i \cdot \\
\textbf{var } m \cdot \\
i := 0; \\
m := 1; \\
\textbf{while } i < N \textbf{ do} \\
\quad i := i + 1; \\
\quad m := m \times i \\
\textbf{od}
\end{array}
$$

$$
\begin{array}{l}
\textbf{var } m \cdot \textbf{var } n \cdot \textbf{var } i \cdot \\
m := 1; n := 1; \\
i := 1; \\
\textbf{while } i < N \textbf{ do} \\
\quad \textbf{var } t \cdot t := m; \\
\quad m := n; \\
\quad n := m + t; \\
\quad i := i + 1 \\
\textbf{od}
\end{array}
$$

Static Semantics

Types? We only have one type (`int`), so type checking is a wash.

Scopes? We have to check that variables are declared before use.

Anything Else? We have to check that variables are *initialized* before they are used!

Indicates that no unsafe reads occur

Set of initialized free variables

$$U; V \vdash s\ \mathbf{ok} \rightsquigarrow W$$

Set of declared free variables

Set of definitely written to free variables

Note: $V \subseteq U$

Static Semantics Rules

$$\frac{}{U; V \vdash \texttt{skip}\ \textbf{ok} \leadsto \emptyset} \qquad \frac{x \in U \qquad \mathrm{FV}(e) \subseteq V}{U; V \vdash x\ \texttt{:=}\ e\ \textbf{ok} \leadsto \{x\}}$$

$$\frac{U \cup \{y\}; V \vdash s\ \textbf{ok} \leadsto W}{U; V \vdash \texttt{var}\ y \cdot s\ \textbf{ok} \leadsto W \setminus \{y\}}$$

$$\frac{\mathrm{FV}(e) \subseteq V \qquad U; V \vdash s_1\ \textbf{ok} \leadsto W_1 \qquad U; V \vdash s_2\ \textbf{ok} \leadsto W_2}{U; V \vdash \texttt{if}\ e\ \texttt{then}\ s_1\ \texttt{else}\ s_2\ \texttt{fi}\ \textbf{ok} \leadsto W_1 \cap W_2}$$

$$\frac{\mathrm{FV}(e) \subseteq V \qquad U; V \vdash s\ \textbf{ok} \leadsto W}{U; V \vdash \texttt{while}\ e\ \texttt{do}\ s\ \texttt{od}\ \textbf{ok} \leadsto \emptyset}$$

$$\frac{U; V \vdash s_1\ \textbf{ok} \leadsto W_1 \qquad U; (V \cup W_1) \vdash s_2\ \textbf{ok} \leadsto W_2}{U; V \vdash s_1; s_2\ \textbf{ok} \leadsto W_1 \cup W_2}$$

Dynamic Semantics

We will use big-step operational semantics. What are the sets of evaluable expressions and values here?

Evaluable Expressions: A pair containing a statement to execute **and** a *state* σ.
Values: The final state that results from executing the statement.
States: mutable mappings from states to values.

States

A *state* is a mutable mapping from variables to their values. We use the following notation:

- To read a variable x from the state σ, we write $\sigma(x)$.
- To update an existing variable x to have value v inside the state σ, we write $(\sigma : x \mapsto v)$.
- To extend a state σ with a new, previously undeclared variable x, we write $\sigma \cdot x$. In such a state, $(\sigma \cdot x)(x)$ is undefined.
- To *remove* a variable x from the set of declared variables, we write $(\sigma|_x)$.
- To exit a local scope for x, returning to the previous scope σ':

$$\sigma|_x^{\sigma'} = \begin{cases} \sigma|_x & \text{if } x \text{ is undeclared in } \sigma' \\ (\sigma|_x) \cdot x & \text{if } x \text{ is declared but undefined in } \sigma' \\ (\sigma : x \mapsto \sigma'(x)) & \text{if } \sigma'(x) \text{ is defined} \end{cases}$$

Evaluation Rules

We will assume we have defined a relation $\sigma \vdash e \Downarrow v$ for arithmetic expressions, much like in the previous lecture.

$$\frac{}{(\sigma, \texttt{skip}) \Downarrow \sigma} \qquad \frac{(\sigma_1, s_1) \Downarrow \sigma_2 \qquad (\sigma_2, s_2) \Downarrow \sigma_3}{(\sigma_1, s_1; s_2) \Downarrow \sigma_3}$$

$$\frac{\sigma \vdash e \Downarrow v}{(\sigma, x := e) \Downarrow (\sigma : x \mapsto v)} \qquad \frac{(\sigma_1 \cdot x, s) \Downarrow \sigma_2}{(\sigma_1, \texttt{var}\ x \cdot s) \Downarrow \sigma_2|_x^{\sigma_1}}$$

$$\frac{\sigma_1 \vdash e \Downarrow v \qquad v \neq 0 \qquad (\sigma_1, s_1) \Downarrow \sigma_2}{(\sigma_1, \texttt{if}\ e\ \texttt{then}\ s_1\ \texttt{else}\ s_2\ \texttt{fi}) \Downarrow \sigma_2}$$

$$\frac{\sigma_1 \vdash e \Downarrow 0 \qquad (\sigma_1, s_2) \Downarrow \sigma_2}{(\sigma_1, \texttt{if}\ e\ \texttt{then}\ s_1\ \texttt{else}\ s_2\ \texttt{fi}) \Downarrow \sigma_2}$$

Evaluation Rules Pt 2

$$\frac{\sigma_1 \vdash e \Downarrow 0}{(\sigma_1, \texttt{while}\ e\ \texttt{do}\ s\ \texttt{od}) \Downarrow \sigma_1}$$

$$\frac{\begin{array}{c}\sigma_1 \vdash e \Downarrow v \quad v \neq 0 \\ (\sigma_1, s) \Downarrow \sigma_2 \quad (\sigma_2, \texttt{while}\ e\ \texttt{do}\ s\ \texttt{od}) \Downarrow \sigma_3\end{array}}{(\sigma_1, \texttt{while}\ e\ \texttt{do}\ s\ \texttt{od}) \Downarrow \sigma_3}$$

Big-Step vs Small-Step Semantics

Consider this (silly) infinite loop:

$$\begin{aligned} p \equiv\ & \textbf{while}\ 1 < 2\ \textbf{do} \\ & \quad \textbf{skip} \\ & \textbf{od} \end{aligned}$$

Can we ever prove $(\sigma_1, p) \Downarrow \sigma_2$? No. We can prove that by induction.

If we had defined a *small-step* semantics instead, we would be able to describe this non-termination situation.

It is not practical to define a denotational semantics for a program with loops or recursion.

Alternative declaration semantics

What should happen when an uninitialised variable is used?

$$(\sigma \cdot y, \textbf{var } x \cdot y := x + 1) \Downarrow ??$$

$$\dfrac{\dfrac{???}{(\sigma \cdot y \cdot x, y := x + 1) \Downarrow ??}}{(\sigma \cdot y, \textbf{var } x \cdot y := x + 1) \Downarrow ??}$$

We can't apply the assignment rule here, because in the state $\sigma \cdot y \cdot x$, $\sigma(x)$ is undefined.

Alternative declaration semantics

Crash and burn: $(\sigma \cdot y, \textbf{var } x \cdot y := x + 1) \not\Downarrow$

$$\frac{(\sigma_1 \cdot x, s) \Downarrow \sigma_2}{(\sigma_1, \texttt{var } x \cdot s) \Downarrow \sigma_2|_x^{\sigma_1}}$$

Default value: $(\sigma \cdot y, \textbf{var } x \cdot y := x + 1) \Downarrow (\sigma \cdot y) : y \mapsto 1$

$$\frac{((\sigma_1 \cdot x) : x \mapsto 0, s) \Downarrow \sigma_2}{(\sigma_1, \texttt{var } x \cdot s) \Downarrow \sigma_2|_x^{\sigma_1}}$$

Junk data: $(\sigma \cdot y, \textbf{var } x \cdot y := x + 1) \Downarrow (\sigma \cdot y) : y \mapsto 3$ (or 4, or whatever we want...)

$$\frac{((\sigma_1 \cdot x) : x \mapsto n, s) \Downarrow \sigma_2}{(\sigma_1, \texttt{var } x \cdot s) \Downarrow \sigma_2|_x^{\sigma_1}}$$

Hoare Logic

For a taste of *axiomatic semantics*, let's define a *Hoare Logic* for TinyImp (without **var**). We write a *Hoare triple* judgement as:

$$\{\varphi\}\ s\ \{\psi\}$$

Where φ and ψ are logical formulae about states, called *assertions*, and s is a statement. This triple states that if the statement s successfully evaluates from a starting state satisfying the *precondition* φ, then the final state will satisfy the *postcondition* ψ:

$$\varphi(\sigma) \wedge (\sigma, s) \Downarrow \sigma' \Rightarrow \psi(\sigma')$$

Proving Hoare Triples

To prove a Hoare triple like:

$$
\begin{array}{l}
\{\mathrm{True}\} \\
i := 0; \\
m := 1; \\
\textbf{while } i \neq N \textbf{ do} \\
\quad i := i + 1; \\
\quad m := m \times i \\
\textbf{od} \\
\{m = N!\}
\end{array}
$$

We *could* prove this using the operational semantics. This is cumbersome, and requires an induction to deal with the **while** loop. Instead, we'll define a set of rules to prove Hoare triples directly (called *a proof calculus*).

Hoare Rules

$$\frac{}{(\sigma, \texttt{skip}) \Downarrow \sigma} \qquad \frac{}{\{\varphi\}\ \texttt{skip}\ \{\varphi\}}$$

$$\frac{(\sigma_1, s_1) \Downarrow \sigma_2 \qquad (\sigma_2, s_2) \Downarrow \sigma_3}{(\sigma_1, s_1; s_2) \Downarrow \sigma_3} \qquad \frac{\{\varphi\}\ s_1\ \{\alpha\} \qquad \{\alpha\}\ s_2\ \{\psi\}}{\{\varphi\}\ s_1; s_2\ \{\psi\}}$$

$$\frac{\sigma \vdash e \Downarrow v}{(\sigma, x := e) \Downarrow (\sigma : x \mapsto v)} \qquad \frac{}{\{\varphi[x := e]\}\ x := e\ \{\varphi\}}$$

Continuing on, we can get rules for `if`, and `while` with a *loop invariant*:

$$\frac{\{\varphi \land e\}\ s_1\ \{\psi\} \quad \{\varphi \land \neg e\}\ s_2\ \{\psi\}}{\{\varphi\}\ \texttt{if}\ e\ \texttt{then}\ s_1\ \texttt{else}\ s_2\ \texttt{fi}\ \{\psi\}} \qquad \frac{\{\varphi \land e\}\ s\ \{\varphi\}}{\{\varphi\}\ \texttt{while}\ e\ \texttt{do}\ s\ \texttt{od}\ \{\varphi \land \neg e\}}$$

Consequence

There is one more rule, called the *rule of consequence*, that we need to insert ordinary logical reasoning into our Hoare logic proofs:

$$\frac{\varphi \Rightarrow \alpha \qquad \{\alpha\}\ \mathsf{s}\ \{\beta\} \qquad \beta \Rightarrow \psi}{\{\varphi\}\ \mathsf{s}\ \{\psi\}}$$

This is the only rule that is not directed entirely by syntax. This means a Hoare logic proof need not look like a derivation tree. Instead we can sprinkle assertions through our program and specially note uses of the consequence rule.

Factorial Example

Let's verify the Factorial program using our Hoare rules:

$$
\begin{array}{l}
\{\text{True}\} \\
\{1 = 0!\}\ i := 0;\ \{1 = i!\} \\
\{1 = i!\}\ m := 1;\ \{m = i!\} \\
\{m = i!\} \\
\textbf{while}\ i \neq N\ \textbf{do}\ \{m = i! \wedge i \neq N\} \\
\quad \{m \times (i+1) = (i+1)!\} \\
\quad i := i + 1; \\
\quad \{m \times i = i!\} \\
\quad m := m \times i \\
\quad \{m = i!\} \\
\textbf{od}\ \{m = i! \wedge i = N\} \\
\{m = N!\}
\end{array}
$$

$$\frac{\{\varphi \wedge e\}\ s_1\ \{\psi\} \quad \{\varphi \wedge \neg e\}\ s_2\ \{\psi\}}{\{\varphi\}\ \texttt{if}\ e\ \texttt{then}\ s_1\ \texttt{else}\ s_2\ \texttt{fi}\ \{\psi\}}$$

$$\frac{}{\{\varphi[x := e]\}\ x := e\ \{\varphi\}}$$

$$\frac{\{\varphi \wedge e\}\ s\ \{\varphi\}}{\{\varphi\}\ \texttt{while}\ e\ \texttt{do}\ s\ \texttt{od}\ \{\varphi \wedge \neg e\}}$$

$$\frac{\{\varphi\}\ s_1\ \{\alpha\} \quad \{\alpha\}\ s_2\ \{\psi\}}{\{\varphi\}\ s_1; s_2\ \{\psi\}}$$

$$\frac{\varphi \Rightarrow \alpha \quad \{\alpha\}\ s\ \{\beta\} \quad \beta \Rightarrow \psi}{\{\varphi\}\ s\ \{\psi\}}$$

note: $(i+1)! = i! \times (i+1)$

Forward-Directed and Backward-Directed

What is the tension between these two rules?

$$\frac{}{\{\varphi[x := e]\}\ x := e\ \{\varphi\}}$$

$$\frac{\{\varphi \wedge e\}\ s_1\ \{\psi\} \quad \{\varphi \wedge \neg e\}\ s_2\ \{\psi\}}{\{\varphi\}\ \texttt{if}\ e\ \texttt{then}\ s_1\ \texttt{else}\ s_2\ \texttt{fi}\ \{\psi\}}$$

It is convenient to write (most of) our rules so that they can always be applied forwards or backwards. Dijkstra-style backward propagation generally works better.

$$\frac{\{\varphi_1\}\ s_1\ \{\psi\} \quad \{\varphi_2\}\ s_2\ \{\psi\}}{\{(e \longrightarrow \varphi_1) \wedge (\neg e \longrightarrow \varphi_2)\}\ \texttt{if}\ e\ \texttt{then}\ s_1\ \texttt{else}\ s_2\ \texttt{fi}\ \{\psi\}}$$