Imperative Programming

Liam O'Connor

CSE, UNSW (and data61)

Semester 2 2018

Imperative Programming

imperō

Definition

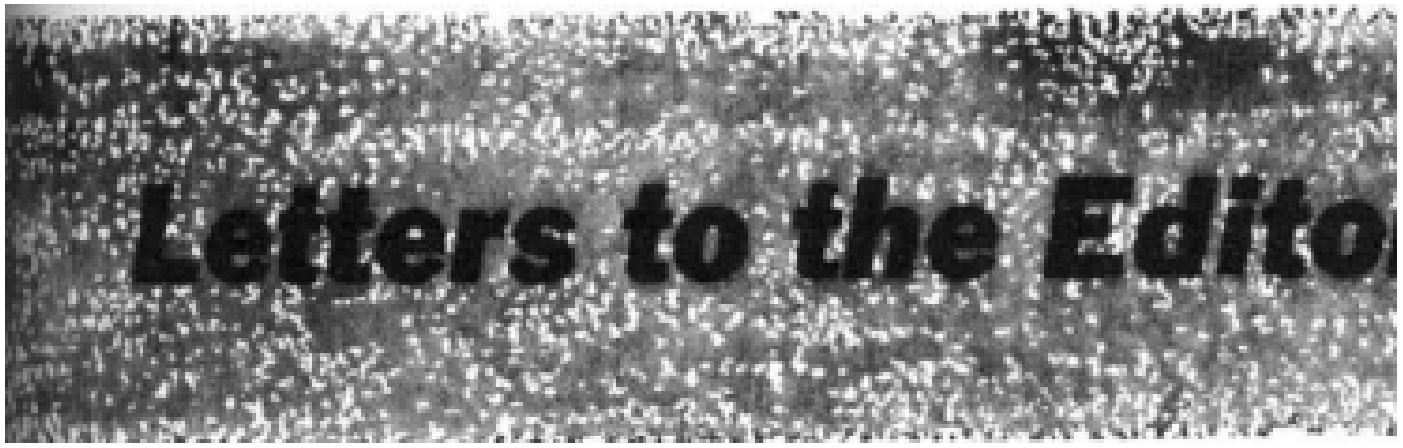
Imperative programming is where programs are described as a series of *statements* or commands to manipulate mutable *state* or cause externally observable *effects*.

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.

Dijkstra



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.

Syntax

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

Grammar

Stmt	::=	`skip`	*Do nothing*
	|	*x* `:=` **Expr**	*Assignment*
	|	`var` *y* · **Stmt**	*Declaration*
	|	`if` **Expr** `then` **Stmt** `else` **Stmt** `fi`	*Conditional*
	|	`while` **Expr** `do` **Stmt** `od`	*Loop*
	|	**Stmt** `;` **Stmt**	*Sequencing*
Expr	::=	⟨*Arithmetic expressions*⟩	

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)

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

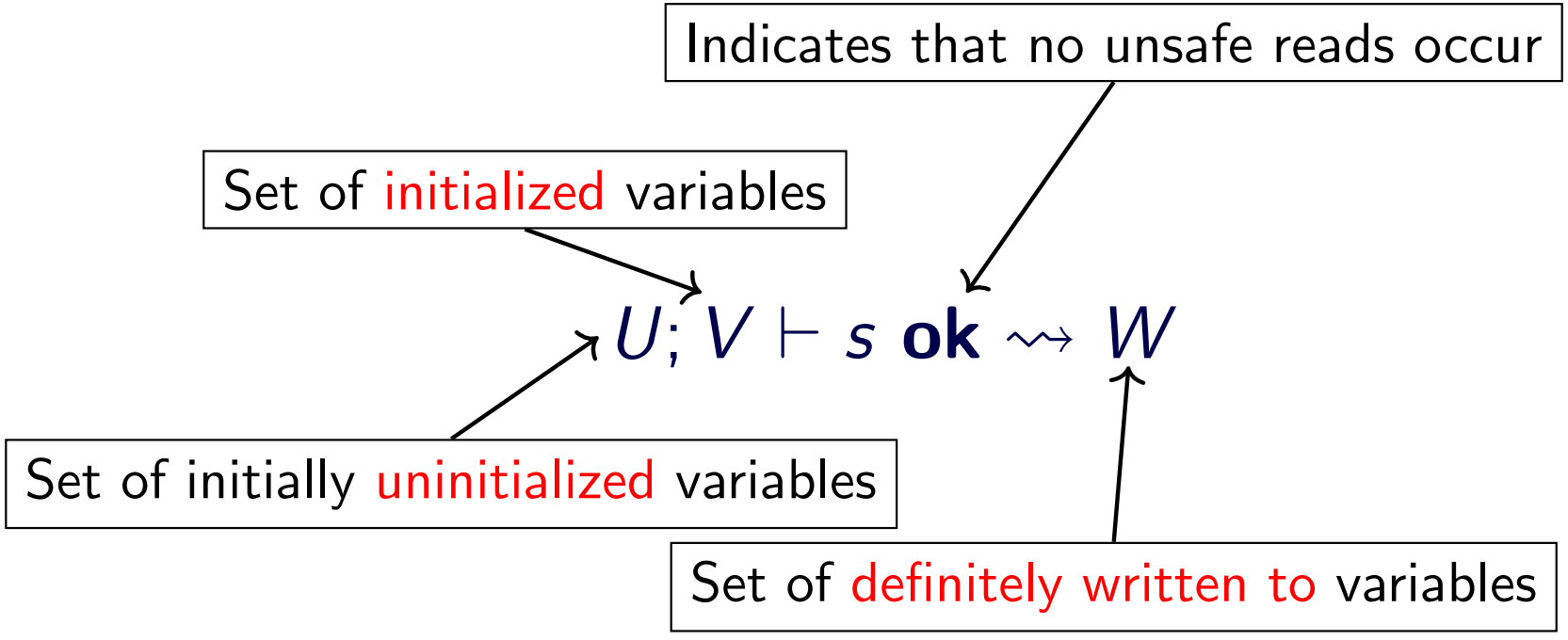
$$
\begin{array}{l}
\textbf{var } m \cdot \textbf{var } n \cdot \textbf{var } i \cdot \\
m := 1; n := 1; \\
i := 1; \\
\textbf{while } i < N \textbf{ do} \\
\quad \textbf{var } t \cdot t := m; \\
\quad m := n; \\
\quad n := m + t; \\
\quad i := i + 1 \\
\textbf{od}
\end{array}
$$

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!



Static Semantics Rules

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

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

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

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

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

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

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, x has undefined value.

Evaluation Rules

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

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

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

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

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

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

Hoare Logic

To give you a taste of *axiomatic semantics*, and also how *formal verification* works, we are going to define what's called a *Hoare Logic* for TinyImp to allow us to prove properties of our program. We write a *Hoare triple* judgement as:

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

Where φ and ψ are logical formulae about state variables, 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 result state will satisfy the *postcondition* ψ.

Proving Hoare Triples

To prove a hoare triple like:

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

It is undesirable to look at the operational semantics derivations of this whole program to compute what the possible final states are for a given input state. Instead we shall 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*:

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

Consequence

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

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

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. Time to do it on the iPad

$$\frac{\{\varphi \wedge e\}\ s_1\ \{\psi\} \quad \{\varphi \wedge \neg e\}\ s_2\ \{\psi\}}{\{\varphi\}\ \texttt{if}\ e\ \texttt{then}\ s_1\ \texttt{else}\ s_2\ \texttt{fi}\ \{\psi\}} \qquad \frac{}{\{\varphi[x := e]\}\ x := e\ \{\varphi\}}$$

$$\frac{\{\varphi \wedge e\}\ s\ \{\varphi\}}{\{\varphi\}\ \texttt{while}\ e\ \texttt{do}\ s\ \texttt{od}\ \{\varphi \wedge \neg e\}} \qquad \frac{\{\varphi\}\ s_1\ \{\alpha\} \qquad \{\alpha\}\ s_2\ \{\psi\}}{\{\varphi\}\ s_1; s_2\ \{\psi\}}$$