CS611 Lecture 02 **The IMP Language** 1 September 2003

Scribe: Adam Arbree, Art Munson Lecturer: Radu Rugina

# 1 IMP: A Simple Imperative Language

We will use IMP, which is a simple imperative language, to begin our study of formal semantics. By simplifying the language, we can focus on just the semantics. IMP contains the following constructs: sequences of commands, while loops, if-then-else branching, and assignments. Only assignments change the program state; the evaluation of expressions has no side effects.

## 1.1 Syntax

- Constants: Either integers $n \in \mathbb{Z}$, or truth values $t \in \mathbb{T}$, where $\mathbb{T} = \{\mathit{true}, \mathit{false}\}$.
- Variables: From the domain **Loc**, the set of all memory locations (e.g. $X, Y \in \mathbf{Loc}$). All of them hold integer values of arbitrary precision. Note that there is no scoping for variables. It is assumed that all of the variables used in the program are globally declared.
- Expressions:
  - Arithmetic: An arithmetic expression $a$ is a member of the set of all arithmetic expressions **Aexp**. Using BNF notation[1], we define the grammar for arithmetic expressions as follows:

$$a \;::=\; n \;|\; X \;|\; a_0 + a_1 \;|\; a_0 - a_1 \;|\; a_0 \times a_1$$

  - Boolean: A boolean expression $b$ is a member of the set of all boolean expresions **Bexp**. Using BNF notation, boolean expressions are defined as follows:

$$b \;::=\; \mathbf{true} \;|\; \mathbf{false} \;|\; a_0 = a_1 \;|\; a_0 < a_1 \;|\; b_0 \wedge b_1 \;|\; b_0 \vee b_1 \;|\; \neg b_0$$

- Commands: A command $c$ is a member of the set of all commands **Com**:

$$c \;::=\; \mathbf{skip} \;|\; X := a \;|\; c_0; c_1 \;|\; \mathbf{if}\, b\, \mathbf{then}\, c_0\, \mathbf{else}\, c_1 \;|\; \mathbf{while}\, b\, \mathbf{do}\, c_0$$

A program in IMP is a command $c \in \mathbf{Com}$. Below is a sample IMP program that calculates the greatest common denominator (GCD) of $M$ and $N$.

$$\begin{array}{l} \mathbf{while}\ \neg(M = N)\ \mathbf{do} \\ \qquad \mathbf{if}\ (M < N)\ \mathbf{then}\ T := M; M := N; N := T \\ \qquad\qquad\qquad\quad\ \mathbf{else\ skip}; \\ \qquad M := M - N \end{array}$$

Note that we are not interested in the problem of parsing, so we allow this grammar to be ambiguous.[2] Where ambiguity cannot be resolved by formatting, it will be explicitly resolved with parentheses. However, parentheses are not part of IMP's abstract syntax shown above. We will assume that there is a concrete syntax which extends the abstract syntax with parentheses, and programs use parentheses to ensure that only one abstract syntax tree exists for each program.

[1] Backus-Naur form, previously Backus Normal form, usually abbreviated BNF. BNF is a "formal metasyntax used to express context-free grammars" (*source:* http://dictionary.reference.com/search?q=backus-naur%20form).

[2] Consider the lack of arithmetic precedence: $a_0 + a_1 \times a_2$. Also consider whether or not $c_1$ is part of the body of the **while** loop in $\mathbf{while}\, b\, \mathbf{do}\, c_0; c_1$

### 1.2 General Properties of IMP

- Errors are not possible in IMP, in part because the language is so simple. For instance, to ensure the absence of errors, division is not included as an arithmetic operation.
- IMP is deterministic. A program run with a given input always produces the same output.
- Termination is guaranteed for all expressions and for commands not containing loops. However, while loops may or may not terminate.
- IMP has no complex language constructs. For example, there are no procedures, scoping, pointers, etc.
- Variables are always statically allocated; there is no dynamic allocation of variables in the heap or on the stack. In fact, there is no stack because the language doesn't have (recursive) functions.
- IMP is Turing-complete. Because the language allows integers of arbitrary precision, we can encode the tape of a Turing machine as an integer.

## 2 Operational Semantics

Operational semantics define how a program executes. Two types of operational semantics exist: large step and small step. Imagine that programs transition through sequences of configurations. In large step operational semantics, transitions can comprise many smaller steps. In constrast, small step operational semantics describes individual transitions between configurations separately. Although either type can be used for most semantic reasoning, sometimes one type is better suited than the other.

### 2.1 Large-Step Operational Semantics of IMP

We define machine configurations for IMP as a command to be executed, paired with the state of memory. A final configuration, reached if the program terminates, is simply a state with no command remaining to be executed.

#### 2.1.1 States

The set of all states, $\Sigma$, is the set of all functions from **Loc** to $\mathbb{Z}$. More formally,

$$\Sigma = \mathbf{Loc} \to \mathbb{Z}$$

A single state, $\sigma \in \Sigma$, is a particular function from **Loc** to $\mathbb{Z}$.

#### 2.1.2 Relations

The evaluation of commands, arithmetic expressions, and boolean expressions in IMP is defined using three relations.

- Commands: $\to_c \subseteq (\mathbf{Com} \times \Sigma) \times \Sigma$
  For example, $(c, \sigma, \sigma') \in \to_c$ reads as, "executing $c$ in $\sigma$ results in $\sigma'$." This can also be written as $\langle c, \sigma \rangle \to_c \sigma'$.

  It is worth pointing out that $\to$ is *not* used in relating $(\mathbf{Com} \times \Sigma)$ to $\Sigma$; instead, the Cartesian product is used. This allows us to model command evaluations that are nondeterministic and nonterminating.
- Arithmetic expressions: $\to_a \subseteq (\mathbf{Aexp} \times \Sigma) \times \mathbb{Z}$. So, $\langle a, \sigma \rangle \to_a n$. Note that the state of memory does not change, because there are no side effects.
- Boolean expressions: $\to_b \subseteq (\mathbf{Bexp} \times \Sigma) \times \mathbb{T}$. So, $\langle b, \sigma \rangle \to_b t$. Again, there are no side effects.

Normally, the subscripts $a, b, c$ on $\to$ are omitted since they are clear from context.