

A Simple Imperative Language Operational Semantics

Lecture 2 CS263

CS 263 Lecture 2

1

Syntax of IMP

- Concrete syntax
 - The rules by which programs can be expressed as strings of characters
 - Deals with issues like keywords, identifiers, statement separators (terminators), comments, indentation, etc.
- Concrete syntax is important in practice
 - For readability, familiarity, parsing speed, effectiveness of error recovery, clarity of error messages
- Well understood principles
 - Use finite automata and context-free grammars
 - Automatic lexer/parser generators

CS 263 Lecture 2

3

Plan

- We'll study a simple imperative language IMP
 - Abstract syntax
 - Operational semantics
 - Denotational semantics
 - Axiomatic semantics

... and relationships between various semantics (with proofs)
- Today: operational semantics (Chapter 2 of Winskel)

CS 263 Lecture 2

2

Abstract Syntax

- We ignore parsing issues and study programs given as abstract syntax trees
- Abstract syntax tree is the parse tree of the program
 - Ignores issues like comment conventions
 - More convenient for formal and algorithmic manipulation

CS 263 Lecture 2

4

IMP Syntactic Entities

- int integer constants
 - $n \in \mathbb{Z}$
- bool boolean constants
 - true, false
- L locations (updateable variables)
 - x, y, ...
- Aexp arithmetic expressions
 - e
- Bexp boolean expressions
 - b
- Com commands
 - c

CS 263 Lecture 2

5

Abstract Syntax (Aexp)

- Arithmetic expressions (Aexp)
 - $e ::= n$ for $n \in \mathbb{Z}$
 - | x for $x \in L$
 - | $e_1 + e_2$ for $e_1, e_2 \in \text{Aexp}$
 - | $e_1 - e_2$ for $e_1, e_2 \in \text{Aexp}$
 - | $e_1 * e_2$ for $e_1, e_2 \in \text{Aexp}$
- Notes:
 - Variables are not declared
 - All variables have integer type
 - No side-effects (in expressions)

CS 263 Lecture 2

6

Abstract Syntax (Bexp)

- Boolean expressions (Bexp)
 - $b ::= \text{true}$
 - $| \text{false}$
 - $| e_1 = e_2$ for $e_1, e_2 \in \text{Aexp}$
 - $| e_1 \leq e_2$ for $e_1, e_2 \in \text{Aexp}$
 - $| \neg b$ for $b \in \text{Bexp}$
 - $| b_1 \wedge b_2$ for $b_1, b_2 \in \text{Bexp}$
 - $| b_1 \vee b_2$ for $b_1, b_2 \in \text{Bexp}$

CS 263 Lecture 2

7

Abstract Syntax (Comm)

- Commands (Comm)
 - $c ::= \text{skip}$
 - $| x := e$ for $x \in L$ and $e \in \text{Aexp}$
 - $| c_1 ; c_2$ for $c_1, c_2 \in \text{Comm}$
 - $| \text{if } b \text{ then } c_1 \text{ else } c_2$ for $c_1, c_2 \in \text{Comm}$ and $b \in \text{Bexp}$
 - $| \text{while } b \text{ do } c$ for $c \in \text{Comm}$ and $b \in \text{Bexp}$
- Notes:
 - The typing rules have been embedded in the syntax definition
 - Other parts are not context-free and need to be checked separately (e.g., all variables are declared)
 - Commands contain all the side-effects in the language
 - Missing: pointers, function calls

CS 263 Lecture 2

8

Analysis of IMP

- Questions to answer:
 - What is the "meaning" of a given IMP expression/command?
 - How would we go about evaluating IMP expressions and commands?
 - How are the evaluator and the meaning related?
 - How can we reason about the effect of a command?

CS 263 Lecture 2

9

An Operational Semantics

- Specifies how expressions and commands should be evaluated
- Operational semantics abstracts the execution of a concrete interpreter
- Depending on the form of the expression
 - 0, 1, 2, ... don't evaluate any further.
 - They are normal forms or values.
 - $e_1 + e_2$ is evaluated by first evaluating e_1 to n_1 , then evaluating e_2 to n_2 .
 - The result of the evaluation is the literal representing $n_1 + n_2$.
 - Similar for $e_1 * e_2$

CS 263 Lecture 2

10

Semantics of IMP

- The meaning of IMP expressions depends on the values of variables
- The value of variables at a given moment is abstracted as a function from L to nat (a state)
- The set of all states is $\Sigma = L \rightarrow \mathbb{Z}$
- We shall use σ to range over Σ

CS 263 Lecture 2

11

Notation: Judgment

- We write $\langle e, \sigma \rangle \Downarrow n$ to mean that e evaluates to n in state σ
 - This is a judgment (a statement about a relation between e, σ and n)
 - In this case, we can view \Downarrow as a partial function of two arguments e and σ
- This formulation is called natural operational semantics
 - or big-step operational semantics
 - the judgment relates the expression and its "meaning"
- How can we define $\langle e_1 + e_2, \sigma \rangle \Downarrow \dots ?$

CS 263 Lecture 2

12

Notation: Rules of Inference

- We express the evaluation rules as rules of inference for our judgment
 - called the derivation rules for the judgment
 - also called the evaluation rules (for operational semantics)
- In general, we have one rule for each language construct:

$$\frac{\langle e_1, \sigma \rangle \Downarrow n_1 \quad \langle e_2, \sigma \rangle \Downarrow n_2}{\langle e_1 + e_2, \sigma \rangle \Downarrow n_1 + n_2}$$

CS 263 Lecture 2

13

Evaluation Rules (for Aexp)

$$\frac{\langle n, \sigma \rangle \Downarrow n}{\langle e_1, \sigma \rangle \Downarrow n_1} \quad \frac{\langle x, \sigma \rangle \Downarrow \sigma(x)}{\langle e_2, \sigma \rangle \Downarrow n_2} \quad \frac{\langle e_1, \sigma \rangle \Downarrow n_1 \quad \langle e_2, \sigma \rangle \Downarrow n_2}{\langle e_1 + e_2, \sigma \rangle \Downarrow n_1 + n_2} \quad \frac{\langle e_1, \sigma \rangle \Downarrow n_1 \quad \langle e_2, \sigma \rangle \Downarrow n_2}{\langle e_1 - e_2, \sigma \rangle \Downarrow n_1 - n_2}$$

$$\frac{\langle e_1, \sigma \rangle \Downarrow n_1 \quad \langle e_2, \sigma \rangle \Downarrow n_2}{\langle e_1 * e_2, \sigma \rangle \Downarrow n_1 * n_2}$$

- This is called structural operational semantics
 - rules defined based on the structure of the expression
- These rules do not impose an order of evaluation !

CS 263 Lecture 2

14

Evaluation Rules (for Bexp)

$$\frac{}{\langle \text{true}, \sigma \rangle \Downarrow \text{true}} \quad \frac{}{\langle \text{false}, \sigma \rangle \Downarrow \text{false}}$$

$$\frac{\langle e_1, \sigma \rangle \Downarrow n_1 \quad \langle e_2, \sigma \rangle \Downarrow n_2}{\langle e_1 = e_2, \sigma \rangle \Downarrow n_1 = n_2} \quad \frac{\langle e_1, \sigma \rangle \Downarrow n_1 \quad \langle e_2, \sigma \rangle \Downarrow n_2}{\langle e_1 \leq e_2, \sigma \rangle \Downarrow n_1 \leq n_2}$$

$$\frac{}{\langle b_1, \sigma \rangle \Downarrow \text{false}} \quad \frac{}{\langle b_2, \sigma \rangle \Downarrow \text{false}}$$

$$\frac{}{\langle b_1 \wedge b_2, \sigma \rangle \Downarrow \text{false}} \quad \frac{}{\langle b_1 \wedge b_2, \sigma \rangle \Downarrow \text{false}}$$

$$\frac{\langle b_1, \sigma \rangle \Downarrow \text{true} \quad \langle b_2, \sigma \rangle \Downarrow \text{true}}{\langle b_1 \wedge b_2, \sigma \rangle \Downarrow \text{true}}$$

CS 263 Lecture 2

15

CS 263 Lecture 2

16

How to Read the Rules?

- Forward, as inference rules
 - if we know that the hypothesis judgments hold then we can infer that the conclusion judgment also holds
 - e.g., if we know that $\langle e_1, \sigma \rangle \Downarrow 5$ and $\langle e_2, \sigma \rangle \Downarrow 7$, then we can infer that $\langle e_1 + e_2, \sigma \rangle \Downarrow 12$

How to Read the Rules?

- Backward, as evaluation rules
 - Suppose we want to evaluate $e_1 + e_2$, i.e., find n s.t. $e_1 + e_2 \Downarrow n$ is derivable using the previous rules
 - By inspection of the rules we notice that the last step in the derivation of $e_1 + e_2 \Downarrow n$ **must be** the addition rule
 - the other rules have conclusions that would not match $e_1 + e_2 \Downarrow n$
 - this is called reasoning by inversion on the derivation rules
 - Thus we must find n_1 and n_2 such that $e_1 \Downarrow n_1$ and $e_2 \Downarrow n_2$ are derivable
 - This is done recursively
- Since there is exactly one rule for each kind of expression we say that the rules are syntax-directed
 - At each step at most one rule applies
 - This allows a simple evaluation procedure as above

CS 263 Lecture 2

17

CS 263 Lecture 2

18

Evaluation of Commands

- The evaluation of Comm has side effects but no direct result.
 - What is the result of evaluating a command ?
- The "result" of a Comm is a new state: $\langle c, \sigma \rangle \Downarrow \sigma'$
 - But the evaluation of Comm might not terminate !

Evaluation Rules (for Comm)

$\frac{\langle e, \sigma \rangle \Downarrow n}{\langle x := e, \sigma \rangle \Downarrow \sigma[x := n]}$	Def: $\sigma[x := n](x) = n$ $\sigma[x := n](y) = \sigma(y)$
$\frac{}{\langle \text{skip}, \sigma \rangle \Downarrow \sigma}$	$\frac{\langle c_1, \sigma \rangle \Downarrow \sigma' \quad \langle c_2, \sigma' \rangle \Downarrow \sigma''}{\langle c_1 ; c_2, \sigma \rangle \Downarrow \sigma''}$
$\frac{\langle b, \sigma \rangle \Downarrow \text{true} \quad \langle c_1, \sigma \rangle \Downarrow \sigma'}{\langle \text{if } b \text{ then } c_1 \text{ else } c_2, \sigma \rangle \Downarrow \sigma'}$	$\frac{\langle b, \sigma \rangle \Downarrow \text{false} \quad \langle c_2, \sigma \rangle \Downarrow \sigma'}{\langle \text{if } b \text{ then } c_1 \text{ else } c_2, \sigma \rangle \Downarrow \sigma'}$
$\frac{\langle b, \sigma \rangle \Downarrow \text{false} \quad \langle c, \sigma \rangle \Downarrow \sigma'}{\langle \text{while } b \text{ do } c, \sigma \rangle \Downarrow \sigma}$	$\frac{\langle b, \sigma \rangle \Downarrow \text{true} \quad \langle c; \text{while } b \text{ do } c, \sigma \rangle \Downarrow \sigma'}{\langle \text{while } b \text{ do } c, \sigma \rangle \Downarrow \sigma'}$

CS 263 Lecture 2

19

Evaluation of Commands. Notes.

- The order of evaluation is important
 - c_1 is evaluated before c_2 in $c_1; c_2$
 - c_2 is not evaluated in "if true then c_1 else c_2 "
 - c is not evaluated in "while false do c "
 - b is evaluated first in "if b then c_1 else c_2 "
 - this is explicit in the evaluation rules
- Conditional constructs have multiple evaluation rules
 - but only one can be applied at one time
- The evaluation rules are not syntax-directed
 - See the rule for while
 - The evaluation might not terminate
- The evaluation rules suggest an interpreter

CS 263 Lecture 2

20

Disadvantages of Natural-Style Operational Sem.

- Natural-style semantics has two disadvantages
 - It is hard to talk about commands whose evaluation does not terminate
 - There is no σ' such that $\langle c, \sigma \rangle \Downarrow \sigma'$
 - But that is true also of ill-formed or erroneous commands (in a richer language)!
 - It does not give us a way to talk about intermediate states
 - Thus we cannot say that on a parallel machine the execution of two commands is interleaved
- Small-step semantics addresses these problems
 - Execution is modeled as a (possible infinite) sequence of states

CS 263 Lecture 2

21

Contextual Semantics

- Contextual semantics is a small-step semantics where the atomic execution step is a rewrite of the program
- We will define a relation $\langle c, \sigma \rangle \rightarrow \langle c', \sigma' \rangle$
 - c' is obtained from c through an atomic rewrite step
 - Evaluation terminates when the program has been rewritten to a terminal program
 - one from which we cannot make further progress
 - For IMP the terminal command is "skip"
 - As long as the command is not "skip" we can make further progress
 - some commands never reduce to skip (e.g., while true do skip)

CS 263 Lecture 2

22

Example of Small-Step Evaluation

- Consider the small-step evaluation of $x := 1; x := x + 1$ in the initial state $[x := 0]$

State

$\langle x := 1; x := x + 1, [x := 0] \rangle$
 $\langle \text{skip}; x := x + 1, [x := 1] \rangle$
 $\langle x := x + 1, [x := 1] \rangle$
 $\langle x := 1 + 1, [x := 1] \rangle$
 $\langle x := 2, [x := 1] \rangle$
 $\langle \text{skip}, [x := 2] \rangle$

CS 263 Lecture 2

23

What is an Atomic Reduction?

- We need to define
 - What constitutes an atomic reduction step?
 - Granularity is a choice of the semantics designer
 - e.g., choice between an addition of arbitrary integers, or an addition of 32-bit integers
 - How to select the next reduction step, when several are possible?
 - This is the order of evaluation issue

CS 263 Lecture 2

24

Redexes

- A **redex** is a syntactic expression or command that can be reduced (transformed) in one atomic step
- Defined as a grammar:

$$r ::= x$$

$$\begin{array}{l} | n_1 + n_2 \\ | x := n \\ | \text{skip}; c \\ | \text{if true then } c_1 \text{ else } c_2 \\ | \text{if false then } c_1 \text{ else } c_2 \\ | \text{while } b \text{ do } c \end{array}$$
- For brevity, we mix expression and command redexes
- Note that $(1 + 3) + 2$ is not a redex, but $1 + 3$ is

CS 263 Lecture 2

25

Local Reduction Rules for IMP

- One for each redex: $\langle r, \sigma \rangle \rightarrow \langle e, \sigma' \rangle$
 - means that in state σ , the redex r can be replaced in one step with the expression e
- $$\begin{array}{l} \langle x, \sigma \rangle \rightarrow \langle \sigma(x), \sigma \rangle \\ \langle n_1 + n_2, \sigma \rangle \rightarrow \langle n, \sigma \rangle \quad \text{where } n = n_1 + n_2 \\ \langle n_1 = n_2, \sigma \rangle \rightarrow \langle \text{true}, \sigma \rangle \quad \text{if } n_1 = n_2 \\ \langle x := n, \sigma \rangle \rightarrow \langle \text{skip}, \sigma[x := n] \rangle \\ \langle \text{skip}; c, \sigma \rangle \rightarrow \langle c, \sigma \rangle \\ \langle \text{if true then } c_1 \text{ else } c_2, \sigma \rangle \rightarrow \langle c_1, \sigma \rangle \\ \langle \text{if false then } c_1 \text{ else } c_2, \sigma \rangle \rightarrow \langle c_2, \sigma \rangle \\ \langle \text{while } b \text{ do } c, \sigma \rangle \rightarrow \langle \text{if } b \text{ then } c; \text{ while } b \text{ do } c \text{ else skip}, \sigma \rangle \end{array}$$

CS 263 Lecture 2

26

The Global Reduction Rule

- General idea of the contextual semantics
 - Identify in the current program the redex to reduce next
 - The program surrounding the redex is called a **context**
 - Reduce the redex "r" to some other expression "e"
 - The resulting expression consists of "e" with the original context
- We use H to range over contexts
- We write $H[r]$ for the expression obtained by placing redex r in context H
- Now we can define a small step

$$\text{If } \langle r, \sigma \rangle \rightarrow \langle e, \sigma' \rangle \text{ then } \langle H[r], \sigma \rangle \rightarrow \langle H[e], \sigma' \rangle$$

CS 263 Lecture 2

27

Contexts

- A context is like an expression (or command) with a marker \bullet in the place where the redex goes
- Examples:
 - To evaluate $\langle (1 + 3) + 2 \rangle$ we use the redex $1 + 3$ and the context $\bullet + 2$
 - To evaluate $\langle \text{if } x > 2 \text{ then } c_1 \text{ else } c_2 \rangle$ we use the redex x and the context $\text{if } \bullet > 2 \text{ then } c_1 \text{ else } c_2$
- Contexts are also called expressions with a hole
- The marker is sometimes called a hole
- $H[r]$ is the expression obtained from H by replacing \bullet with the redex r

CS 263 Lecture 2

28

Contextual Semantics. Example.

- Consider the small-step evaluation of $x := 1; x := x + 1$ in the initial state $[x := 0]$

State	Context	Redex
$\langle x := 1; x := x + 1, [x := 0] \rangle$	$\bullet; x := x + 1$	$x := 1$
$\langle \text{skip}; x := x + 1, [x := 1] \rangle$	\bullet	$\text{skip}; x := x + 1$
$\langle x := x + 1, [x := 1] \rangle$	$x := \bullet + 1$	x
$\langle x := 1 + 1, [x := 1] \rangle$	$x := \bullet$	$1 + 1$
$\langle x := 2, [x := 1] \rangle$	\bullet	$x := 2$
$\langle \text{skip}, [x := 2] \rangle$		

CS 263 Lecture 2

29

Contexts. Notes (I)

- Contexts are defined by a grammar:

$$H ::= \bullet \mid n + H \mid H + e \mid x := H \mid \text{if } H \text{ then } c_1 \text{ else } c_2 \mid H; c$$
- A context has exactly one \bullet marker
- A redex is never a value
- Contexts specify precisely how to find the next redex
 - Consider $e_1 + e_2$ and its decomposition as $H[r]$
 - If e_1 is n_1 and e_2 is n_2 then $H = \bullet$ and $r = n_1 + n_2$
 - If e_1 is n_1 and e_2 is not n_2 then $H = n_1 + H_2$ and $e_2 = H_2[r]$
 - If e_1 is not n_1 then $H = H_1 + e_2$ and $e_1 = H_1[r]$
 - In the last two cases the decomposition is done recursively
 - Check that in each case the solution is unique

CS 263 Lecture 2

30

Contextual Semantics. Notes (II).

- E.g. $c = c_1; c_2$
 - either $c_1 = \text{skip}$ and then $c = H[\text{skip}; c_2]$ with $H = \bullet$
 - or $c_1 \neq \text{skip}$ and then $c_1 = H[r]$; so $c = H'[r]$ with $H' = H; c_2$
- E.g. $c = \text{if } b \text{ then } c_1 \text{ else } c_2$
 - either $b = \text{true}$ or $b = \text{false}$ and then $c = H[r]$ with $H = \bullet$
 - or b is not a value and $b = H[r]$; so $c = H'[r]$ with $H' = \text{if } H \text{ then } c_1 \text{ else } c_2$
- Decomposition theorem:
 - If c is not "skip" then there exist unique H and r such that c is $H[r]$
 - "Exist" means progress
 - "Unique" means determinism

CS 263 Lecture 2

31

Contextual Semantics. Notes.

- Why is it not a good idea to use the following redexes:
 - "while true do c " and "while false do c "
 - along with the context "while H do c " ?
- What if we want to express short-circuit evaluation of \wedge ?
 - Define the following contexts, redexes and local reduction rules
$$H ::= \dots \mid H \wedge b_2$$
$$r ::= \dots \mid \text{true} \wedge b \mid \text{false} \wedge b$$
$$\langle \text{true} \wedge b, \sigma \rangle \rightarrow \langle b, \sigma \rangle$$
$$\langle \text{false} \wedge b, \sigma \rangle \rightarrow \langle \text{false}, \sigma \rangle$$
 - the local reduction kicks in before b_2 is evaluated

CS 263 Lecture 2

32

Contextual Semantics. Notes.

- One can think of the \bullet as representing the program counter
- The advancement rules for \bullet are non trivial
 - At each step the entire command is decomposed
 - This makes contextual semantics inefficient to implement directly
- The major advantage of contextual semantics is that it allows a mix of local and global reduction rules
 - For IMP we have only local reduction rules: only the redex is reduced
 - Later in the course it will be useful to work on the context too

CS 263 Lecture 2

33