

Computer Science 203  
 Programming Languages  
 Fall 2005 - Lecture 2

Cormac Flanagan  
 University of California, Santa Cruz

Due today: Homework 1

A Simple Language of  
 Arithmetic Expressions

syntax and operational semantics

Review of ARITH

- Syntax - what programs look like

$e ::= n$	integer literals
$  e_1 + e_2$	sum
$  e_1 * e_2$	product

- Semantics
  - what programs mean
- Operational Semantics
  - describe program meaning in terms of run-time behavior
  - $e \Downarrow n$

Operational Semantics as Inference Rules

$$\frac{n \Downarrow n}{e_1 \Downarrow n_1 \quad e_2 \Downarrow n_2 \quad n \text{ is the sum of } n_1 \text{ and } n_2} e_1 + e_2 \Downarrow n$$

$$\frac{e_1 \Downarrow n_1 \quad e_2 \Downarrow n_2 \quad n \text{ is the product of } n_1 \text{ and } n_2}{e_1 * e_2 \Downarrow n}$$

- Meaning: "above the line" implies "below the line"
- These rules are
  - *evaluation rules* for the big-step operational semantics
  - *derivation rules* for the judgement  $e \Downarrow n$

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  $e_1 \Downarrow 5$  and  $e_2 \Downarrow 7$ , then we can infer that  $e_1 + e_2 \Downarrow 12$ .

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 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 conclusions of other rules 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. And 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 i.e. an algorithm

## OCAML code for ARITH Operational Semantics

```
type exp =  
  | Int of int  
  | Sum of exp * exp  
  | Prod of exp * exp;;  
  
let anexp = Prod(Sum(Int 3, Int 5), Sum(Int 4, Int 2));;  
  
let rec eval e =  
  match e with  
  | Int n -> n  
  | Sum (e1,e2) -> eval(e1) + eval(e2)  
  | Prod (e1,e2) -> eval(e1) * eval(e2);;  
  
eval anexp;;
```

Lecture 2

7

## Semantic Properties: Uniqueness of Results

- ARITH is a deterministic language: all expressions evaluate to at most one value.  
 $\forall e. \forall n. \forall n'. e \Downarrow n \wedge e \Downarrow n' \Rightarrow n = n'$

Lecture 2

8

## An Imperative Language: IMP

syntax and operational semantics

a little more interesting

Lecture 2

9

Lecture 2

10

## IMP Syntactic Entities

- int integer literals
  - n
- bool booleans
  - p in { true, false }
- L locations (assignable variables)
  - x, y, ...
- Aexp arithmetic expressions
  - e
- Bexp boolean expressions
  - b
- Comm commands
  - c

Lecture 2

11

## Abstract Syntax (Aexp)

- For arithmetic expressions (Aexp)

```
e ::= n      for n ∈ Z  
    | x      for x ∈ L  
    | e1 + e2 for e1, e2 ∈ Aexp  
    | e1 - e2 for e1, e2 ∈ Aexp  
    | e1 * e2 for e1, e2 ∈ Aexp
```

- Notes:
  - Variables are not declared.
  - All variables have integer type.
  - There are no side-effects.

Lecture 2

12

## Abstract Syntax (Bexp)

- For boolean expressions (Bexp)

$b ::=$  true  
 | false  
 |  $e_1 = e_2$  for  $e_1, e_2 \in \text{Aexp}$   
 |  $e_1 < e_2$  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  $b_1, b_2 \in \text{Bexp}$

Lecture 2

13

## Abstract Syntax (Comm)

- For commands (Comm)

$c ::=$  skip  
 |  $x := e$  for  $x \in L$  and  $e \in \text{Aexp}$   
 |  $c_1 ; c_2$  for  $c_1, c_2 \in \text{Comm}$   
 | if  $b$  then  $c_1$  else  $c_2$  for  $c_1, c_2 \in \text{Comm}$  and  $b \in \text{Bexp}$   
 | while  $b$  do  $c$  for  $c \in \text{Comm}$  and  $b \in \text{Bexp}$

- Notes:

- The typing rules have been embedded in the syntax definition.
- Other checks may not be context-free and need to be specified separately (e.g., all variables are declared).
- Commands contain all the side-effects in the language.
- (Missing: pointers, function calls, ...)

Lecture 2

14

## Semantics of IMP

- The meaning of IMP expressions depends on the values of variables.
- A state  $\sigma$  is a function from  $L$  to  $Z$ 
  - Represents the value of variables at a given moment
  - The set of all states is  $\Sigma = L \rightarrow Z$ .

Lecture 2

15

## Operational Semantics of IMP

- Evaluation judgment for expressions
  - A ternary relation: on an expression, a state, and a value.
  - We write:  $\langle e, \sigma \Downarrow n$
  - The evaluation of expressions does not have side-effects, so no resulting state on the right
  - In this case we can also view this judgment as a function of two arguments (written to the left of  $\Downarrow$ )
- Evaluation judgement for commands
  - A ternary relation: on an expression, a state, and a new state.
  - Evaluation of a command has side effects but no direct result
  - The "result" of a Comm is a new state:  $\langle c, \sigma \Downarrow \sigma'$
  - The evaluation of a command might not terminate.

Lecture 2

16

## Evaluation Rules (for Aexp)

$$\frac{}{\langle n, \sigma \Downarrow n}$$

$$\frac{}{\langle x, \sigma \Downarrow \sigma(x)}$$

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

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

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

Lecture 2

17

## Evaluation Rules (for Bexp)

$$\frac{}{\langle \text{true}, \sigma \Downarrow \text{true}}$$

$$\frac{}{\langle \text{false}, \sigma \Downarrow \text{false}}$$

$$\frac{\langle e_1, \sigma \Downarrow n_1 \quad \langle e_2, \sigma \Downarrow n_2 \quad p \text{ is } n_1 = n_2}{\langle e_1 = e_2, \sigma \Downarrow p}$$

$$\frac{\langle e_1, \sigma \Downarrow n_1 \quad \langle e_2, \sigma \Downarrow n_2 \quad p \text{ is } n_1 < n_2}{\langle e_1 < e_2, \sigma \Downarrow p}$$

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

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

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

Lecture 2

18

## Evaluation Rules (for Comm)

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

Lecture 2

19

## Evaluation of Commands: Notes

- The order of evaluation is important and explicit.
  - $c_1$  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  $c_2$ "
- The evaluation rules are not syntax-directed.
  - See the rule for while.
- The evaluation rules do suggest an interpreter.
- Conditional constructs have multiple evaluation rules, but only one can be applied at one time.

- Evaluate from the zero-initialized store:

`while x < 5 do x := x + 1`

Lecture 2

20

## Homework 2 (for Tuesday 4 Oct 05)

- Write OCAML code to implement the operational semantics of IMP
- Some guidelines
  - write types for AExp, BExp, Comm
  - represent variables as strings
  - represent stores as a function from variables to integers
- Handy documentation:
  - <http://caml.inria.fr/pub/docs/manual-ocaml/index.html>

Lecture 2

21

## Homework 2 Template

```

type variable = string;;
type store = (variable -> int);;

let modif (f:store) (y:variable) (v:int) =
  function (x:variable) ->
    if x = y then v else f(x);;

type aexp =
  | Int of int
  | Loc of variable
  | ...;;
type bexp = ...;;
type comm = ...;;

let rec aexpeval (e:aexp) (sigma:store) = ...
let rec bexpeval (b:bexp) (sigma:store) = ...
let rec commeval (c:comm) (sigma:store) = ...

let c = While(Loc "x", Int 5, Set("x", Sum(Loc "x", Int 1)));;
let r = commeval c zero_store;;
r("z");;
r("x");;

```

Lecture 2

22

## Cheating

- All work you turn in must be your own
- If you don't know if something is allowed, please ask
- Any cheating will result in failure of the course and other standard measures
- You are encouraged to discuss course material and assignments with others
- You are not allowed do homeworks with others
- You may use any conversations, texts, or other material, as long as you cite your sources

Lecture 2

23