

Lecture 11 — Evaluating expressions

- Today's class: MP 5
 - Evaluating expressions
 - Subset of MiniJava for MP 5
 - States, values, evaluation
 - “Structured Operational Semantics” (SOS)
 - Statements
- Plan for next few weeks of semester
 - Evaluating expressions
 - Statements; object-oriented programming
 - Type-checking
 - Compilation

Exercise: simple expression evaluation

```
type exp = Operation of exp * binary_operation * exp  
          | Id of string | Integer of int  
and binary_operation = Equal | LessThan | Plus
```

```
type value = Int of int | Bool of bool
```

```
let rec eval e dict =
```

```
and apply bop v1 v2 =
```

Evaluating expressions in MiniJava

- **Abstract syntax of MJ expressions:**

```
type exp = Null | True | False | Integer of int
         | String of string | Id of id | Not of exp
         | Operation of exp * binary_operation * exp
         | MethodCall of exp * id * (exp list)
and id = string
and binary_operation = And | Or | LessThan | Plus | Minus
                    | Multiplication | Division | Equals
```

For MP5, this, new (both objects and arrays), and float and array operations are omitted, and Equals added (equality comparison, not assignment).

- **Will use the same approach in MP5 as on exam problem...**

eval for MJ

```
type value = IntV of int | StringV of string | BoolV of bool | NullV
and state = (varname * value) list
and varname = string
```

```
let rec eval (e:exp) (sigma:state) (prog:program) : value = match e with
```

```
    Null ->
```

```
  | True ->
```

```
  | False ->
```

```
  | Integer i ->
```

```
  | String s ->
```

```
  (* assume id is in state sigma *)
```

```
  | Id id ->
```

applyOp for MJ (cont.)

```
type value = IntV of int | StringV of string | BoolV of bool | NullV
```

```
  | Operation(e1, bop, e2) -> (* for non-boolean operations *)  
    applyOp bop (eval e1 sigma prog) (eval e2 sigma prog)
```

```
let applyOp (bop:binary_operation) (v1:value) (v2:value) : value =  
  match bop with  
  Multiplication ->
```

```
    Plus ->
```

Kinds of errors

● Type errors

- Operations applied to wrong type of value, e.g. `Not 3, if ("abc") . . .`, etc.
- Method call with wrong number of arguments
- Undefined variables
- I.e. errors that would be caught by the compiler in Java or C or C++.

● Run-time, or value, errors

- Subscript out of bounds
- Division by zero

eval for MJ, with exceptions

```
type value = IntV of int | StringV of string | BoolV of bool | NullV  
and state = (varname * value) list  
and varname = string  
exception TypeError of string  
exception RuntimeError of string
```

```
| Id id ->
```

```
| Not e ->
```

Language definitions

- We will give formal definitions in a style called “structured operational semantics” (SOS). SOS describes evaluation of an expression as a function of the evaluation of subexpressions.
- The following notation should be read “expression e evaluates to value v in state σ and program π ”:

$$e, \sigma, \pi \Downarrow v$$

- E.g we can write “Integer $i, \sigma, \pi \Downarrow \text{IntV } i$ ”, meaning: “expression Integer i evaluates to value IntV i , for any i , in any state and program.”
- We often give e in concrete syntax because it looks better.

Ex: SOS for binary operations

$$\begin{array}{l} \text{(BINOPINT)} \quad e_1 + e_2, \sigma, \pi \Downarrow \text{IntV } (i_1 + i_2) \\ \quad e_1, \sigma, \pi \Downarrow \text{IntV } i_1 \\ \quad e_2, \sigma, \pi \Downarrow \text{IntV } i_2 \end{array}$$

$$\text{(BINOPINT)} \quad e_1 * e_2, \sigma, \pi \Downarrow \text{IntV } (i_1 * i_2)$$

$$\text{(LESTHAN)} \quad e_1 < e_2, \sigma, \pi \Downarrow$$

Ex: SOS for boolean operations

(ORTRUE) $e_1 || e_2, \sigma, \pi \Downarrow \text{BoolV true}$
 $e_1, \sigma, \pi \Downarrow \text{BoolV true}$

(ORFALSE) $e_1 || e_2, \sigma, \pi \Downarrow \text{BoolV } t$
 $e_1, \sigma, \pi \Downarrow \text{BoolV false}$
 $e_2, \sigma, \pi \Downarrow \text{BoolV } t$

(ANDFALSE) $e_1 \&\& e_2, \sigma, \pi \Downarrow \text{BoolV false}$

(ANDTRUE) $e_1 \&\& e_2, \sigma, \pi \Downarrow \text{BoolV } t$

(NOT) $!e, \sigma, \pi \Downarrow \text{BoolV (not } b)$

Boolean operations

- Unlike all other operations, `||` and `&&` do not always evaluate both arguments; they are “non-strict.”
- Given SOS rules for `||`:

$$\begin{array}{l} e_1 || e_2, \sigma, \pi \Downarrow \text{BoolV } \text{true} \\ e_1, \sigma, \pi \Downarrow \text{BoolV } \text{true} \end{array}$$

$$\begin{array}{l} e_1 || e_2, \sigma, \pi \Downarrow \text{BoolV } t \\ e_1, \sigma, \pi \Downarrow \text{BoolV } \text{false} \\ e_2, \sigma, \pi \Downarrow \text{BoolV } t \end{array}$$

fill in clause in eval:

| Operation(e1, Or, e2) ->

- Note that the *absence* of rules for `&&` and `||`, when e_1 or e_2 is non-boolean, is significant.

Subset of MJ for MP 5

- MJ programs have the form:

```
class C [extends B] {  
    <field declarations>  
    <method declarations>  
}  
// more classes
```

where method declarations have the form:

```
<type> f (< parameter declarations > ) {  
    <local variable declarations>  
    <statements>  
    return <expression> ;  
}
```

Subset of MJ for MP 5 (cont.)

- For MP 5, there are syntactic restrictions, and also some significant departures from Java semantics.
- Syntactic restrictions:
 - *One* class, which must contain a method named `main`.
 - No fields.
 - Only statements are: assignment (simple and array), if, and block (i.e. statement sequences).
 - Expressions related to objects and arrays — `new C`, `this`, `e1[e2]`, `new C[e]`, `e.length` — are omitted.
 - *Note:* We have left the concrete and abstract syntax alone; we are just ignoring these parts of it (for this week).

Subset of MJ for MP 5 (cont.)

- Semantic differences from Java:
 - No objects or arrays.
 - Type declarations are ignored. (Must be included for syntactic reasons, but have no effect on execution.)
 - *Dynamic typing*: Types are not checked at assignment; meaning of binary operations is determined by type of value, *not* declared type of variables. For example, can write `x = 1; y = x+1; x = "abc"; y = x+1;`. First `+` is integer addition, second is string concatenation.

Statements

- **You will also need to write function `exec`: $\text{statement} \rightarrow \text{state} \rightarrow \text{program} \rightarrow \text{state}$ to execute some simple statements:**

```
statement = Block of (statement list)
           | If of exp * statement * statement
           | Assignment of id * exp
```

```
let rec exec s sigma prog = match s with
  Assignment(s, e) ->
```

```
  | If(e,s1,s2) ->
```

SOS for statements

- Will also use SOS to define exec:
 - “ $s, \sigma, \pi \Rightarrow \sigma'$ ” means that statement s , if it starts in state σ will change it (by assignment statements) to state σ' .
E.g.

$x = 10, [(y,3); (x,4)], \text{program}(\dots) \Rightarrow [(y,3); (x,10)]$

$\{x = 10; y = x\}, [(y,3); (x,4)], \text{program}(\dots)$
 $\Rightarrow [(y,10); (x,10)]$

Ex: SOS rules for statements

(STMT-LIST) $x = e, \sigma, \pi \Rightarrow \sigma$ with v bound to x

(ASSIGN) $\{ S_1; S_2; \dots; S_n \}, \sigma, \pi \Rightarrow \sigma_n$

(IF-TRUE) $\text{if } (e) S_1 \text{ else } S_2, \sigma, \pi \Rightarrow \sigma'$

(IF-FALSE) $\text{if } (e) S_1 \text{ else } S_2, \sigma, \pi \Rightarrow \sigma'$

eval for MJ (cont.)

- We return to expressions to consider the one case we skipped:

```
type value = IntV of int | StringV of string | BoolV of bool | NullV  
and state = (varname * value) list  
and varname = string  
  
| MethodCall(_, m, args) ->
```