

# Lecture 11 — Executing MJ programs

- Once a program has been parsed and transformed to an AST, even without type-checking, we can execute programs by *interpretation*, which involves traversing the AST. For MP6, you will write an interpreter for part of MiniJava, using dynamic typing.
  - Evaluating expressions
  - Executing statements
  - SOS rules for interpretation

# From lecture 1: What you will learn this semester

- How to implement programming languages
  - Writing lexical analyzers and parsers
  - Translating programs to machine language
  - Implementing run-time systems
- How to write programs in a functional programming language
- How to formally define languages (including the definitions of **type rules** and of program execution)
- Key differences between statically-typed languages (e.g. C, Java) and dynamically-typed languages (Python, JavaScript)
- Plus a few other things...

# Grammar for (almost) MiniJava

```
Program -> ClassDeclList
ClassDecl -> class id { VarDeclList MethodDeclList }
VarDecl -> Type id ;
MethodDecl -> Type id ( FormalList ) { VarDeclList StmtList return Exp ; }
Formal -> Type id
Type -> int [ ] | boolean | int | id
Stmt -> { StmtList } | if ( Exp ) Stmt else Stmt
    | while ( Exp ) Stmt | System.out.println ( Exp ) ;
    | id = Exp ; | id [ Exp ] = Exp ;
Exp -> Exp Op Exp | Exp [ Exp ] | Exp . length
    | Exp . id ( ExpList ) | integer | true | false | id
    | this | new int [ Exp ] | new id ( ) | ! Exp | ( Exp )
Op -> && | < | <= | == | + | - | *
ExpList -> Exp ExpRest |
ExpRest -> , Exp ExpRest |
FormalList -> Type id FormalRest |
FormalRest -> , Type id FormalRest |
ClassDeclList = ClassDeclList VarDecl |
MethodDeclList = MethodDeclList MethodDecl |
VarDeclList = VarDeclList VarDecl |
StmtList = StmtList Stmt |
```

# 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 =
```

*(see spring 2012  
midterm 1  
4b answer)*

# 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 MP6, this, new (both objects and arrays), and float and array operations are omitted.**

# 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 -> NullV
| True -> BoolV true
| False -> BoolV false
| Integer i -> IntV i
| String s -> StringV s
(* assume id is in state sigma *)
| Id id -> fetch id sigma
```

# 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 -> match (v1, v2) with
```

```
    (IntV i1, IntV i2) -> IntV (i1 * i2)
```

```
  Plus ->
```

```
    match (v1, v2) with
```

```
      (IntV i1, IntV i2) -> IntV (i1 + i2)
```

```
      (StringV s1, StringV s2) -> StringV (s1 ^ s2)
```

# Kinds of errors

- **Type errors, i.e. errors that would be caught by the Java compiler.**
  - **Operations applied to wrong type of value, e.g. `Not 3, if ("abc") . . .`, etc.**
  - **Method call with wrong number of arguments**
  - **Undefined variables**
- **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 -> if (binds id sigma)
           then (fetch id sigma)
           else (raise TypeError)
| Not e -> match (eval e sigma prog) with
           BoolV b -> BoolV (not b)
           | _ -> raise TypeError
```

# Language definitions

- We will give formal definitions in “structured operational semantics” (SOS), just as we did for type-checking. 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  $\sigma$  and program  $\pi$ ”:

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

- E.g we can write “Integer  $i, \sigma, \pi \Downarrow \text{IntV } i$ ”, meaning: “expression Integer  $i$  evaluates to value  $\text{IntV } i$ , for any  $i$ , in any state and program.”
- In MP6,  $e$  will be an AST, but in the rules we use concrete syntax because it looks better.

# Ex: SOS for binary operations

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

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

$e_1, \sigma, \pi \Downarrow \text{IntV } i_1$

$e_2, \sigma, \pi \Downarrow \text{IntV } i_2$

(LESSTHAN)  $e_1 < e_2, \sigma, \pi \Downarrow \text{BoolV } (i_1 < i_2)$

$e_1, \sigma, \pi \Downarrow \text{IntV } i_1$

$e_2, \sigma, \pi \Downarrow \text{IntV } i_2$

# 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) -> match (eval e1 sigma prog) with  
  BoolV true -> BoolV true

| BoolV false -> match (eval e2 sigma prog) with  
  BoolV b -> BoolV b | \_ -> raise TypeError

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

| \_ -> raise TypeError

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

*see MP6 spec*

# Subset of MJ for MP 6

- 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 6 (cont.)

- For MP 6, 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 6 (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: statement → state → program → 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) ->
```

*asgn s (eval e sigma prog) sigma*

*| If(e, s1, s2) -> match (eval e sigma prog) with*

*Bool V b -> if b then eval stmt s1 sigma prog  
else eval stmt s2 sigma prog*

*| \_ -> raise TypeError*

# SOS for statements

- Will also use SOS to define `exec`:

- “ $s, \sigma, \pi \Rightarrow \sigma'$ ” means that statement  $s$ , if it starts in state  $\sigma$  will change it (by assignment statements) to state  $\sigma'$ .  
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)  $\text{if } (e) S_1 \text{ else } S_2, \sigma, \pi \Rightarrow \sigma'$

see MP6  
spec

## 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) ->
```

- ① lookup  $m$
- ② evaluate args using original sigma yielding  $argVals$
- ③ Create state matching  $argVals$  with variable list from  $m$ , yielding  $\sigma$
- ④ Execute the body of  $m$  starting in  $\sigma$ , yielding  $\sigma'$
- ⑤ Evaluate return expr of  $m$  in  $\sigma'$ ; return this  $value$

# Wrap-up

- Today we discussed:
  - “Interpretation” — executing a program by traversing its AST
  - Specifying how to interpret programs by giving SOS rules
- We discussed it because:
  - Understanding interpretation is a big step toward understanding dynamically-typed languages. It is also good preparation for compilation.
- What to do now:
  - **MP6. *Start early!*** This is a hard MP, and has by far the most complex write-up.



