

## 02—ePL: An Overture

---

CS4215: Programming Language Implementation

Martin Henz

January 20, 2012

Generated on Thursday 19 January, 2012, 22:57

- 1 The Syntax of ePL
- 2 Dynamic Semantics of ePL Programs
- 3 Static Semantics for ePL
- 4 A Virtual Machine for **ePL**

- 1 The Syntax of ePL
- 2 Dynamic Semantics of ePL Programs
- 3 Static Semantics for ePL
- 4 A Virtual Machine for ePL

## Syntax of ePL

$$\frac{}{n} \qquad \frac{}{\text{true}} \qquad \frac{}{\text{false}}$$
$$\frac{E}{p_1[E]} \qquad \frac{E_1 \ E_2}{p_2[E_1, E_2]}$$

$$P_1 = \{\backslash\},$$

$$P_2 = \{ |, \&, +, -, *, /, =, >, < \}.$$

## Syntactic Conventions

---

- We can use parentheses in order to group expressions together.
- We use the usual infix and prefix notation for operators. The binary operators are left-associative and the usual precedence rules apply such that  $1 + 2 * 3 > 10 - 4$  stands for  $> [+ [1, * [2, 3]], - [10, 4]]$

## Examples

---

42

-15 \* (7 + 2)

\ false & true | false

17 < 20 - 4 & 10 = 4 + 11

## Implementation of Syntax

### Lexer/Scanner

Language syntax is typically implemented in a two-step process:  
(1) lexical analysis, (2) parsing

### Lexical analysis

Split program string into sequence of tokens

### Parsing

Construct a tree-like data structure that corresponds to the structure of the program

# Syntax of ePL Programs

---

```
public interface Expression {}
```



## Syntax of ePL Programs

---

```
public class BoolConstant  
  implements Expression {  
    public String value;  
    public BoolConstant(String v) {  
      value = v;  
    }  
  }
```

## Syntax of ePL Programs

---

```
public class BinaryPrimitiveApplication
  implements Expression {
  public String operator;
  public Expression argument1, argument2;
  public BinaryPrimitiveApplication(String op,
    Expression a1, Expression a2) {
    operator = op;
    argument1 = a1;
    argument2 = a2;
  }
}
```

- 1 The Syntax of ePL
- 2 Dynamic Semantics of ePL Programs
- 3 Static Semantics for ePL
- 4 A Virtual Machine for **ePL**

# Values

---

Goal of evaluating an expression is to reach a *value*, an expression that cannot be further evaluated.

In ePL, a value is either an integer constant, or a boolean constant.

In the following rules, we denote values by  $v$ .

## Contraction

---

$$\frac{}{p_1[v_1] >_{\text{ePL}} v} \text{[OpVals}_1\text{]}$$

$$\frac{}{p_2[v_1, v_2] >_{\text{ePL}} v} \text{[OpVals}_2\text{]}$$

One instance of the second rule is:

$$\frac{}{+[1, 1] >_{\text{ePL}} 2}$$

## One-Step Evaluation

$$\frac{E >_{\text{ePL}} E'}{E \mapsto_{\text{ePL}} E'} \text{[Contraction]} \quad \frac{E \mapsto_{\text{ePL}} E'}{p_1[E] \mapsto_{\text{ePL}} p_1[E']} \text{[OpArg}_1]$$

$$\frac{E_1 \mapsto_{\text{ePL}} E'_1}{p_2[E_1, E_2] \mapsto_{\text{ePL}} p_2[E'_1, E_2]} \text{[OpArg}_2]$$

$$\frac{E_2 \mapsto_{\text{ePL}} E'_2}{p_2[E_1, E_2] \mapsto_{\text{ePL}} p_2[E_1, E'_2]} \text{[OpArg}_3]$$

## Evaluation Order

---

One-step evaluation does not prescribe the order in which a given ePL expression is evaluated. Both of the following statements hold:

$$3 * 2 + 4 * 5 \mapsto_{\text{ePL}} 3 * 2 + 20$$

$$3 * 2 + 4 * 5 \mapsto_{\text{ePL}} 6 + 4 * 5$$

## Evaluation

Evaluation is the reflexive transitive closure of one-step evaluation.

$$\frac{E_1 \mapsto_{\text{ePL}} E_2}{E_1 \mapsto_{\text{ePL}^*} E_2} [\mapsto_{\text{ePL}^*_B}]$$

$$\frac{}{E \mapsto_{\text{ePL}^*} E} [\mapsto_{\text{ePL}^*_R}]$$

$$\frac{E_1 \mapsto_{\text{ePL}^*} E_2 \quad E_2 \mapsto_{\text{ePL}^*} E_3}{E_1 \mapsto_{\text{ePL}^*} E_3} [\mapsto_{\text{ePL}^*_T}]$$



## Implementation

Idea

Keep reducing expression until it is not reducible any longer

Code snippet

```
static public Expression  
  evaluate(Expression exp){  
    return reducible(exp) ?  
      evaluate(oneStep(exp))  
    : exp;  
}
```

- 1 The Syntax of ePL
- 2 Dynamic Semantics of ePL Programs
- 3 Static Semantics for ePL**
- 4 A Virtual Machine for ePL

## A Type System for ePL

---

Not all expressions in ePL make sense. For example,

`true + 1`

does not make sense, because `true` is a boolean expression, whereas the operator `+` to which `true` is passed as first argument, expects integers as arguments.

## Typing Relation

---

The set of well-typed expressions is defined by the binary *typing relation*

$$“ : ” : \text{ePL} \rightarrow \{\text{int}, \text{bool}\}$$

We use infix notation for “:”, writing  $E : t$ , which is read as “the expression  $E$  has type  $t$ .”

## Examples

---

- $1+2 : \text{int}$
- $\text{false} \ \& \ \text{true} : \text{bool}$
- $10 < 17-8 : \text{bool}$

but:

- $\text{true} + 1 \ 1 : t$
- $3 + 1 * 5 : \text{bool}$

do **not** hold for any  $t$

## Typing Relation

$$\frac{}{n : \text{int}} \text{[NumT]}$$

$$\frac{}{\text{true} : \text{bool}} \text{[TrueT]}$$

$$\frac{}{\text{false} : \text{bool}} \text{[FalseT]}$$

$$\frac{E : \text{bool}}{\backslash[E] : \text{bool}} \text{[PrimT}_1\text{]}$$

$$\frac{E_1 : t_1 \quad E_2 : t_2}{p[E_1, E_2] : t} \text{[PrimT}_2\text{]}$$

## Types of Primitive Operations

$p$	$t_1$	$t_2$	$t$
+	int	int	int
-	int	int	int
*	int	int	int
/	int	int	int
&	bool	bool	bool
	bool	bool	bool
\	bool		bool
=	int	int	bool
<	int	int	bool
>	int	int	bool

## Well-typed Expressions

---

An expression  $E$  is well-typed, if there is a type  $t$  such that  $E : t$ .



## A Proof

---

————— [NumT]      ————— [NumT]  
2 : int                      3 : int

————— [PrimT]      ————— [NumT]  
2\*3 : int                      7 : int

---

2\*3>7 : bool

## Implementation

### Idea

Compute the type of an expression *bottom-up*, starting from the constants at the leaves of the syntax tree

### Checking

At each node, check that the components have the right type wrt the operator

# Type Safety

---

Type safety is a property of a given language with a given static and dynamic semantics. In a type-safe language, certain problems are guaranteed not to occur at runtime for well-typed programs.

Problems:

- No progress,
- No preservation

- 1 The Syntax of ePL
- 2 Dynamic Semantics of ePL Programs
- 3 Static Semantics for ePL
- 4 A Virtual Machine for ePL**

# Motivation

---

- How do we remember intermediate results?
- How do we know what to do next?

Idea: Translate ePL to a machine language. Execute machine code using an emulator.

## Definition of eVML

---

$$\frac{}{\text{DONE}}$$
$$\frac{s}{\text{LDCI } i . s}$$
$$\frac{s}{\text{LDCB } b . s}$$

## Definition of eVML (cont'd)

$\frac{s}{\text{PLUS}.s}$	$\frac{s}{\text{MINUS}.s}$	$\frac{s}{\text{TIMES}.s}$	$\frac{s}{\text{DIV}.s}$
$\frac{s}{\text{AND}.s}$	$\frac{s}{\text{OR}.s}$		$\frac{s}{\text{NOT}.s}$
$\frac{s}{\text{LT}.s}$	$\frac{s}{\text{GT}.s}$		$\frac{s}{\text{EQ}.s}$

## Example

---

The instruction sequence

[LDCI 1, LDCI 2, PLUS, DONE]

represents a valid eVML program.



# Compiling ePL to eVML

$\text{ : ePL} \rightarrow \text{eVML}$

$E \hookrightarrow s$

---

$Es.DONE$

## Compiling **ePL** to eVML (cont'd)

---

---

$n \hookrightarrow \text{LDCI } n$

---

$\text{true} \hookrightarrow \text{LDCB true}$

---

$\text{false} \hookrightarrow \text{LDCB false}$

## Compiling ePL to eVML (cont'd)

$$E_1 \hookrightarrow s_1 \quad E_2 \hookrightarrow s_2$$

---

$$E_1 + E_2 \hookrightarrow s_1.s_2.PLUS$$

$$E_1 \hookrightarrow s_1 \quad E_2 \hookrightarrow s_2$$

---

$$E_1 / E_2 \hookrightarrow s_1.s_2.DIV$$

...

$$E_1 \hookrightarrow s_1 \quad E_2 \hookrightarrow s_2$$

---

$$E_1 * E_2 \hookrightarrow s_1.s_2.TIMES$$

## Examples

---

$(1 + 2) * 3$

[LDCI 1, LDCI 2, PLUS, LDCI 3, TIMES, DONE]

$1 + (2 * 3)$

[LDCI 1, LDCI 2, LDCI 3, TIMES, PLUS, DONE].

## Executing eVML Code

---

Registers:

- $pc$ : program counter,
- $os$ : operand stack

## Example

---

$$pc = 2$$

$$s = [\text{LDCI } 1, \text{LDCI } 2, \text{PLUS}, \text{LDCI } 3, \text{TIMES}, \text{DONE}]$$

$$s(pc) = \text{PLUS}$$

## Transition Function

---

$$s(pc) = \text{LDCI } i$$

---

$$(os, pc) \Rightarrow_s (i.os, pc + 1)$$

$$s(pc) = \text{LDCB } b$$

---

$$(os, pc) \Rightarrow_s (b.os, pc + 1)$$

## Transition Function (cont'd)

$$s(pc) = \text{PLUS}$$

---

$$(i_2.i_1.os, pc) \Rightarrow_s (i_1 + i_2.os, pc + 1)$$



## End Configuration

---

$$s(pc) = \text{DONE}$$

The result of the computation can be found on top of the operand stack of the end configuration.

$$R(M_s) = v, \text{ where } (\langle \rangle, 0) \Rightarrow_s^* (\langle v.os \rangle, pc), \text{ and } s(pc) = \text{DONE}$$

## Example

---

[LDCI 10, LDCI 20, PLUS, LDCI 6, TIMES, DONE]

$(\langle \rangle, 0) \Rightarrow (\langle 10 \rangle, 1) \Rightarrow (\langle 20, 10 \rangle, 2) \Rightarrow$   
 $(\langle 30 \rangle, 3) \Rightarrow (\langle 6, 30 \rangle, 4) \Rightarrow (\langle 180 \rangle, 5)$

# Implementation

---

## Registers

Keep registers in local variables

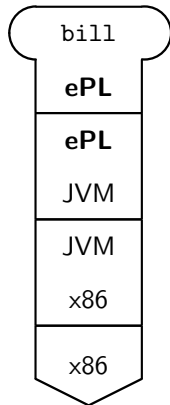
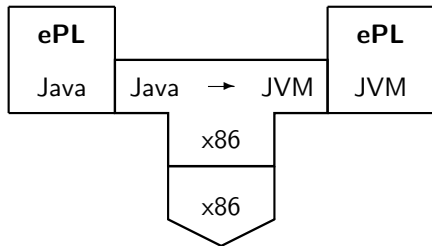
## Execution

Use a switch statement to interpret each instruction

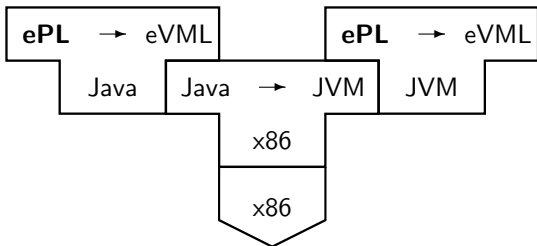
## Implementation

```
public static Value run(INSTRUCTION[]  
                        instructionArray) {  
    int pc = 0;  
    Stack <Value> os = new Stack <Value> ();  
    loop:  
    while (true) {  
        INSTRUCTION i = instructionArray[pc];  
        switch (i.OPCODE) {  
            case OPCODES.LDCI: os.push(  
                                new IntValue(Integer.  
                                    parseInt(i.VALUE));  
                                pc++;  
                                break;  
            ... } } }  
    ... } } }
```

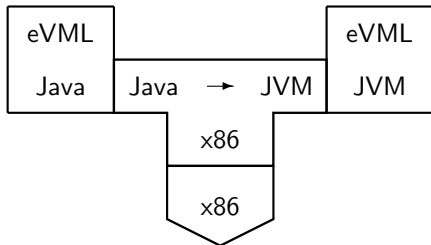
## Virtual Machine as Interpreter



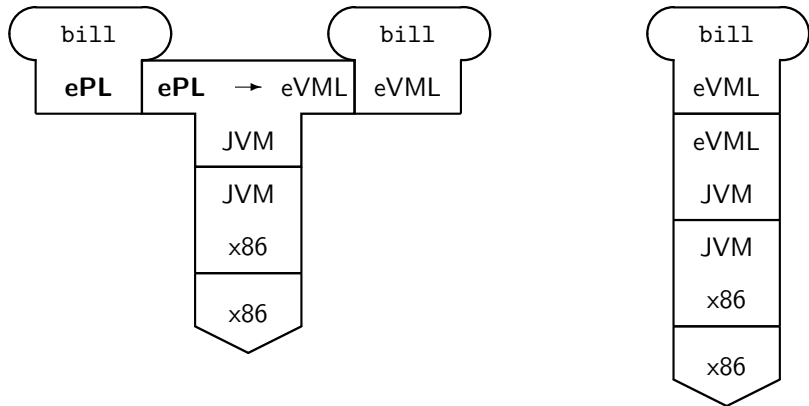
## Compiling the ePL Compiler



## Compiling the eVML Emulator



## Compilation and Execution of Example Program





## Next Week

---

- Introduction of simPL