# 02—ePL: An Overture

CS4215: Programming Language Implementation

Martin Henz

January 20, 2012

Generated on Thursday 19 January, 2012, 22:56

# Syntax of ePL

$$\frac{}{n} \qquad \frac{}{\texttt{true}} \qquad \frac{}{\texttt{false}}$$

$$\frac{E}{p_1[E]} \qquad \frac{E_1 \quad E_2}{p_2[E_1, E_2]}$$

$P_1 = \{\backslash\}$,
$P_2 = \{|,\&,+,-,*,/,=,>,<\}$.

# 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;
   }
```

# 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

$$\frac{}{p_1[v_1] >_{\mathrm{ePL}} v}[\mathrm{OpVals}_1] \qquad \frac{}{p_2[v_1, v_2] >_{\mathrm{ePL}} v}[\mathrm{OpVals}_2]$$

One instance of the second rule is:

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

## One-Step Evaluation

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

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

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

# Evaluation Order

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

$$\texttt{3 * 2 + 4 * 5} \mapsto_{\mathrm{ePL}} \texttt{3 * 2 + 20}$$
$$\texttt{3 * 2 + 4 * 5} \mapsto_{\mathrm{ePL}} \texttt{6 + 4 * 5}$$

# Evaluation

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

$$\frac{E_1 \quad \mapsto_{\text{ePL}} \quad E_2}{E_1 \mapsto_{\text{ePL}}{}^{*} \; E_2}[\mapsto_{\text{ePL}}{}^{*}_{B}] \qquad \frac{}{E \mapsto_{\text{ePL}}{}^{*} \; E}[\mapsto_{\text{ePL}}{}^{*}_{R}]$$

$$\frac{E_1 \mapsto_{\text{ePL}}{}^{*} \; E_2 \qquad E_2 \mapsto_{\text{ePL}}{}^{*} \; E_3}{E_1 \mapsto_{\text{ePL}}{}^{*} \; E_3}[\mapsto_{\text{ePL}}{}^{*}_{T}]$$

# 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;
}
```

# 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*

$$\text{“}:\text{”} : \mathrm{ePL} \to \{\mathtt{int}, \mathtt{bool}\}$$

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

# Examples

- `1+2 : int`
- `false & true : bool`
- `10 < 17-8 : bool`

but:

- `true + 1 1` : $t$
- `3 + 1 * 5` : bool

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

# Typing Relation

$$\frac{}{n : \texttt{int}} \textbf{[NumT]}$$

$$\frac{}{\texttt{true} : \texttt{bool}} \textbf{[TrueT]} \qquad \frac{}{\texttt{false} : \texttt{bool}} \textbf{[FalseT]}$$

$$\frac{E : \texttt{bool}}{\backslash[E] : \texttt{bool}} \textbf{[PrimT}_1\textbf{]} \qquad \frac{E_1 : t_1 \qquad E_2 : t_2}{p[E_1, E_2] : t} \textbf{[PrimT}_2\textbf{]}$$

# 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

$$\dfrac{\dfrac{\dfrac{}{2 : \texttt{int}}[\textbf{NumT}] \qquad \dfrac{}{3 : \texttt{int}}[\textbf{NumT}]}{\texttt{2*3} : \texttt{int}}[\textbf{PrimT}] \qquad \dfrac{}{7 : \texttt{int}}[\textbf{NumT}}{\texttt{2*3>7} : \texttt{bool}}$$

# 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

# 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

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

# Definition of eVML (cont'd)

$$\frac{s}{\texttt{PLUS}.s} \qquad \frac{s}{\texttt{MINUS}.s} \qquad \frac{s}{\texttt{TIMES}.s} \qquad \frac{s}{\texttt{DIV}.s}$$

$$\frac{s}{\texttt{AND}.s} \qquad \frac{s}{\texttt{OR}.s} \qquad \frac{s}{\texttt{NOT}.s}$$

$$\frac{s}{\texttt{LT}.s} \qquad \frac{s}{\texttt{GT}.s} \qquad \frac{s}{\texttt{EQ}.s}$$

## Example

The instruction sequence

$$[\texttt{LDCI } 1, \texttt{LDCI } 2, \texttt{PLUS}, \texttt{DONE}]$$

represents a valid eVML program.

# Compiling **ePL** to eVML

$$: \textbf{ePL} \to \text{eVML}$$

$$\frac{E \hookrightarrow s}{E s.\texttt{DONE}}$$

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

$$\frac{}{n \hookrightarrow \texttt{LDCI}\ n}$$

$$\frac{}{\texttt{true} \hookrightarrow \texttt{LDCB true}}$$

$$\frac{}{\texttt{false} \hookrightarrow \texttt{LDCB false}}$$

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

$$\frac{E_1 \hookrightarrow s_1 \qquad E_2 \hookrightarrow s_2}{E_1 + E_2 \hookrightarrow s_1.s_2.\texttt{PLUS}}$$

$$\frac{E_1 \hookrightarrow s_1 \qquad E_2 \hookrightarrow s_2}{E_1 * E_2 \hookrightarrow s_1.s_2.\texttt{TIMES}}$$

$$\frac{E_1 \hookrightarrow s_1 \qquad E_2 \hookrightarrow s_2}{E_1 / E_2 \hookrightarrow s_1.s_2.\texttt{DIV}}$$

...

# 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

$pc = 2$

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

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

# Transition Function

$$\frac{s(pc) = \texttt{LDCI}\ i}{(os, pc) \rightrightarrows_s (i.os, pc + 1)}$$

$$\frac{s(pc) = \texttt{LDCB}\ b}{(os, pc) \rightrightarrows_s (b.os, pc + 1)}$$

# Transition Function (cont'd)

$$\frac{s(pc) = \mathrm{PLUS}}{(i_2.i_1.os, pc) \rightrightarrows_s (i_1 + i_2.os, pc + 1)}$$

# End Configuration

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

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

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

# Example

$$[\mathtt{LDCI\ 10}, \mathtt{LDCI\ 20}, \mathtt{PLUS}, \mathtt{LDCI\ 6}, \mathtt{TIMES}, \mathtt{DONE}]$$

$$(\langle\rangle, 0) \rightrightarrows (\langle 10\rangle, 1) \rightrightarrows (\langle 20, 10\rangle, 2) \rightrightarrows$$

$$(\langle 30\rangle, 3) \rightrightarrows (\langle 6, 30\rangle, 4) \rightrightarrows (\langle 180\rangle, 5)$$

# Implementation

## Registers

Keep registers in local variables

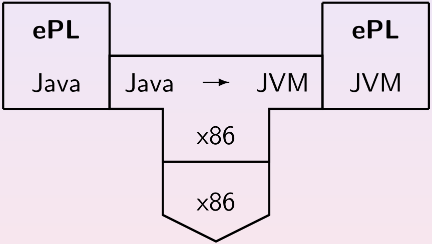
## Execution

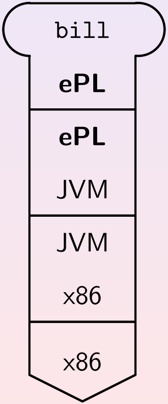
Use a switch statement to interpret each instruction

# Implementation

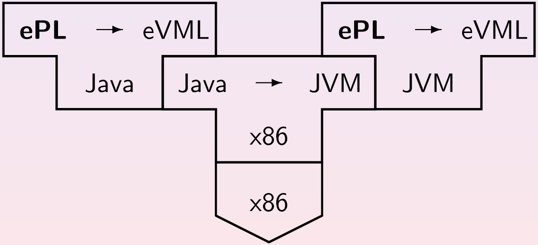
```
public static Value run(INSTRUCTION[]
                            instructionArray) {
  int pc = 0;
  Stack <Value> os = new Stack <Value> ();
  loop:
  while (true) {
    INSTRUCTION i = instructionArray[pc];
    switch (i.OPCODE) {
      case OPCODES.LDCI: os.push(
                           new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

# Virtual Machine as Interpreter

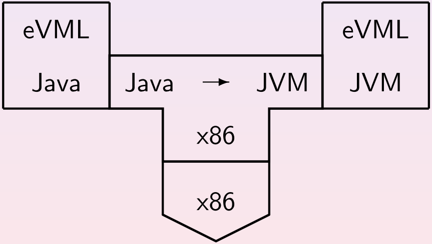


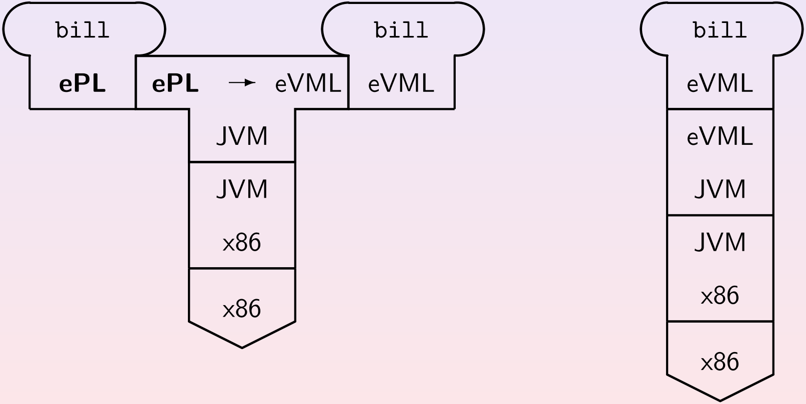
# Compiling the ePL Compiler



# Compiling the eVML Emulator



# Compilation and Execution of Example Program



# Next Week

- Introduction of simPL