ENS DE LYON

Lyon 1

CAP, ENSL, 2024/2025

**Homework (DM)**
**Compilation and Program Analysis (CAP)**

# From Erlang to Lisp

Instructions:

1. Every single answer must be informally explained AND formally proved.
2. Using LaTeX is NOT mandatory at all.
3. Vous avez le droit de rédiger en Français.

---

In this Homework, we consider an alternative reality: LISP has won! More precisely, LISP machines[1] have become the primary architecture for desktop computers. In this alternative reality, C never become the "langua franca" of computer programs. Instead, LISP took its place as the "portable assembly" of programming languages: a prime compilation target for higher level programming languages! Furthermore, recent LISP machines even feature new capabilities such as concurrency and parallelism.

Beside this overwhelming success by functional programmers for world domination, a few pockets of resistance remains in academic communities to develop object oriented programming. In this homework, we wish to develop **Erlisp** a promising language for distributed computations where each processes is an object which can communicate asynchronously via message passing. We wish to compile this language to **AssembLISP** for the modern Intel ML86_64 (for **MultiLisp**[2]) architecture supporting asynchronous computations !

We will first define **AssembLISP** and study some of its properties, before moving to **Erlisp**.

[1]https://en.wikipedia.org/wiki/Lisp_machine
[2]https://en.wikipedia.org/wiki/MultiLisp

## 1 AssembLISP: an imperative lambda calculus with asynchronous computations

We first define the syntax of **AssembLISP** in Figure 1. It contains the constructs of the lambda calculus with lambda terms that define functions, function application and variables. It is an imperative lambda calculus with creation of cells identified by their location; writing in cells, i.e. assignment; and reading, i.e. dereferencing. The language also has a conditional statement. The three next constructs, i.e. spawn, launch, and get, deal with concurrency and will be described later. The last line contains elements that belong to expressions because they can appear at runtime but never in the source program, we will describe them along with the semantics

$$
\begin{array}{rll}
e \in Expr ::= & () \mid tt \mid ff \mid n & \text{Constants (unit, booleans, integers)} \\
 & \mid e_1 + e_2 \mid e_1 < e_2 \mid e_1 \wedge e_2 \mid \cdots & \text{int and bool operations} \\
 & \mid \mathbf{x} & \text{Variables} \\
 & \mid \lambda \mathbf{x}.e & \text{Functions} \\
 & \mid \texttt{new}\ (e) & \text{New references} \\
 & \mid !e & \text{Dereferencing} \\
 & \mid e := e & \text{Assignment} \\
 & \mid \texttt{if}\ e\ \{\ e_1\ \}\ \texttt{else}\ \{\ e_2\ \} & \text{Conditional} \\
 & \mid e_1\ e_2 & \text{Application} \\
 & \mid \texttt{spawn} & \text{Thread creation} \\
 & \mid \texttt{launch}\ e_1\ \{\ e_2\ \} & \text{Launches a computation} \\
 & \mid \texttt{get}\ e & \text{Retrieves the result} \\
 & \mid l \in \mathbb{L} \mid f \in \mathcal{F} \mid t \in \mathcal{T} & \text{expressions that appear at runtime}
\end{array}
$$

Figure 1: AssembLISP Syntax

### 1.1 Local semantics

Figure 2 defines the runtime syntax and the semantics for **AssembLISP** (without concurrency for the moment). The reduction uses a store that maps cell locations to values and represent the memory. $\mathbb{L}$ denotes the set of locations and $l$ will represent one location in the syntax ($l \in \mathbb{L}$). Among the expressions of the syntax, some of them are values that cannot be evaluated any more; they are denoted with $v$ variables in the semantics and contain not only the constants but also locations and lambda terms. They also contain other runtime elements that will appear later ($f$ and $a$). The store maps locations to values.

The semantics uses evaluation contexts to specify the point in the term at which reduction occurs. An evaluation context is an expression with a single hole denoted $[]$, the notation $C[e]$ denotes the evaluation context $C$ where the hole has been replaced by the expression $e$, it is thus an expression. Evaluation context enforce left-to-right evaluation of expressions, for example, look at the terms for application: there can always be a hole on the left side ($C\ e$), whereas to have a hole on the right side the left side must be evaluated to a value ($v\ C$).

Rule Context of small step semantics reduces under an evaluation context. Rule IfTrue and IfFalse deal with conditionals. The Beta-reduction rule evaluate a function application when its left part is a function and its right part is an evaluated argument. In other words, we define a call-by-value semantics. Evaluation proceeds with the body of the function where the argument

Configurations are

$$c ::= (e, \sigma)$$

Where the store $\sigma$ is:

$$\sigma : \mathbb{L} \mapsto \mathbb{V}$$

Values are:

$$v \in \mathbb{V} ::= () \mid tt \mid f\!f \mid n \mid \lambda \mathbf{x}.e \mid l \mid f \mid a$$

with $l \in \mathbb{L}$
Initial configuration for a program $e$: $(e, \emptyset)$
**Small step relation for an imperative lambda calculus:** $c \to c'$

$$C ::= [] \mid \texttt{if}\ C\ \{\ e_1\ \}\ \texttt{else}\ \{\ e_2\ \} \mid C\ e \mid v\ C \mid !C \mid l := C \mid C := e \mid \texttt{new}\ C \mid \texttt{launch}\ C\ \{\ e_2\ \} \mid \texttt{get}\ C \mid C + e \mid v + C$$

$$\textsc{Context}\ \frac{(e,\sigma) \to (e',\sigma')}{(C[e],\sigma) \to (C[e'],\sigma')} \qquad \textsc{IfTrue}\ \frac{}{(\texttt{if}\ tt\ \{\ e_1\ \}\ \texttt{else}\ \{\ e_2\ \},\sigma) \to (e_1,\sigma)}$$

$$\textsc{IfFalse}\ \frac{}{(\texttt{if}\ f\!f\ \{\ e_1\ \}\ \texttt{else}\ \{\ e_2\ \},\sigma) \to (e_2,\sigma)} \qquad \textsc{Beta}\ \frac{}{((\lambda \mathbf{x}.e)\ v,\sigma) \to (e[v/\mathbf{x}],\sigma)} \qquad \textsc{RefNew}\ \frac{l \notin \mathrm{dom}(\sigma)}{(\texttt{new}\ v,\sigma) \to (l,\sigma[l \mapsto v])}$$

$$\textsc{RefRead}\ \frac{}{(!l,\sigma) \to (\sigma(l),\sigma)} \qquad \textsc{RefWrite}\ \frac{}{(l := v,\sigma) \to ((),\sigma[l \mapsto v])} \qquad \textsc{Add}\ \frac{n = n_1 + n_2}{(n_1 + n_2,\sigma) \to (n,\sigma)}$$

Figure 2: Operational semantics for sequential **AssembLISP**

has been replaced by the parameter passed by value, $e[v/\mathbf{x}]$ denotes this substitution. Rules RefNew, RefRead and RefWrite deal with locations: creating a fresh location, accessing the stored value and finally modifying a stored value. Rule Add demonstrate builtin operations, here the addition. We suppose we have one such rule for all basic unary and binary operations on integers and booleans.

We also use the following `let` construct (that is directly translated to a lambda term):

$$\texttt{let}\ \mathbf{x} = e\ \texttt{in}\ e' \triangleq (\lambda \mathbf{x}.e')\ e$$

**Question #1**

Explain the difference between the store we used in the while language during the course and this one.

**Question #2**

Add a `While` construct to the language: Extend the syntax and define its small step semantics with a dedicated rule. You may need to extend other intermediate constructs, explain.

**Question #3**

Add the sequence `;` to the language: extend the syntax. Give a translation rule that exploit the call-by-value semantics to transform the sequence into a term of the original language

**Question #4**

Write a lambda expression `SUM` that takes an integer $n$ as parameter and sums the $n$ first integers with a while loop. You will need to use an accumulator inside the loop. What construct do you use for it?

**Question #5**

We wish to add tuples to our language. We add the syntax $(e_0, \dots, e_{n-1})$ to create a tuple of size $n$ and $e.k$ to access the $k^{th}$ field.

Give a definition by **translation** for tuples of three elements (construction and access).

In the following we will consider `while`, sequence and arbitrary tuples usable in **AssembLISP** programs (but it will not be necessary to consider them in proofs).

## 1.2 Concurrent AssembLISP

Concurrency is based on the notions of threads and futures, both of them have identifiers: variables $f \in \mathcal{F}$, represent future identifiers, while variables $t \in \mathcal{T}$ represent thread identifiers. A future is a construct used in concurrent programming languages to represent the result of a computation being performed. Figure 3 describes the runtime syntax and the semantics for concurrency with thread, tasks, and futures. :: is the list constructor and $\uplus$ the disjoint union on maps. [] is the empty list.

Configurations are now built with the same store as above, but instead of an expression being evaluated, we have a thread map $\Theta$ and a future map $\Phi$. The thread map maps each thread identifier $t$ to a queue of tasks to be performed by the thread. Each task is an expression to evaluate and a future identifier where to store the result. The thread always evaluates the first element of the list until it can fulfil the future before moving to the next task.

Rules of the operational semantics work as follows. Local performs a local reduction according to Figure 2. Spawn Creates a new thread with an empty tasks queue, and returns the identifier of the created thread. Launch launches a new task, i.e. it creates a fresh future identifier $f'$, adds to the list of tasks of $t'$ a new one that will fulfil $f'$, returns to the caller a reference to the future $f'$, and adds $f'$ to $\Phi$ for the moment mapped to $\varnothing$, meaning the future is still unresolved. Resolve triggers when a task is finished, i.e. its evaluation reduced to a value; it maps $f$ (the future corresponding to the finished task) to the computed value in $\Phi$; the task is removed from the list, meaning the next one can start its execution. Get is used to fetch a future value in $\Phi$; it only reduces if the future has been already resolved, else nothing happens.

To evaluate a program $P$, we pick a thread identifier $t$ and a future identifier $f$ and create a runtime configuration:

$$([t \mapsto (f, P)], [f \mapsto \varnothing], \varnothing)$$

**Question #6**

What is a final configuration if all computations are finished?

Configurations are now: $cc ::= (\Theta, \sigma, \Phi)$
Future map $\Phi$ maps future identifiers $f$ to values or $\varnothing$: $\Phi : \mathcal{F} \mapsto \mathbb{V} \mid \varnothing$
Thread map $\Theta$ maps thread identifiers $t$ to queues of tasks: $\Theta : \mathcal{T} \mapsto (\mathcal{F} \times \text{Expr})\ \texttt{List}$
A task is a pair (future, expressions)

$$\textsc{Local in } t \quad \frac{(e,\sigma) \to (e',\sigma')}{([t \mapsto (f,e) :: el] \uplus \Theta, \sigma, \Phi) \to_{||} ([t \mapsto (f,e') :: el] \uplus \Theta, \sigma', \Phi)}$$

$$\textsc{Spawn in } t \quad \frac{t' \notin \text{dom}(\Theta) \cup \{t\}}{([t \mapsto (f, C[spawn]) :: el] \uplus \Theta, \sigma, \Phi) \to_{||} ([t \mapsto (f, C[t']) :: el] \uplus [t' \mapsto []] \uplus \Theta, \sigma, \Phi)}$$

$$\textsc{Launch in } t \quad \frac{t \neq t' \qquad f' \notin \text{dom}(\Phi)}{\begin{array}{c}([t \mapsto (f, C[\texttt{launch}\ t'\ \{\ e\ \}]) :: el] \uplus [t' \mapsto el'] \uplus \Theta, \sigma, \Phi) \to_{||} \\ ([t \mapsto (f, C[f']) :: el] \uplus [t' \mapsto el' :: (f', e)] \uplus \Theta, \sigma, \Phi[f' \mapsto \varnothing])\end{array}}$$

$$\textsc{Resolve in } t \quad \frac{}{([t \mapsto (f,v) :: el] \uplus \Theta, \sigma, \Phi) \to_{||} ([t \mapsto el] \uplus \Theta, \sigma, \Phi[f \mapsto v])}$$

$$\textsc{Get in } t \quad \frac{\Phi(f') = v \qquad v \neq \varnothing}{([t \mapsto (f, C[\texttt{get}\ f']) :: el] \uplus \Theta, \sigma, \Phi) \to_{||} ([t \mapsto C[v] :: el] \uplus \Theta, \sigma, \Phi)}$$

Figure 3: Operational semantics for concurrent **AssembLISP**: $cc \to_{||} cc'$

**Question #7**

We define a program CONC as follows:

$$\texttt{CONC} \triangleq \begin{array}{l}\texttt{let}\ \mathbf{x} = \texttt{new}\ 0\ \texttt{in}\ \texttt{let}\ \mathbf{t_1} = \texttt{spawn}\ \texttt{in}\ \texttt{let}\ \mathbf{t_2} = \texttt{spawn}\ \texttt{in} \\ \texttt{launch}\ \mathbf{t_1}\ \{\ \mathbf{x} := !\mathbf{x} + 1\ \};\ \texttt{launch}\ \mathbf{t_1}\ \{\ \mathbf{x} := !\mathbf{x} + 2\ \};\ \texttt{launch}\ \mathbf{t_2}\ \{\ \mathbf{x} := !\mathbf{x} + 1\ \}\end{array}$$

What are the possible outcomes of this computation? Sketch the derivation of the semantics for one of these outcomes. Explain what can happen in a couple of lines.

**Question #8**

Here is a simple example program that also uses futures:

$$\texttt{FUTEX} \triangleq (\texttt{get}\ (\texttt{launch}\ \texttt{spawn}\ \{\ 1 + 1\ \})) + 1$$

Derive the semantics for this simple example (you can omit trivial steps).

**Question #9**

Write a term that spawns 2 threads, launches two tasks that compute SUM(5) and SUM(4) on the 2 threads, and retrieves the 2 results to compute and return SUM(5)+SUM(4)

**Question #10**

Write an invariant relating the value of a future in $\Phi$ and its occurrences in the rest of the configuration. There are in fact be three invariants: one corresponding to the thread that computes the value of the future, another one for the expressions that can refer to this future, for this one we use $e \in e'$ to state that the expression $e$ is a sub-expression of $e'$, finally there is one for future identifiers appearing in the store, or in other future values.

We give the structure of the invariant, fill the blanks $\cdots$ as precisely as possible:

For al runtime configurations $(\Theta, \sigma, \Phi)$:

$$\begin{aligned}
(f, e) = \Theta(t) &\implies \Phi(f) = \cdots \\
(f, e) = \Theta(t) \land f' \in e &\implies \cdots \\
\sigma(l) = f \lor \Phi(f') = f &\implies \cdots
\end{aligned}$$

**Question #11 (Difficult)**

We define a program `LOOP` as follows:

$$\texttt{LOOP} \triangleq \begin{array}{l}
\texttt{let } \mathbf{n} = \texttt{new } 1 \texttt{ in} \\
\texttt{let } \mathbf{t} = \texttt{spawn in} \\
\texttt{let } \mathbf{run} = \lambda\mathbf{f}.(!n := 2{*}!n + 1;\ \texttt{launch } \mathbf{t}\ \{\ \mathbf{f}\ \mathbf{t}\ \})\ \texttt{in} \\
\texttt{launch } \mathbf{t}\ \{\ \mathbf{run}\ \mathbf{run}\ \}; \\
!\mathbf{n}
\end{array}$$

During the execution of this program, the thread **t** `launch`es a task to itself. Why is this not possible given the current reduction rules ?

Propose a new reduction rule which allow this program to run fully.

What does the program do with your rule ?

## 2 Erlisp: An object oriented language for distributed computing

We are now ready to model **Erlisp**. Its syntax is shown in Figure 4. Our goal for the reminder will be to compile this language to our target **AssembLISP**.

**Erlisp** is a purely object-oriented language, without any first-class functions. Its syntax is given in Figure 4. Programs are made of a list of object declarations, terminated by a "main" expression. Object can contains attributes, which are mutable variables only accessible in the objects, and methods, which are procedures that can be called externally. Both attributes and methods are defined with expressions containing basic constructions such as operators, `let`, `if` and `while`, along with object-oriented features. Attributes can be accessed like any other variables, or write with $\mathbf{x} \leftarrow e$. Methods can be called with $o.\text{meth}(arg)$, which returns a future containing the result. The content of the future can be retrieved with `get` as previously. A special variable, **self** can be used inside methods to designate the surrounding object.

Here is an example program:

```
collatz = {
  n = 123456789
  run(x) =
    if n = 0 mod 2 then n <- n/2 else n <- 3n+1;
    self.run(x);
    ()
  fetch(x) =
    n
}
let _ = collatz.run(()) in
let n1 = get(collatz.fetch(())) in
let n2 = get(collatz.fetch(())) in
n1 = n2
```

It runs the Collatz series continuously in a thread, but allow fetching the current result at any point. `fetch`ing will return a future containing a value of the series. The program then test if two arbitrary values in the series are equal.

We provide a partial translation from **Erlisp** to **AssembLISP** in Figure 5. The idea of this translation is to associate each object to a specific thread, which will be dedicated to execute the methods of its object. Attributes can only be accessed from inside the object's methods. When called, methods return a future of the result. In this initial version, objects only support one attribute and one method. $\mathcal{P}(\cdot)$ is the compilation function on programs. $\mathcal{E}_a(\cdot)$ is the compilation function on expressions in a context where variable $a$ is an attribute. Rule NewObj translates some object declaration into the appropriate **AssembLISP** code which initializes all the arguments using references, spawn a new thread, and defines all the methods as lambdas. Rule Meth translate a method call to code launching a task executing the method.

**Question #12**

Complete the rule Let and AttrWrite.

**Question #13**

Consider the following program

$$
\begin{array}{rcll}
e \in \mathit{Expr} & ::= & () \mid tt \mid ff \mid n & \text{Constants (unit, booleans, integers)} \\
 & \mid & e_1 + e_2 \mid e_1 < e_2 \mid e_1 \wedge e_2 \mid \cdots & \text{int and bool operations} \\
 & \mid & \texttt{while}\ e\ \{\ e'\ \} & \text{While} \\
 & \mid & \texttt{if}\ e\ \{\ e_1\ \}\ \texttt{else}\ \{\ e_2\ \} & \text{Conditional} \\
 & \mid & \texttt{let}\ \mathbf{x} = e_1\ \texttt{in}\ e_2 & \text{Let Binding} \\
 & \mid & \mathbf{self} & \text{Self} \\
 & \mid & \mathbf{x} & \text{Variable Read} \\
 & \mid & \mathbf{x} \leftarrow e & \text{Variable Write} \\
 & \mid & e.\mathrm{meth}(e) & \text{Asynchronous Method Call} \\
 & \mid & \texttt{get}\ e & \text{Retrieve the result} \\
f \in \mathit{Field} & ::= & \mathbf{x} = e & \text{Attribute Definition} \\
 & \mid & \mathrm{meth}(\mathbf{x}) = e & \text{Method Definition} \\
p \in \mathit{Program} & ::= & e & \text{Main expression} \\
 & \mid & \mathbf{o} = \{\overline{f}\};\ p & \text{Object Declaration}
\end{array}
$$

Figure 4: Syntax of **Erlisp**

$$
\textsc{Get}\quad \frac{}{\mathcal{E}_{\mathbf{a}}(\texttt{get}\ e) = \texttt{get}\ \mathcal{E}_{\mathbf{a}}(e)}
\qquad
\textsc{While}\quad \frac{}{\mathcal{E}_{\mathbf{a}}(\texttt{while}\ e_b\ \{\ e\ \}) = \texttt{while}\ \mathcal{E}_{\mathbf{a}}(e_b)\ \{\ \mathcal{E}_{\mathbf{a}}(e)\ \}}
$$

$$
\textsc{If}\quad \frac{}{\mathcal{E}_{\mathbf{a}}(\texttt{if}\ e\ \{\ e_1\ \}\ \texttt{else}\ \{\ e_2\ \}) = \texttt{if}\ \mathcal{E}_{\mathbf{a}}(e)\ \{\ \mathcal{E}_{\mathbf{a}}(e_1)\ \}\ \texttt{else}\ \{\ \mathcal{E}_{\mathbf{a}}(e_2)\ \}}
$$

$$
\textsc{Let}\quad \frac{\text{TO COMPLETE}}{\mathcal{E}_{\mathbf{a}}(\texttt{let}\ \mathbf{x} = e\ \texttt{in}\ e') = \text{TO COMPLETE}}
\qquad
\textsc{Op}\quad \frac{}{\mathcal{E}_{\mathbf{a}}(e_1 \oplus e_2) = \mathcal{E}_{\mathbf{a}}(e_1) \oplus \mathcal{E}_{\mathbf{a}}(e_2)}
\qquad
\textsc{Var}\quad \frac{\mathbf{x} \neq \mathbf{a}}{\mathcal{E}_{\mathbf{a}}(\mathbf{x}) = \mathbf{x}}
$$

$$
\textsc{AttrRead}\quad \frac{}{\mathcal{E}_{\mathbf{a}}(\mathbf{a}) = !\mathbf{a}}
\qquad
\textsc{AttrWrite}\quad \frac{}{\mathcal{E}_{\mathbf{a}}(\mathbf{z} \leftarrow \mathbf{e}) = \text{TO COMPLETE}}
$$

$$
\textsc{Meth}\quad \frac{\mathbf{x}_{\mathbf{self}} \text{ and } \mathbf{x}_{arg} \text{ fresh in } \mathit{FreeVariables}(e) \cup \{a\}}{\mathcal{E}_{\mathbf{a}}(e.\mathrm{meth}(e')) = \begin{array}{l}\texttt{let}\ \mathbf{x}_{\mathbf{self}} = \mathcal{E}_{\mathbf{a}}(e)\ \texttt{in} \\ \texttt{let}\ \mathbf{x}_{arg} = \mathcal{E}_{\mathbf{a}}(e')\ \texttt{in} \\ \texttt{launch}\ \mathbf{x}_{\mathbf{self}}.0\ \{\ \mathbf{x}_{\mathbf{self}}.1\ \mathbf{x}_{\mathbf{self}}\ \mathbf{x}_{arg}\ \}\end{array}}
$$

$$
\textsc{NewObj}\quad \frac{e' = \mathcal{E}_{\emptyset}(e) \qquad e'_m = \mathcal{E}_{\mathbf{a}}(e_m) \qquad \mathbf{x} \neq \mathbf{a}}{\mathcal{P}(\mathbf{o} = \{\mathbf{a} = e;\ \mathrm{meth}(\mathbf{x}) = e_m\}; p) = \begin{array}{l}\texttt{let}\ \mathbf{o}\ = \\ \quad \texttt{let}\ \mathbf{a} = \texttt{new}\ e'\ \texttt{in} \\ \quad (\texttt{spawn},\ \lambda\mathbf{self}.\lambda\mathbf{x}.e'_m) \\ \texttt{in}\ \mathcal{P}(p)\end{array}}
\qquad
\textsc{Main}\quad \frac{}{\mathcal{P}(e) = \mathcal{E}_{\emptyset}(e)}
$$

Figure 5: Compilation of **Erlisp** to **AssembLISP**

```
Counter = {
  state = 0
  add(x) = state <- state + x; state
};
let x = Counter.add(10) in
let y = Counter.add(15) in
get(x) + get(y)
```

Compile this program and execute it (you can skip trivial steps).

**Question #14**

Here is an alternative rule for METH. Why is it incorrect ? Illustrate with examples.

$$\textsc{Meth-Incorrect}\quad \frac{e_0' = \mathcal{E}_{\mathbf{a}}(e_0) \qquad e_1' = \mathcal{E}_{\mathbf{a}}(e_1')}{\mathcal{E}_{\mathbf{a}}(e_0.\text{meth}(e_1)) = \texttt{launch}\ e_0'.0\ \{\ e_0'.1\ e_0'\ e_1'\ \}}$$

**Question #15**

We now wish to support an arbitrary number of attributes. Expand the rules NEWOBJ, ATTR-READ and ATTRWRITE for this purpose (use n-ary tuples).

## 2.1 Multiple methods with dynamic dispatch

We now wish to support multiple methods.

**Question #16**

Write an **Erlisp** object with a counter attribute and three methods with unit arguments: **incr**, which increments the counter and returns the new value, **decr**, which decrements the counter and returns the new value, and **reset**, which resets the counter to zero and returns unit.

To help identify which method has been called in objects with multiple methods, we require a new feature in our **AssembLISP**. Symbols are simple labels that start with an uptick (**'symbol**). They are values whose only available operation is the equality test.

$$e \in Expr ::= \cdots \mid \textbf{'symbol} \qquad\qquad v \in \mathbb{V} ::= \cdots \mid \textbf{'symbol}$$

Figure 6: AssembLISP extended with Symbols

**Question #17**

Write an **AssembLISP** program with a function **counter_method** taking as argument a label among **'incr**, **'decr** and **'reset** and implementing the same counter semantics as the previous question.

**Question #18**

Extend NEWOBJ and METH to support multiple methods in an object. You can use "..." for part of the rules that are identical to previous answers.

## 2.2 Encapsulation

**Question #19**

Let us consider the following object definition:

```
pass_checker = {
  secret = 42
  validate(s) = (s = secret)
}
```

Can the **secret** attribute be written or read by any other object ? What part of the translation ensures that?

**Question #20**

We are interested in formalizing this encapsulation property. For this purpose, we consider what happens if the memory is filled with a new inert value, $\perp$, except for the object being executed.

Given a program **Erlisp** $p$, We consider an execution of its compiled program $s_0 = \mathcal{P}(p)$. Let **o** an object definition in $p$, We denote $t_o$ its thread (resulting from the `spawn` instruction) and $A_o$ the set of memory location containing its attributes during the execution of $s_0$.

Let $\Theta, \sigma_1, \sigma_1', \Phi, \Phi_1', el, el_1'$ such that

$$([t_o \mapsto el] \uplus \Theta, \sigma_1, \Phi) \rightarrow_{||} ([t_o \mapsto el_1'] \uplus \Theta_1', \sigma_1', \Phi_1')$$

where $t_o$ is the thread in which the reduction is performed, i.e. the rule applied is of the form ??? $\textsc{in}$ $t_o$ (e.g. LOC in $t_o$)

Let $\sigma_2$ such that $\sigma_2(\ell) = \begin{cases} \sigma_1(\ell) & \text{if } \ell \in A_o \\ \perp & \text{otherwise} \end{cases}$.

Then, there exists $\sigma_2', \Phi_2', el_2'$ such that

$$([t_o \mapsto el] \uplus \Theta, \sigma_2, \Phi) \rightarrow_{||} ([t_o \mapsto el_2'] \uplus \Theta_2', \sigma_2', \Phi_2')$$

We try to relate $\Theta_2'$, $\sigma_2', \Phi_2', el_2'$ with what happens in the first reduction that leads to $el_1'$.

(a) In each of the following 3 cases relate $\Theta_2'$, $\sigma_2', \Phi_2', el_2'$ with the other variables and explain informally what happens.

1. the reduction corresponds to a method call;
2. the reduction corresponds to a **get**;
3. the reduction corresponds to an read or write to an attribute.

(b) Give the complete encapsulation property that characterizes $\Theta_2'$, $\sigma_2', \Phi_2', el_2'$ independently of the reduction applied.

**Question #21 (Difficult)**

Where is evaluated the code computing the initial value of attributes? To perfect our encapsulation, we wish to compute this code in the object's thread as well. Adjust the compilation and state explicitly which instructions are now executed in the object thread.