# Lazy Lists in L1++: Semantics and Implementation

Programming Languages Project - Phase 1

## 1 Big Step Evaluation Rules

The lazy list primitives extend L1++ with delayed evaluation capabilities. The formal semantics are defined by the following evaluation rules:

### 1.1 Lazy Cons Construction

$$\frac{\mathcal{E}' = \text{copy}(\mathcal{E})}{\mathcal{E}; \mathcal{S}; \text{lcons}(M, N) \Downarrow \text{lcons}(\mathcal{E}', M, N); \mathcal{S}} \tag{1}$$

Unlike strict cons, `lcons` does not evaluate its arguments $M$ and $N$. Instead, it captures the current environment $\mathcal{E}$ by creating a copy $\mathcal{E}'$, storing the unevaluated expressions for future computation.

### 1.2 Pattern Matching on Lazy Lists

$$\frac{\begin{array}{c} \mathcal{E}; \mathcal{S}; M \Downarrow \text{lcons}(\mathcal{E}', H, T); \mathcal{S}' \\ \mathcal{E}'; \mathcal{S}'; H \Downarrow V_H; \mathcal{S}'' \\ \mathcal{E}'; \mathcal{S}''; T \Downarrow V_T; \mathcal{S}''' \\ \mathcal{E}[x \mapsto V_H][l \mapsto V_T]; \mathcal{S}'''; R \Downarrow U; \mathcal{S}'''' \end{array}}{\mathcal{E}; \mathcal{S}; \text{match } M\{\text{nil} \to N \mid (x :: l) \to R\} \Downarrow U; \mathcal{S}''''} \tag{2}$$

Pattern matching forces evaluation: when matching against a lazy list, the head expression $H$ and tail expression $T$ are evaluated using the captured environment $\mathcal{E}'$, producing values $V_H$ and $V_T$ that are bound to pattern variables $x$ and $l$, respectively.

## 2 Implementation

### 2.1 Core Data Structure

The `VLazyList` class implements ”stateful” lazy evaluation - i.e., keeps track of evaluation occurrences and only evaluates once.

```java
public class VLazyList implements IValue {
    private Environment<IValue> env;
    private ASTNode headExpr, tailExpr;
    private boolean evaluated = false;
    private IValue head, tail;

    public void evaluate() throws InterpreterError {
        if (!evaluated) {
            head = headExpr.eval(env);
            tail = tailExpr.eval(env);
            evaluated = true;
        }
    }
}
```

### 2.2 Lazy Construction

The ASTLCons node creates lazy lists without evaluation:

```
public IValue eval(Environment<IValue> e) {
    Environment<IValue> capturedEnv = e.copy();
    return new VLazyList(capturedEnv, head, tail);
}
```

### 2.3 Forcing Evaluation

Pattern matching in ASTMatch triggers evaluation:

```
case VLazyList lazyList -> {
    IValue head = lazyList.getHead(); // Forces evaluation
    IValue tail = lazyList.getTail();
    Environment<IValue> newEnv = e.beginScope();
    newEnv.assoc(headVar, head);
    newEnv.assoc(tailVar, tail);
    return consCase.eval(newEnv);
}
```

## 3 Example: Infinite Streams

```
let fibo = fn a, b => { a ?: (fibo b (a+b)) };
let stream = fibo (0) (1);
match stream {
    nil -> 0
    | h :: t -> h  // Returns 0, forces evaluation
}
```

This creates an infinite Fibonacci stream where elements are computed on-demand through pattern matching.