# Programming Languages
# Project Phase 1

Inês Sousa, 99962

May 26, 2025

## 1 Implementation of Lazy Lists

This implementation supports lazy lists in L1++ through three main AST nodes and the corresponding runtime values.

### AST Nodes

- `ASTNil`: Represents the empty list. When evaluated, it returns a `VList` instance flagged as empty.
- `ASTList(head, tail)`: Represents a lazy `cons` cell with a `head` and `tail` which are unevaluated AST nodes. It includes a flag `in_match` that controls whether the list should evaluate its elements immediately or stay lazy.
- `ASTMatch(list, body, nil_body, head_parameter, tail_parameter)`: Represents pattern matching over lazy lists. It matches on the `list` expression and chooses between the `nil_body` if the list is empty or evaluates the `body` in a new environment binding the head and tail variables.

### Runtime Representation — `VList`

The runtime list representation is the `VList` class, which models both empty and non-empty lazy lists:

- `VList()` constructs an empty list, marked by `empty = true`.
- `VList(ASTNode head, ASTNode tail)` constructs a lazy list node where `head` and `tail` are unevaluated AST nodes. This node has `empty = false` and `in_match = false`.
- `VList(IValue vhead, IValue vtail)` constructs an evaluated list node where `vhead` and `vtail` are forced values. This node has `empty = false` and `in_match = true`.

## Lazy Evaluation Strategy

- When an `ASTList` node is evaluated normally (outside a match), it returns a `VList` holding the unevaluated `head` and `tail` AST nodes, preserving laziness.
- When an `ASTList` node is evaluated inside a `match` (signaled by `in_match = true`), it recursively evaluates its `head` and `tail`, producing a fully evaluated `VList` node.
- The `inside_match()` method propagates the `in_match` flag recursively to the `tail` if it is also an `ASTList`, ensuring that nested lists are fully evaluated during pattern matching.

## Pattern Matching Implementation

`ASTMatch.eval` works as follows:

(a) Evaluate the `list` expression once, getting an `IValue`.

(b) Check that this value is a `VList`; if not, throw an error.

(c) If the list is empty (`vlist.empty == true`), evaluate and return `nil_body`.

(d) If the list is non-empty:

- Begin a new environment scope.
- If the list has already been evaluated in a previous match (`vlist.in_match == true`), bind the head and tail variables to the already evaluated values `vhead` and `vtail`.
- Otherwise (lazy list not yet forced):
  - Evaluate the `head` expression and bind it to the head variable.
  - If the tail is another `ASTList`, call `inside_match()` on it to mark it and its tails for forced evaluation.
  - Evaluate the `tail` expression and bind it to the tail variable.
- Evaluate and return the `body` in this extended environment.

## Summary

This implementation achieves lazy lists by storing list cells as unevaluated AST nodes until pattern matching forces evaluation. The `in_match` flag ensures that evaluation happens exactly once per `match`, and recursive propagation of this flag guarantees nested lists are fully forced only when needed. This design preserves the lazy semantics and prevents repeated evaluations, while pattern matching provides a clean way to destructure lazy lists safely.