# Swimft

Alunos: Beatriz e Vitor

Data: 19/07/2019

Universidade Federal Fluminense

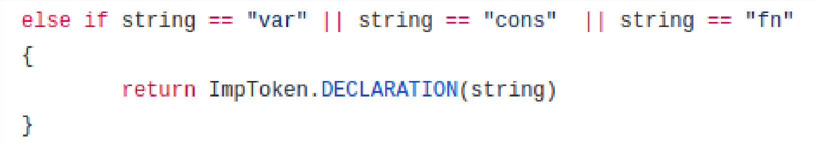
# Objetivos desta apresentação

- Apresentação de conclusão da terceira e última etapa do trabalho

# O que foi feito

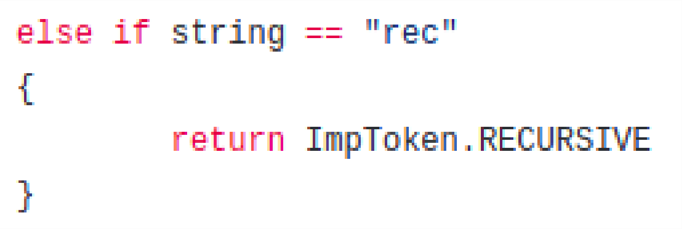
- Lexer adaptado para Imp-2
- Parser para Imp-2
- Pi Framework: implementado compilador com declarações de operações e funções recursivas

# O que foi feito: Lexer



```
else if string == "var" || string == "cons"  || string == "fn"
{
        return ImpToken.DECLARATION(string)
}
```

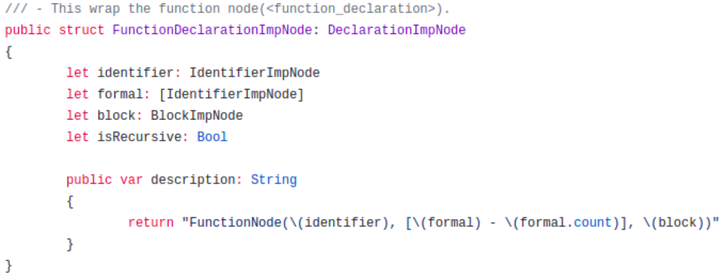
**Figura 1:** fn - declaration token



```
else if string == "rec"
{
        return ImpToken.RECURSIVE
}
```

**Figura 2:** rec - recursive token

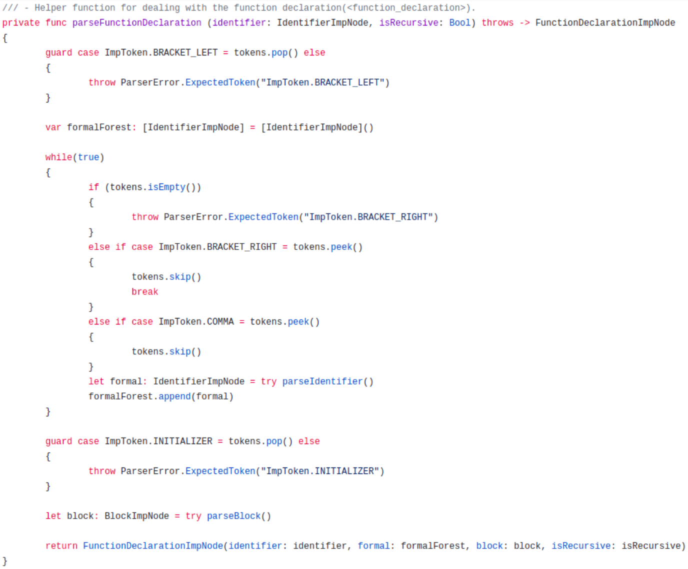
# O que foi feito: Parser



```swift
/// - This wrap the function node(<function_declaration>).
public struct FunctionDeclarationImpNode: DeclarationImpNode
{
        let identifier: IdentifierImpNode
        let formal: [IdentifierImpNode]
        let block: BlockImpNode
        let isRecursive: Bool

        public var description: String
        {
                return "FunctionNode(\(identifier), [\(formal) - \(formal.count)], \(block))"
        }
}
```

**Figura 3:** estrutura do nó imp - FunctionDeclaration

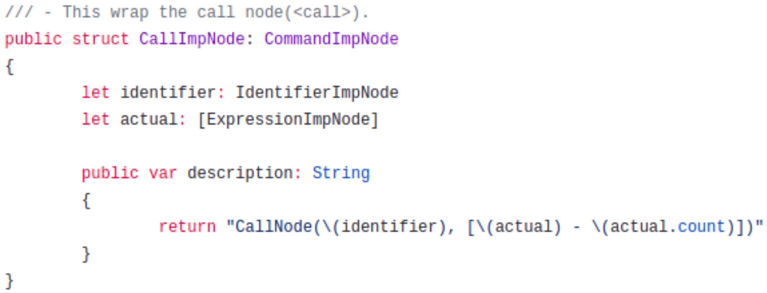
# O que foi feito: Parser



```
/// - Helper function for dealing with the function declaration(<function_declaration>).
private func parseFunctionDeclaration (identifier: IdentifierImpNode, isRecursive: Bool) throws -> FunctionDeclarationImpNode
{
        guard case ImpToken.BRACKET_LEFT = tokens.pop() else
        {
                throw ParserError.ExpectedToken("ImpToken.BRACKET_LEFT")
        }

        var formalForest: [IdentifierImpNode] = [IdentifierImpNode]()

        while(true)
        {
                if (tokens.isEmpty())
                {
                        throw ParserError.ExpectedToken("ImpToken.BRACKET_RIGHT")
                }
                else if case ImpToken.BRACKET_RIGHT = tokens.peek()
                {
                        tokens.skip()
                        break
                }
                else if case ImpToken.COMMA = tokens.peek()
                {
                        tokens.skip()
                }
                let formal: IdentifierImpNode = try parseIdentifier()
                formalForest.append(formal)
        }

        guard case ImpToken.INITIALIZER = tokens.pop() else
        {
                throw ParserError.ExpectedToken("ImpToken.INITIALIZER")
        }

        let block: BlockImpNode = try parseBlock()

        return FunctionDeclarationImpNode(identifier: identifier, formal: formalForest, block: block, isRecursive: isRecursive)
}
```

**Figura 4:** processamento de Function token

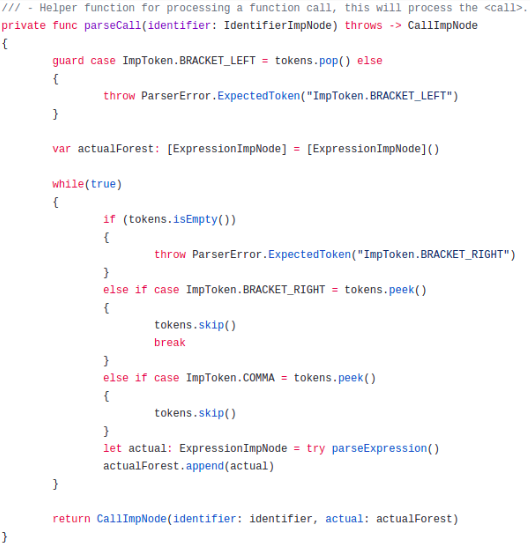
# O que foi feito: Parser



```
/// - This wrap the call node(<call>).
public struct CallImpNode: CommandImpNode
{
        let identifier: IdentifierImpNode
        let actual: [ExpressionImpNode]

        public var description: String
        {
                return "CallNode(\(identifier), [\(actual) - \(actual.count)])"
        }
}
```

**Figura 5:** estrutura do nó imp - Call

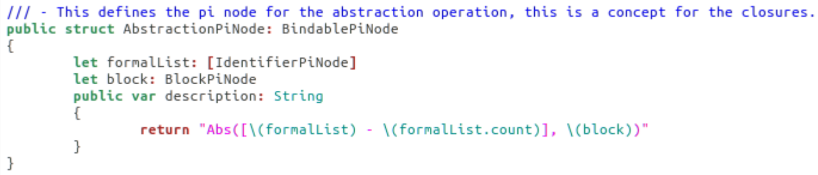
# O que foi feito: Parser



```
/// - Helper function for processing a function call, this will process the <call>.
private func parseCall(identifier: IdentifierImpNode) throws -> CallImpNode
{
        guard case ImpToken.BRACKET_LEFT = tokens.pop() else
        {
                throw ParserError.ExpectedToken("ImpToken.BRACKET_LEFT")
        }

        var actualForest: [ExpressionImpNode] = [ExpressionImpNode]()

        while(true)
        {
                if (tokens.isEmpty())
                {
                        throw ParserError.ExpectedToken("ImpToken.BRACKET_RIGHT")
                }
                else if case ImpToken.BRACKET_RIGHT = tokens.peek()
                {
                        tokens.skip()
                        break
                }
                else if case ImpToken.COMMA = tokens.peek()
                {
                        tokens.skip()
                }
                let actual: ExpressionImpNode = try parseExpression()
                actualForest.append(actual)
        }

        return CallImpNode(identifier: identifier, actual: actualForest)
}
```

**Figura 6:** processamento de Call token

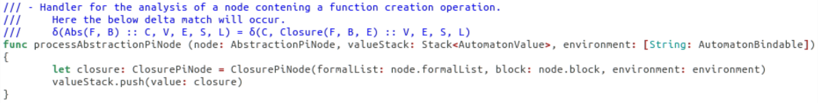
# O que foi feito: PiFramework



```
/// - This defines the pi node for the abstraction operation, this is a concept for the closures.
public struct AbstractionPiNode: BindablePiNode
{
        let formalList: [IdentifierPiNode]
        let block: BlockPiNode
        public var description: String
        {
                return "Abs([\(formalList) - \(formalList.count)], \(block))"
        }
}
```

**Figura 7:** estrutura do pi node - Abs

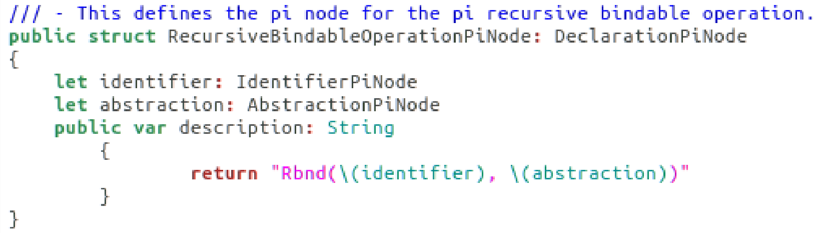
# O que foi feito: PiFramework



```
/// - Handler for the analysis of a node contening a function creation operation.
///     Here the below delta match will occur.
///     δ(Abs(F, B) :: C, V, E, S, L) = δ(C, Closure(F, B, E) :: V, E, S, L)
func processAbstractionPiNode (node: AbstractionPiNode, valueStack: Stack<AutomatonValue>, environment: [String: AutomatonBindable])
{
        let closure: ClosurePiNode = ClosurePiNode(formalList: node.formalList, block: node.block, environment: environment)
        valueStack.push(value: closure)
}
```

**Figura 8:** Abs handler

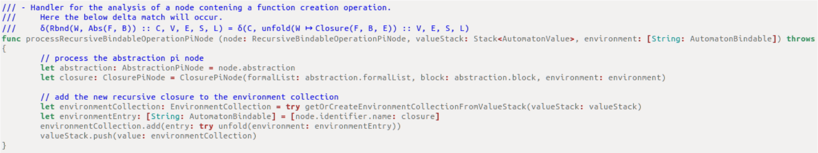
# O que foi feito: PiFramework



```
/// - This defines the pi node for the pi recursive bindable operation.
public struct RecursiveBindableOperationPiNode: DeclarationPiNode
{
    let identifier: IdentifierPiNode
    let abstraction: AbstractionPiNode
    public var description: String
        {
                return "Rbnd(\(identifier), \(abstraction))"
        }
}
```

**Figura 9:** estrutura do pi node - Rbnd

# O que foi feito: PiFramework



```
/// - Handler for the analysis of a node contening a function creation operation.
///     Here the below delta match will occur.
///     δ(Rbnd(W, Abs(F, B)) :: C, V, E, S, L) = δ(C, unfold(W ↦ Closure(F, B, E)) :: V, E, S, L)
func processRecursiveBindableOperationPiNode (node: RecursiveBindableOperationPiNode, valueStack: Stack<AutomatonValue>, environment: [String: AutomatonBindable]) throws
{
        // process the abstraction pi node
        let abstraction: AbstractionPiNode = node.abstraction
        let closure: ClosurePiNode = ClosurePiNode(formalList: abstraction.formalList, block: abstraction.block, environment: environment)

        // add the new recursive closure to the environment collection
        let environmentCollection: EnvironmentCollection = try getOrCreateEnvironmentCollectionFromValueStack(valueStack: valueStack)
        let environmentEntry: [String: AutomatonBindable] = [node.identifier.name: closure]
        environmentCollection.add(entry: try unfold(environment: environmentEntry))
        valueStack.push(value: environmentCollection)
}
```

**Figura 10:** Rbnd handler

# O que foi feito: Pi Framework

- Vamos ao código...