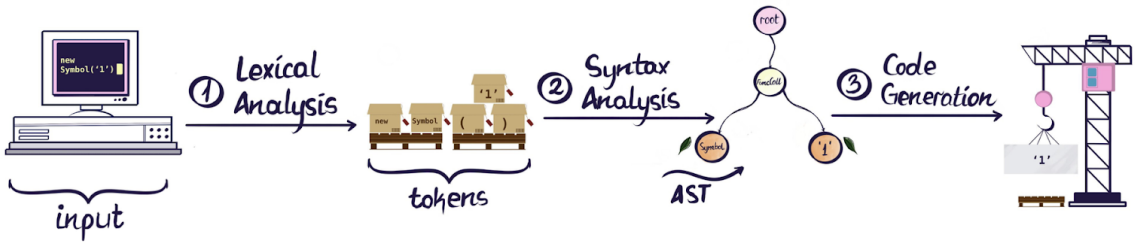
Linguagem Vikings: aspectos da AST (Abstract Sintax Tree)

A linguagem vikings busca implementar após a análise sintática uma AST ou seja uma árvore sintática abstrata para que enfim possa ser gerado um código de máquina. A imagem abaixo foi retirada deste link: https://www.twilio.com/blog/abstract-syntax-trees qual é um artigo escrito por Dominik Kundel que explica o que é uma AST e como pode ser utilizada.



Dentro deste artigo existe um link que mostra o AST Explorer https://astexplorer.net/ neste site é possível gerar a AST dentre várias linguagens distintas.
A implementação da AST para linguagem vikings baseou-se no artigo citado acima, nas aulas da disciplina de compiladores, mas principalmente baseou-se em como é gerado a AST dentro do AST Explorer visando que já existe um padrão para as linguagens tradicionais como Java, C#, Javascript, Python entre outras já padronizadas pelo AST Explorer em busca da linguagem vikings também ser padronizada ao mesmo nível das linguagens citadas acima.

Detalhes sobre a implementação:
É feito um loop em todos os tokens verificados pela análise sintática, e busca encontrar os tokens que identificam declarações de variáveis, atribuições a variáveis, entrada e saída de dados, estruturas condicionais com if, else e while.

- Escopo do Programa
  AST do Vikings começa assim Program e o body contém todas as informações do corpo do programa
  Program {
  body: [
  ]
  }

- Declarações de variáveis:
  Program {
  body: [
  VariableDeclaration {
  type: INT
  name: a

```
            },
            VariableDeclaration {
                type: FLOAT
                name: b
            },
            VariableDeclaration {
                type: BOOL
                name: c
            },
        ]
}
```

- Atribuições de variáveis:
    - Atribuições Simples (a=2; ou a=b;):

```
ExpressionStatement {
        AssignmentExpression {
            operator : =
            left : Indentifier {
                name : a
            },
            right : Literal {
                value : 3
            },
        },
},
```

    - Atribuições Complexas (a = 2 + b; ou a = 2 + 2;):

```
ExpressionStatement {
        AssignmentExpression {
            operator : =
            left : Indentifier {
                name : a
            },
            right : BinaryExpression {
                left : Literal {
                    value : 2
                },
                operator : +
                right : Identifier {
                    name : b
                },
            }
        },
},
```

- Entrada e saída de dados:

```
InputOutputDeclaration {
        type: SCAN
        id: a
},
InputOutputDeclaration {
```

```
        type: PRINT
        id: b
},
```

- estruturas condicionais:
  - If
    - Condição simples negativa (if(!a))

```
IfStatement {
        condition:  UnaryExpression {
                operator: !
                argument: Identifier {
                        name: c
                }
        },
        body: BlockStatement {
                body: [
                ]
        }
}
```

    - Condição simples (If(a ==b))

```
IfStatement {
        condition:  BinaryExpression {
                left: Identifier {
                        name: a
                }
                operator: ==
                right: Identifier {
                        name: b
                }
        },
        body: BlockStatement {
                body: [
                ]
        }
}
```

    - Condição completas (If(a == b & a < 1))

```
IfStatement {
        condition:  LogicalExpression {
                left:  BinaryExpression {
                        left: Identifier {
                                name: a
                        }
                        operator: ==
                        right: Identifier {
                                name: b
                        }
                }
                operator: &
                right:  BinaryExpression {
                        left: Identifier {
```

```
                    name: a
                }
                operator: <
                right: Literal {
                    value: 1
                }
            }
        },
        body: BlockStatement {
            body: [
            ]
        }
    }
```

- Else

```
ElseStatement {
        body: BlockStatement {
            body: [
            ]
        }
    }
```

- While
  - Condição simples negativa (while(!a))

```
WhileStatement {
        condition:  UnaryExpression {
            operator: !
            argument: Identifier {
                name: c
            }
        },
        body: BlockStatement {
            body: [
            ]
        }
    }
```

  - Condição simples (while(a ==b))

```
WhileStatement {
        condition:  BinaryExpression {
            left: Identifier {
                name: a
            }
            operator: ==
            right: Identifier {
                name: b
            }
        },
        body: BlockStatement {
            body: [
            ]
        }
```

```
}
```

- Condição completas (while(a == b & a < 1))

```
WhileStatement {
        condition:  LogicalExpression {
                left:  BinaryExpression {
                        left: Identifier {
                                name: a
                        }
                        operator: ==
                        right: Identifier {
                                name: b
                        }
                }
                operator: &
                right:  BinaryExpression {
                        left: Identifier {
                                name: a
                        }
                        operator: <
                        right: Literal {
                                value: 1
                        }
                }
        },
        body: BlockStatement {
                body: [
                ]
        }
}
```