TRABALHO DE COMPILADORES

O objetivo deste projeto é implementar um scanner e parser para a linguagem Kotlin, recorrendo à utilização de ferramentas como Alex e Happy

1. Alex – Analisador Léxico
2. Happy – Analisador Sintático
3. AST - Árvore de Sintaxe Abstrata

O objetivo é analisar o código fonte da linguagem, converter em tokens, validar a sua estrutura sintática e gerar a devida AST

● Estrutura do Projeto

O projeto está dividido em três partes principais:
1. Lexer.x - Analisador Léxico
2. Parser.y - Analisador Sintático
3. AST.hs - Definição da Árvore de Sintaxe Abstrata

1.Lexer.x

Este arquivo é responsável por realizar a conversão da entrada de texto em tokens

Tokens:
O lexer define vários tokens, por exemplo:
- `Num` : Representa números inteiros
- `STR` : Representa strings literais
- `IDENT` : Representa identificadores

Expressões Regulares:
São regularmente utilizadas para facilitar a identificação de certos padrões nos tokens:
- `[A-Za-z_][A-Za-z0-9_]*` : Permite-nos informar que estamos perante um identificador
- `"/*"(\*$white|(\*[^\/])|[^\*]|$white)*"*/"` : Ignora comentários de múltiplas linhas

2.Parser.y

Este arquivo processa os tokens gerados pelo lexer e define as regras de gramática que são usadas para gerar o arquivo `Parser.hs` , construindo a AST responsável pela representação hierárquica do programa

Gramática:

Específica como os tokens devem ser organizados para formar um programa sintaticamente válido

As produções definidas em `Parser.y` implementam a estrutura de `Parser.hs`

Define produções para expressões e declarações da linguagem:

- `Exp "+" Exp` : Operações Aritméticas e Lógicas
- `Var`(mutáveis) e `Val`(imutáveis) : Declaração de Variáveis
- `IF`, `ELSE` e `WHILE` : Estruturas condicionais

A precedência e associatividade:

São conceitos fundamentais para resolver a ambiguidade das gramáticas. Para definir a prioridade dos operadores e especificar os casos em que eles não podem ser combinados de forma associativa utilizamos os modificadores %left e %nonassoc

Exemplos de Produções:

- `Block : "{" Stm* "}"` : Bloco de Código
- `Assi : id "=" Exp` : Atribuição

3.AST.hs

Este arquivo define a representação hierárquica e simplificada do programa, onde cada nó representa uma expressão ou comando. A AST permite uma representação mais organizada e manipulável do código kotlin, sendo útil para análises adicionais ou para execução do código

Representação de Expressões:

- `Num` : Representa um número
- `Str` : Representa uma string
- `BinOp` : Representa operações binárias como soma, subtração, etc
- `NotOp` : Representa operações unárias como negação
- `Ident` : Representa identificadores

Operações Binárias são representadas pelo tipo Op que inclui operações do tipo:

- `Add` : Soma
- `Sub` : Subtração
- `Mul` : Multiplicação
- `Div` : Divisão

Comandos do Programa:
- `Assign` : Atribuição de valor
- `IF`, `IfElse`, `While` : Estruturas condicionais
- `Print`, `Println` : Comandos de saída
- `Fun` : Definição de funções

- **Para compilar e executar o projeto, siga os passos abaixo:**

1. Gerar o Lexer: `alex lexer.x` (Gera o código Haskell para o lexer a partir de lexer.x)
2. Gerar o Parser: `happy parser.y` (Gera o código Haskell para o parser a partir de parser.y)
3. Compilar o Código: `ghc -o main Main.hs` (Compila o código Haskell, criando um executável)
4. Executar o Programa: `./Main.hs` (Executa o programa compilado)

Exemplo do Programa:
`Foram criados vários ficheiros de teste, que podem ser testados diretamente após compilar o ficheiro Main.hs , usando o seguinte comando, ./main < input#.txt sendo # o número do ficheiro.`

`(input1.txt)`
- Tokens gerados : `FUN, MAIN, LPAREN, RPAREN, LBRACKET, VAR IDENT 'x', ASSIGN, NUM 4, SEMICOLON, VAL IDENT 'y', PLUS, MYTRUE, MYFALSE, RBRACKET`
- Estrutura da AST : `Main (Block [Var "a" (Num 10),Val "b" (Num 20),Var "c" (BinOp Add (Ident "a") (BinOp Mul (Ident "b") (Num 2))),Print (PrintStr "Resultado: "),Println (PrintExp (Ident "c"))])`