**TRABALHO PRÁTICO 01**

**COMPILADORES**

**Grupo COMP TP1 T02 G11**

Robert Gleison dos Reis Pereira up202200496

Lucas ……………… up………….

Sumário

**Estrutura de Arquivos**  
  
**-> examples:** Diretório onde se encontram exemplos de código Kotlin para serem usados como input no compilador criado.

**-> Main.hs:** Arquivo que comporta a chamada inicial do programa (main :: IO()) e gerencia as componentes de analisador léxico e parsing. Também responsável por ler os inputs do diretorio “examples”, e fazer as chamadas para o analisador léxico, e em seguida, para o parser. Nesta primeira fase do projeto, a árvore sintática abstrata é o final do código.

A chamada para o analisador léxico entrega um input em formato texto e retorna uma lista de tokens baseado no input.

A chamada para o parser entrega uma lista de tokens gerados pelo analisador léxico e retorna uma árvore sintática abstrata.

**-> Lexer.x:** Onde se encontra o código responsável pelo analisador léxico. São instruções feitas com a ferramenta Alex, que, ao ser compilada, gera um arquivo Haskell (Lexer.hs), este sim, responsável por pegar um input e transformar numa lista de tokens.

**-> Parser.y:** Onde se encontra o código responsável pelo parser. São instruções feitas com a ferramenta Happy, que, ao ser compilada, gera um arquivo Haskell (Parser.hs), este sim, responsável por transformar uma lista de tokens em uma árvore sintática abstrata.

Outros arquivos gerados como: Lexer.o, Lexer.hi… são arquivos de metadados e arquivos auxiliares de compilação. Portanto, não gerados pelo grupo, mas pelo compilador.

**Lexer.x**

Código declarativo para geração do analisador léxico. Foi feito com a ferramenta Alex (<https://haskell-alex.readthedocs.io/en/latest/>).

O código foi dividido em:

-> Criar conceitos de caracteres alfabéticos, alphanumericos e número usando regex. Assim evita-se repetição de regex.

-> Criação dos tokens da linguagem Kotlin.

Exemplos:

-- Special operators

| RANGE | SAFE\_ACCESS | ELVIS | SCOPE\_RES | ARROW …

-- Assignment operators

| ASSIGN | PLUS\_ASSIGN | MINUS\_ASSIGN | TIMES\_ASSIGN …

-- Arithmetic and comparison operators

| PLUS | MINUS | TIMES | DIVIDE | MOD …

-> Mapeamento de caracteres e respectivos tokens. Responsável por transformar o input texto numa lista de tokens. Sendo eles:

- Delimitadores ( vírgulas, parênteses, colchetes…)

- Operadores especiais ( arrow, range, null safety (?)...)

- Operadores aritméticos ( soma, multiplicação, mod…)

- Operadores lógicos ( and, or, not, greater than…)

- Palavras chaves (val, fun, if, else, return …)

- Tipos (Int, Float, Double…)

- Literais ( valores de string, valores de booleanos e valores de números)

- Identificadores (ID, para atribuir um valor a uma referência a ser reconhecida no futuro)

Exemplos:

"(" { \\_ -> LPAREN }

")" { \\_ -> RPAREN }

"}" { \\_ -> RBRACE }

"+" { \\_ -> PLUS }

"-" { \\_ -> MINUS }

Ao compilar o arquivo Lexer.x, será gerado um arquivo Haskell chamado Lexer.hs, que será responsável pelo processamento do input da forma que foi declarado nas regras. Retorna uma lista de tokens.

Para a compilação, o seguinte comando é utilizado:

alex Lexer.x -o Lexer.hs

**Parser.y**

Código declarativo para geração do parser. Foi feito com a ferramenta Happy (<https://haskell-happy.readthedocs.io/en/latest/>).

O código foi dividido em:

-> Gerar os tokens utilizados durante o parsing. Sendo eles:

- Delimitadores ( vírgulas, parênteses, colchetes…)

- Operadores especiais ( arrow, range, null safety (?)...)

- Operadores aritméticos ( soma, multiplicação, mod…)

- Operadores lógicos ( and, or, not, greater than…)

- Palavras chaves (val, fun, if, else, return …)

- Tipos (Int, Float, Double…)

Exemplos:

Int { INT }

Float { FLOAT }

'<' { LTHAN }

'<=' { LTE }

'{' { LBRACE }

'}' { RBRACE }

-> Declaração de precedência de operadores, no caso, quais operações terão prioridade em relação a outras no processamento (multiplicação, por exemplo, tem precedência sobre soma, mas um soma dentro de parênteses tem precedẽncia sobre multiplicação).

-> Declaração da gramática da linguagem Kotlin. Aqui serão utilizados os tokens e a precedência de operadores dentro de regras especificadas para evitar ambiguidades na compilação do código. Isto é, um código deve ter apenas uma maneira de ser lido, evitando que o mesmo código possa gerar resultados diferentes.

– Colocar exemplo de gramática –

Ao compilar o arquivo Parser.y, será gerado um arquivo Haskell chamado Parser.hs, que será responsável pelo processamento da lista de tokens da forma que foi declarado nas regras. Retorna uma árvore sintática abstrata.

Para a compilação, o seguinte comando é utilizado:

happy Parser.y -o Parser.hs

**INSTRUÇÕES DE USO**

Para teste do código se precisa de um input em formato texto com um código Kotlin. O usuário pode utilizar seu próprio exemplo ou utilizar algum do diretório “examples”.

Para isso, basta confirmar que o lexer e parser estão compilados usando:

alex Lexer.x -o Lexer.hs

happy Parser.y -o Parser.hs

Em seguida, compilar o Main.hs e utiliza com algum input:

ghc Main.hs -o Main

./Main < examples/example1.kt

**Conclusão**

Durante essa primeira parte do trabalho, foram trabalhados os conceitos e analisadores léxicos e parsing. No momento atual, se consegue extrair uma lista de tokens e uma árvore abstrata de qualquer código Kotlin.

**Trabalho Futuro**

Implementação das próximas etapas de um compilador.