

**UNIDADE CURRICULAR:** Compilação

**CÓDIGO:** 21018

**DOCENTES:** Jorge Morais e Rúdi Gualter (tutor)

**A preencher pelo estudante**

**NOME, N.º DE ESTUDANTE, TURMA**

* + Diogo Moreira, 2104140, turma 2
  + José Ribeiro, 1601792, turma 1
  + Silvana Oliveira, 2102785, turma 2
  + Vítor Frango, 1802925, turma 1
  + **EQUIPA: Compile Wizards**

**CURSO:** Licenciatura Engenharia Informática

**DATA DE ENTREGA:** 28 de abril 2024

**TRABALHO / RESOLUÇÃO PARTE 1:**

**Os ficheiros criados para a resolução do trabalho foram os seguintes: MontPy.g4, MontPyParserTest.java, MontPyParser.java, MontPyLexerTest.java, MontPyLexer.java, README.txt (contém as instruções de compilação e execução). De seguida explica-se com detalhe o objetivo e função de cada um dos ficheiros acima enumerados:**

**- MontPy.g4: arquivo com a gramática do ANTLR utilizado para definir a estrutura da linguagem MontPy, servindo de base para gerar os analisadores léxicos e sintáticos. A partir dele o ANTLR gera as classes Java para a analise do código;**

**- MonPyLexer.java: é o analisador léxico gerado a partir do MontPy.g4. Utiliza as definições de tokens para identificar os elementos lexicais na linguagem MontPy, a sua implementação contém:**

**- Tokens: palavras-chave, operadores e tipos de dados;**

**- Métodos de construção: constroem objetos de token para processar dados de entrada;**

**- Funções de controlo: efetuam a gestão e analisam a interpretação do texto;**

**- MontPyParser.java: é o analisador sintático gerado pelo ANTLR, utiliza o TokenStream para criar a estrutura sintática a partir do texto, incluem:**

**- Regras de gramática: definem como os tokens são combinados para criar expressões e declarações;**

**- Métodos de analise: analisam expressões, estruturas de controlo e chamadas de funções.**

**- MontPyLexerTest.java: através de um arquivo de entrada, testa o analisador léxico, o lexer identifica os tokens da linguagem e incluem:**

**- Entrada de teste: arquivo a ser analisado;**

**- Loop de tokens: loop para iterar sobre os tokens gerados pelo lexer identificando os possíveis erros.**

**- MontPyParserTest.java: teste para o analisador sintático, utiliza o lexer para criar um TokenStream e testar o parser, contém:**

**- Árvore de analise: gera uma árvore de análise para validar o comportamento do parser;**

**- ParserTreeWalker: percorre a árvore de análise e verifica se está correta.**

**No decorrer da construção da árvore sintática os desafios encontrados foram de níveis ou complexidades dispares, no entanto, todos eles estavam interligados. Identificamos os seguintes desafios:**

1. **Complexidade da gramática: não só ao nível das regras e casos especiais, mas também pelas próprias ambiguidades de interpretação que a mesma pode gerar, levando por si só a alguma dificuldade na construção da árvore sintática;**
2. **Erros de sintaxe: durante a análise lexical, foram gerados tokens inválidos criados pelo lexar que introduziram erros desde o início do processo. Estes tokens errados prejudicaram a progressão da análise, criando obstáculos adicionais que necessitaram de uma resolução imediata para permitir a continuidade do projeto;**
3. **Gestão dos Scopes: a correta análise dos scopes exigiu que as variáveis e as funções estivessem no contexto apropriado de forma a evitar erros que resultavam em árvores sintáticas incorretas;**
4. **Precedência de operadores: os diferentes níveis de precedência e associatividade precisaram de ser interpretados corretamente de forma a evitar uma construção errada da árvore sintática;**
5. **Tratamentos de erros: foi essencial desenvolver o parser de modo que este pudesse não só identificar erros de sintaxe eficientemente, mas também reportá-los de maneira clara e útil, o que permitiu uma intervenção mais rápida na correção dos erros.**

Conforme solicitado no enunciado e diante dos desafios encontrados, as decisões tomadas foram organizadas dentro da estrutura da gramática MontPy. Isso incluiu a definição de regras de sintaxe, a hierarquização dos componentes e o uso de expressões regulares. As escolhas feitas na construção da árvore sintática visaram assegurar tanto a correção quanto a eficiência do processo. Essas escolhas foram subagrupadas por meio da divisão em componentes distintos, agrupamento por precedências, estruturas de controlo e tratamento de erros.

**Foram realizados diferentes testes ao projeto, que podem ser encontrados na pasta “testes” enviada em anexo. Apresentamos de seguida os testes e as respetivas *Parse Tree Inspector* que consideramos mais relevantes:**

|  |
| --- |
| Uma imagem com captura de ecrã, texto, Software de multimédia, software  Descrição gerada automaticamente  Figura 1 - Código e output do terminal da análise léxica (teste\_lexica1.txt) |
| Uma imagem com texto, software, Software de multimédia, Software gráfico  Descrição gerada automaticamente  Figura 2 - Código e output do terminal da análise sintática (teste\_sintaxe.txt) |
| Figura 3 - Código e output do terminal do teste de erro de precedências (teste\_precedencia.txt) |

**O que quer dizer FCC**

Na nossa opinião FCC refere-se a Fedjmike’s C Compiler (FCC), que é um compilador de código aberto, disponível em <https://github.com/Fedjmike/fcc>, que implementa uma linguagem semelhante ao C, no entanto com diferenças, umas mais significativas que outras, como por exemplo: Sistemas de módulos simples; Tipo booleano: inclui os literais ‘true’ e ‘false’; Semântica diferente: Numespace unificado, operadores lógicos (‘||’ , ‘&&’ e ‘!’, funções retornando booleanos, funções variadas: como por exempro ‘va\_arg’, Tipos definidos em ‘stdin.h’, Protótipos vazios; Suporte para C99 e C11: Stricts/unions anónimos, inicializadores designados e literais compostos, comentários estilo C++; Recursos removidos: Switch, Pré processador, Bitfields, Goto e labels; Saída de código: gera código assembly para CPUs x86(32 bits) com suporte experimental para AMD64/64 bits);

Auto-hospedado visto ser capaz de compilar o próprio código fonte.

**Webgrafia:**

**ANTLR Doc: https://github.com/antlr/antlr4/blob/master/doc/index.md ;**

**The ANTLR Mega Tutorial: https://tomassetti.me/antlr-mega-tutorial/**

**The Definitive ANTLR4 Reference, Terence Parr. The Pragmatic Programmers, 2012.**

**Compilers: principles, techniques and tools, 2nd Ed., Aho, Lam, Setti, Ullman, Addison-Wesley, 2007.**

**Essentials of Compilation: An Incremental Approach in Python, Jeremy G. Siek. MIT Press, 2023**

**Compiladores – Da Teoria à Prática, Pedro Reis Santos e Thibault Langlois. FCA, 2015.**

Fedjmike’s C Compiler (FCC), disponível em: <https://github.com/Fedjmike/fcc>