**O relatório da parte I e a parte II deve ter o texto com espaçamento de 1,5, letra tamanho 10 ou 11, evite alterar formato para caber mais texto(para remover)**

**Introdução - MOCC (My Own C Compiler)**

Este relatório descreve o desenvolvimento de um compilador para uma linguagem fictícia no contexto da unidade curricular de Compilação (2024/25). A linguagem MOC adota uma sintaxe semelhante à linguagem C, com restrições e simplificações específicas que visam facilitar a construção de um compilador com ferramentas como **ANTLR4**, não nos vamos alongar na especificação que nos foi dada para a linguagem, mas aqui [link](https://elearning.uab.pt/pluginfile.php/3918150/mod_assign/introattachment/0/MOCC.pdf?forcedownload=1) do enunciado para contextualização:

Por motivos de compatibilidade e possíveis diferenças de comportamento entre versões, recomendamos ANTLR 4.13.2 e Python 3.11, com as quais todos os testes foram realizados.

Sendo o foco desta atividade, a análise léxica e sintática, foram gerados os seguintes ficheiros:

MOC.g4 – Gramática (Lexer e Parser)

* Adicionar aqui cada ficheiro e o que é

Como utilizar:

* Instruções

**Gramática**

**Começámos a definição da gramática pela análise léxica, através da implementação de tokens de gramática antlr, sendo a base que permite construir os elementos básios da nossa linguagem, segue aqui a sua definição:**

* **Palavras-chave**: *int, double, void, main, read, readc, reads, write, writec, writev, writes, if, else, while, for, return,*
* **Operadores**: *+, -, \*, /, %, <, <=, >, >=, ==, !=, &&, ||, !, =,*
* **Pontuação e símbolos**: *,, ;, [, ], {, }, (, ),*
* **Literais**: *números inteiros, números reais e strings,*
* **Identificadores**: *nomes de variáveis, funções, etc.*
* **Comentários**: *comentários de linha (//) e bloco (/\* ... \*/) – ignorados pelo lexer,*
* **Espaços em branco**: ignorados pelo lexer.

A seguir foram implementadas regras sintáxicas, através da implementação de regras de parser que definem se a estrutura dos programas é válida, definidas pelas seguintes regras:

* **programa**: estrutura global que combina os protótipos e o corpo principal do programa.
* **funcao, prototipo, funcaoPrincipal, prototipoPrincipal**: usas para definir e declarar funções, incluindo main.
* **declaracao, variavel:** regras para declaração de variáveis simples e vetores, com ou sem inicialização.
* **instrucao:** regra base para todas as instruções possíveis, incluindo blocos, condições, ciclos, leitura/escrita e retorno.
* **expressao:** criada para suportar operações, nomeadamente: aritméticas, lógicas, relacionais e também operações de casting, chamadas a funções e acessos a vetores.
* **chamadaFuncao:** criada para suportar funções comargumentos: suporte para chamadas como read(), fact(n) e leitura de strings.

Esta informação está toda disponível e comentada corretamente no ficheiro MOC.g4.

Funcionamento do Parser:

O ponto de entrada do parser é a regra programa, que é responsável pela definição da estrutura principal da linguagem definida por esta gramática. Esta regra obriga a que o programa seja composto por dois elementos obrigatórios, um dos elementos é o protótipo e o outro o corpo, terminando com o fim do ficheiro. A ideia de ter esta regra é garantir que os protótipos, sejam obrigatoriamente declaradas antes das funções.

Especificando um pouco o funcionamento dos protótipos, aqui a ideia foi garantir que, esteja garantido pelo menos o protótipo da função main, via prototipoPrincipal, podendo haver antes ou depois declarações dos outros protótipos.

Na regra corpo, podemos ter, nenhuma ou várias unidades, unidades essas que podem ser declarações de variáveis ou de definições de funções com corpo, porém esta regra obriga a que, mesmo que não tenhamos nenhuma unidade declarada, temos de ter a função principal, main.

Na definição dos protótipos e das funções identificamos que parametros estes podem receber, feito com recurso à regra parametros. Com esta regra conseguimos definir os vários tipos de cabeçalhos que podemos ter, ou seja, podemos ter funções que não têm argumentos, ou ter por exemplo a possibilidade de passar vetores como argumentos. A regra parâmetro pode suportar parametros com ou sem identificadores assim como vetores, enquanto a regra tipo, tal como o nome sugere, restringe os tipos suportados pela linguagem, int, double e void.

Na definição das funções é também necessário prever a declaração das variáveis no corpo do programa. Então com recurso à regra declaracao é permitido declarar uma ou mais variáveis dos tipos indicados acima. As variáveis são agrupadas numa listaVariaveis, onde cada elemento é definido pela regra variável, permitindo a flexibilidade para diferentes formas de declaração, podemos ter uma simples variável com identificador, até declarações que são inicializadas, vetores com ou sem valores atribuídos.

Para garantir que estas inicializações e outras operações sejam possíveis, é feita a analise das expressões para permitir a construção de instruções envolvendo operações aritméticas, lógicas, comparações, chamadas de função, acessos a vetores, e conversões de tipo (castings). Dividida por tipos para garantir a presidência e retirar a ambiguidade.

A base destas expressões é construída por elementos como literais numéricos, identificadores e estruturas como chamadas a funções ou acessos a vetores. Para permitir funções com múltiplos argumentos, a regra argumentos aceita uma lista de expressões separadas por vírgulas.

Para garantir as funções de leitura é feita com recurso á regra chamadaFuncao, que define chamadas sem argumentos e com sintaxe fixa.

Seguindo para as instruções que compõe o corpo das funções, começando na regra bloco, que impõe a obrigatoriedade de ter instruções dentro de chavetas. As instruções em si são organizadas por regras específicas, nomeadamente para garantir os ciclos if-else, while, for, etc. Foi feito um esforço para retirar a ambiguidade que é comum existir nos ciclos if-else, separando as instruções if emparelhadas e por emparelhar.

Durante o desenvolvimento da gramática, surgiram alguns desafios específicos que exigiram ajustes manuais, como por exemplo:

* Algumas palavras, como **main**, **read** e **writes**, causaram conflitos na gramática devido a serem identificadores válidos, e embora estivessem definidas como tokens, não eram reconhecidas corretamente pelo parser em algumas regras. Foram utilizadas literalmente nas regras do parser ('main', 'read', etc.), o que resolveu momentaneamente o nosso problema. Mais tarde, após uma análise teórica, compreendemos que a origem do conflito residia na posição incorreta dos tokens no lexer, o que gerava ambiguidade com outras regras, nomeadamente com o token **identificador**. A reorganização das declarações de tokens resolveu o problema de forma definitiva.
* As expressões v(i) (chamada de função) e v[i] (acesso a vetor) tinham sido construídas inicialmente como uma única regra, e apresentavam ambiguidade sintáxica, foi necessário **separar explicitamente** em regras distintas (**chamadaGenerica**, **acessoVetor**), garantindo que o parser conseguisse distinguir corretamente cada forma.
* A regra original de **expressao** tratava apenas operações simples e diretas, mas não reconhecia corretamente estruturas mais complexas como: fact(n), v[i] e castings de (int) x, (double) y. Estas construções fazem parte da base da hierarquia das expressões e, por isso, a limitação residia na definição da regra primary, que representa os elementos terminais de uma expressão. Neste caso tivemos de fazer uma total reformulação da regra primary e as suas ramificações, e forma a suportar expressões compostas e encadeadas, garantindo assim que é respeitado a precedência dos operadores e eliminado ambiguidades.
* Existiu alguns problemas com os protótipos. Sabíamos que tínhamos de garantir a obrigatoriedade da declaração dos protótipos antes das funções e isso foi rapidamente conseguido, mas da forma como o estávamos a fazer obrigava a que o protótipo declarado tivesse obrigatoriamente a função declarada também. E no nosso entendimento, e se a linguagem MOCC é uma derivação da linguagem C, onde, mesmo que o protótipo seja declarado, a função não precisa de existir, com excepção da função main, faria sentido recriar essa situação na nossa gramática. Porém, as várias tentativas levavam sempre a falha sempre que a função não existisse, mas o protótipo sim. Depois de algumas tentativas conseguimos contornar esse problema.

Ver se queremos adicionar mais alguma coisa aqui

**Validação e testes**

**A nossa estratégia de testes teve como objetivo cobrir os vários aspetos possíveis do compilador, começando por validar cada um dos cenários de sucesso (happy path) da gramática identificados no enunciado (declarações e atribuições) de modo unitário para nos ajudar a validar durante a construção. De seguida avançámos para testes de integração, tendo como dados iniciais os exemplos de programas completos providenciados, focando-nos finalmente nos casos de limite e falha.**

**Nos Anexos podem encontrar um resumo mais completo dos testes concluídos.**

**Conclusão e comentários**

**Se quisermos adicionar aqui a informação do nosso Visitor ou Listener , etc..**

Andreia Romão (1702430)

Cátia Santos (1702194)

**Referências  
Adicionar chatgpt**

**Anexos**

* **Casos Validados com Sucesso**

**Adicionar em formato tabela resumo (metemos depois em anexo os testes)**

* **Casos de Erro Intencional**

**Adicionar em formato tabela resumo (metemos depois em anexo os testes)**