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

Este relatório descreve o desenvolvimento de um compilador para uma linguagem fictícia no contexto da UC 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**, aqui segue o [link](https://elearning.uab.pt/pluginfile.php/3918150/mod_assign/introattachment/0/MOCC.pdf?forcedownload=1) do enunciado para contextualização. Sendo o foco desta atividade, a análise léxica e sintática, foram gerados os seguintes ficheiros:

* **main.py** – Ficheiro principal onde estão os comandos antlr4 necessários para compilar e verificar o código com as nossas customizações.
* **MOC.g4** – Ficheiro de gramática, inclui regras léxicas(tokens) e sintáticas(parser).
* **MOCErrorListener.py** – Ficheiro de erros customizáveis para lexer e parser.
* **MOCVisitorDEBUG.py** – Ficheiro Visitor customizado (ver comentários na conclusão).
* **reset\_antlr.sh** – Script auxiliar para limpar o ambiente de antlr4 e voltar a recriar. Usado cada vez que temos de modificar a gramática.
* **README.md** – Instruções de uso e informação técnica do programa.

**Gramática**

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

* **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 de regras de parser que definem se a estrutura dos programas é válida:

* **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.

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 determina 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. Relativamente aos protótipos, a ideia foi garantir que, exista pelo menos sempre o protótipo da função *main*, via prototipoPrincipal, podendo existir antes ou depois das declarações dos outros protótipos.

Na regra corpo, podemos ter, nenhuma ou várias unidades, que podem ser declarações de variáveis ou de definições de funções com corpo, no entanto, mesmo sem nenhuma unidade declarada, temos de ter a função principal, *main*, sempre presente.

Na definição dos protótipos e das funções identificamos os parametros que podem receber, através da regra **parâmetros**. Permite funções com ou sem argumentos e suporta parâmetros com ou sem identificadores, assim como vetores. Adicionalmente temos a regra **tipo**, que restringe os tipos suportados pela linguagem, *int*, *double* e *void*.

Devido á declaração de variáveis no corpo do programa para as funções existe também a regra **declaracao** que define que permite uma ou mais variáveis dos tipos indicados acima. As variáveis são agrupadas numa *listaVariaveis*, onde cada elemento é definido pela regra **variável**, permitindo maior flexibilidade para diferentes formas de declaração.

De seguida com a regra **expressão** é feita a análise que permitir a construção de instruções envolvendo operações aritméticas, lógicas, comparações, chamadas de função, acessos a vetores, e conversões de tipo (castings), dividida pelos tipos de forma a lidar com ambiguidade. Para permitir funções com múltiplos argumentos, a regra **argumentos** aceita uma lista de expressões separadas por vírgulas.

As funções de leitura utilizam a regra **chamadaFuncao**, que define chamadas sem argumentos e com sintaxe fixa.

Finalmente as instruções estão organizadas por tipo de estrutura, com regras especificas para tipos de condicionais e instruções emparelhadas e por emparelhar para melhor lidar com as especificidades de cada caso.

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 por serem identificadores válidos, e embora estivessem definidas como tokens, não eram reconhecidas corretamente pelo parser em algumas regras.Após uma análise teórica, compreendemos que a origem do conflito residia na posição incorreta dos tokens no lexer, o que gerava ambiguidade, nomeadamente com o token **identificador**. Foi resolvido com a reorganização das declarações de tokens.
* As expressões v(i) (chamada de função) e v[i] (acesso a vetor) tinham sido construídas inicialmente como uma única regra, que apresentava 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, 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, havia uma limitação na definição da regra primary, que representa os elementos terminais de uma expressão. Neste caso houve uma total reformulação da regra primary e as suas ramificações, de forma a suportar expressões compostas e encadeadas, garantindo assim que é respeitado a precedência dos operadores e eliminando ambiguidades.
* Os protótipos foram um desafio, a obrigatoriedade da declaração dos protótipos antes das funções foi logo implementada, mas da forma como estávamos a defini-la, obrigava a que o protótipo declarado tivesse obrigatoriamente a função declarada também. Sendo a linguagem MOCC uma derivação da linguagem C, onde, mesmo que o protótipo seja declarado, a função não precisa de existir (sendo exceção a main), quisemos emular esse comportamento, mas demorámos a chegar a uma combinação certa para recriar este comportamento.
* Depois de considerar vários casos para a indentação, e após alguma reflexão sobre as regras da linguagem C, concluímos que não nos teríamos de preocupar com a indentação. Ao contrário do Python, em C podemos colocar o código todo numa única linha, mesmo não sendo o mais indicado, não existe essa limitação. Não havendo nenhum requerimento específico sobre isto no enunciado, a indentação não foi considerada para ser analisada pela gramática.

**Listener**

**Para melhor a perceção dos erros e ajudar o utilizador, investimos algum tempo na criação de um Listener customizado, onde tratamos os erros do Lexer e Parser, de forma a gerar mensagens mais claras e limpas. Os erros foram divididos em erros de lexer/token – Erros Léxicos, e erros de parser – Erros Sintáticos, exemplos podem ser encontrados, em anexo.**

Os erros estão a ser dados por ordem cronológica de eventos, ou seja, linha a linha, coluna a coluna, de forma a ser os mais realista possível a validação do programa.

Existem ainda aspetos a melhorar, mas pensamos que tenhamos conseguido mapear todos os erros mais frequentes.

**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**

**De mencionar a quantidade de tempo perdido pela dificuldade de perceção do enunciado, a criar um Visitor que acabámos por não utilizar no nosso programa e também tempo de estudo em árvores sintáticas abstratas (AST) quando apenas era necessária uma árvore simples, num semestre onde o tempo é bastante escasso, foi um pouco frustrante. A parte positiva, foi a ajuda que o Visitor nos deu a encontrar erros na nossa gramática, utilizando o mesmo para debugging passo a passo, para apanhar as falhas e corrigi-las.**

**É certo de que teremos melhorias a fazer, talvez na simplificação de algumas regras na gramática que possam ajudar com o trabalho seguinte, mas tendo em conta as adversidades e tempo disponível, estamos satisfeitas com o trabalho apresentado**

Andreia Romão (1702430)

Cátia Santos (1702194)

**Referências**

* **Universidade Aberta. (n.d.). Fórum da unidade curricular Compilação [Fórum Moodle]. Acedido através da plataforma Moodle da UAb.**
* **OpenAI. (2025). ChatGPT (versão utilizada para apoio ao desenvolvimento e validação do código).**
* **Aho, A. V., Lam, M. S., Sethi, R., & Ullman, J. D. (n.d.). Compiladores: Princípios, Técnicas e Ferramentas**
* ANTLR Documentation. (n.d.). *ANTLR Documentation*. Disponível em <https://github.com/antlr/antlr4/blob/master/doc>
* Tomassetti, F. (n.d.). *The ANTLR Mega Tutorial*. Disponível em <https://tomassetti.me/antlr-mega-tutorial/>

**Anexos**

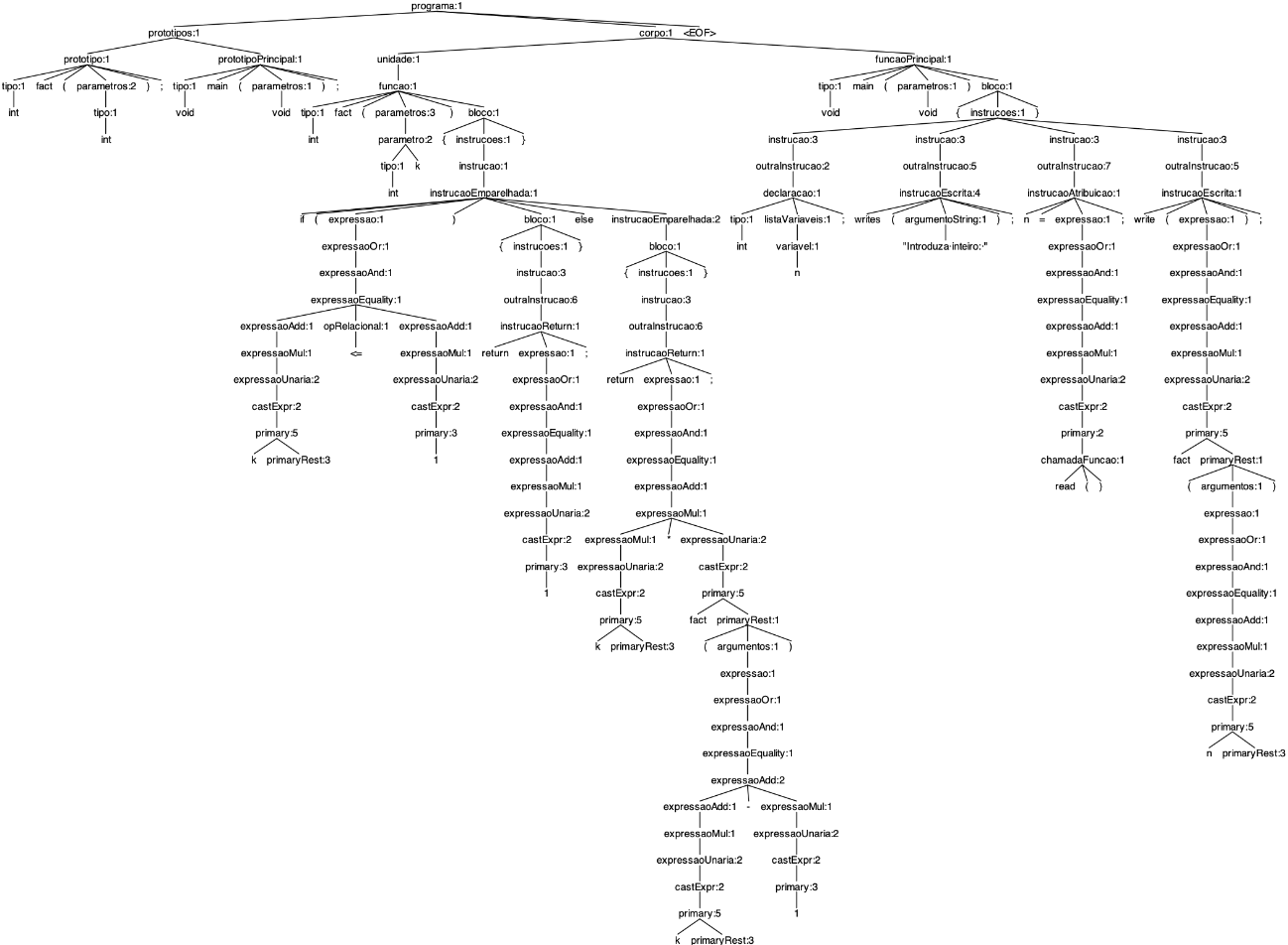
**Em todos os testes são corridos os comandos do antlr para verem os erros da gramática, e o nosso programa, para compararem com a nossa adaptação do Listener.**

**Por motivos de compatibilidade e possíveis diferenças de comportamento entre versões, recomendamos ANTLR 4.13.2 e Python 3.1X, com as quais todos os testes foram realizados.**

**Execução dos exemplos de sucesso de *chrome-extension://efaidnbmnnnibpcajpcglclefindmkaj/https://elearning.uab.pt/pluginfile.php/3959004/mod\_resource/content/1/caso\_sucesso.pdf***

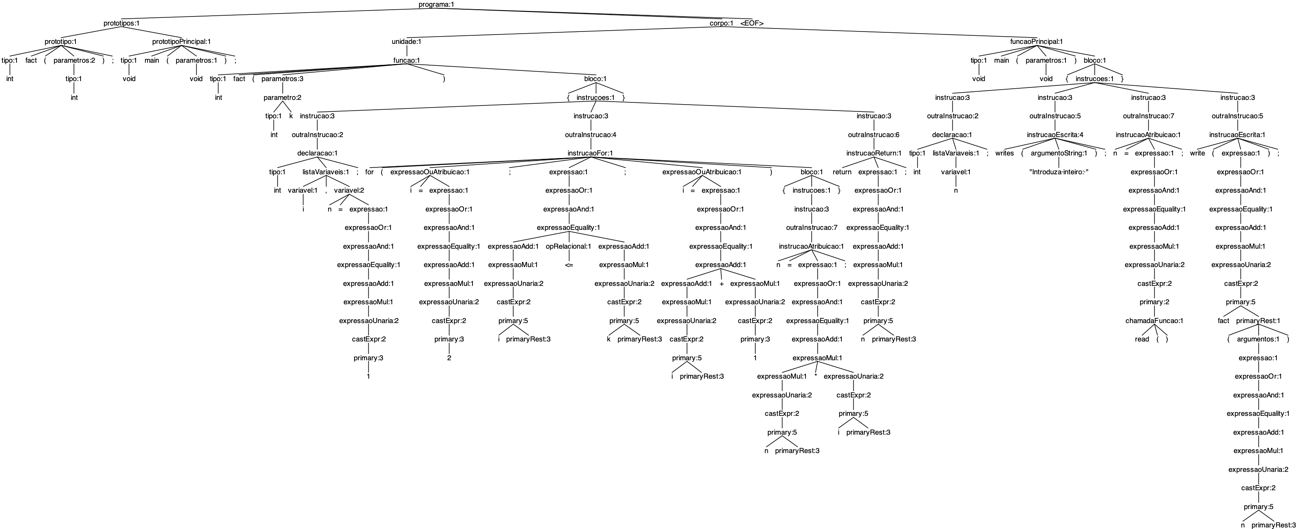
**Exemplo 1 – casos\_sucesso.pdf (corretamente formatado)**

|  |
| --- |
| /\* exemplo 1  fatorial versão recursiva \*/  int fact(int);  void main(void);  int fact(int k) {  if (k <= 1) {  return 1;  } else {  return k \* fact(k - 1);  }  }  void main(void) {  int n;  writes("Introduza inteiro: ");  n = read();  write(fact(n));  } |



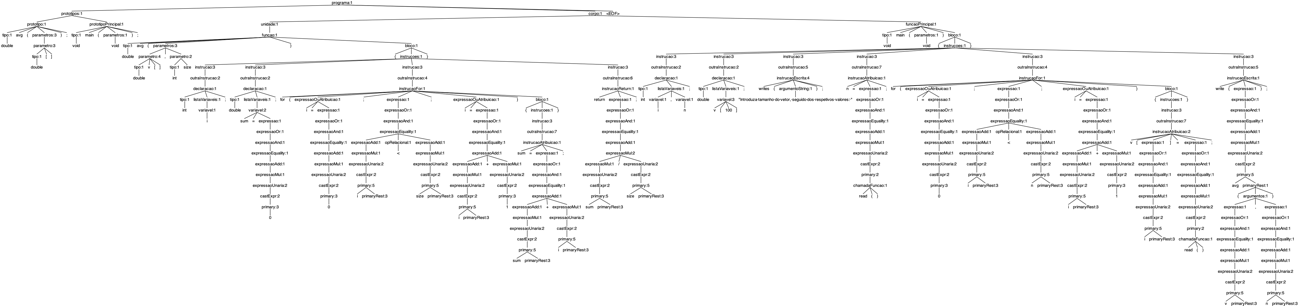
**Exemplo 2 – casos\_sucesso.pdf (corretamente formatado)**

|  |
| --- |
| /\* exemplo 2 \*/  int fact(int);  void main(void);  int fact(int k) {  int i, n = 1;  for (i = 2; i <= k; i = i + 1) {  n = n \* i;  }  return n;  }  void main(void) {  int n;  writes("Introduza inteiro: ");  n = read();  write(fact(n));  } |

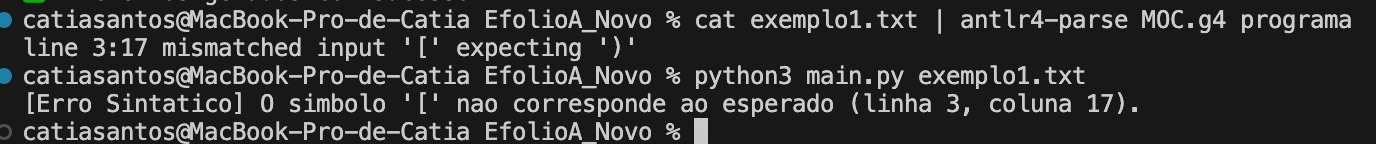


**Exemplo 3 – casos\_sucesso.pdf (corretamente formatado)**

|  |
| --- |
| /\* exemplo 3  média de uma lista de valores positivos \*/  double avg(double[]);  void main(void);  double avg(double v[], int size) {  int i;  double sum = 0;  for (i = 0; i < size; i = i + 1) {  sum = sum + i;  }  return sum / size;  }  void main(void) {  int i, n;  double v[100];  writes("Introduza tamanho do vetor, seguido dos respetivos valores: ");  n = read();  for (i = 0; i < n; i = i + 1) {  v[i] = read();  }  write(avg(v, n));  } |



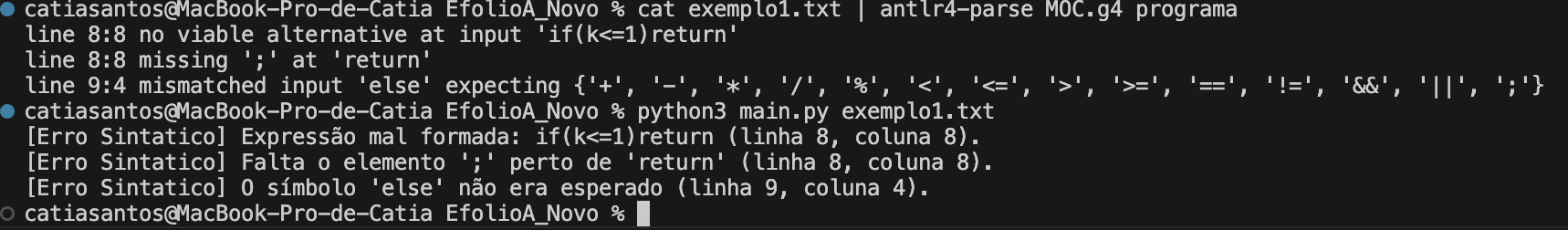
**Já agora a título de mostrar um pouco a funcionalidade do nosso ErrorListener e gramática, de acordo com o problema de protótipos que tínhamos mencionado para este exemplo no Moodle, caso não tivesse sido declaro como parâmetro, este seria o erro do parser vs o erro tratado pelo listener:**

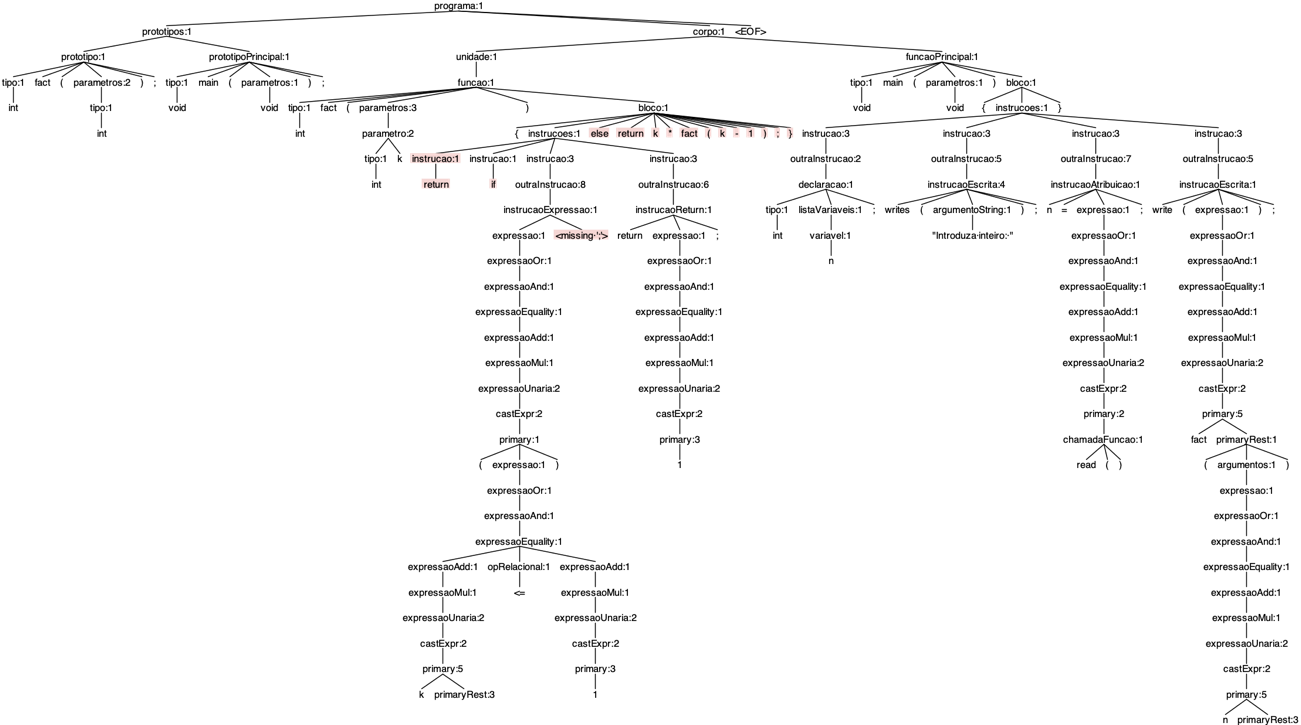


**E agora os testes iniciais do enunciado, testes de falha disponíveis em:** <https://elearning.uab.pt/pluginfile.php/3918150/mod_assign/introattachment/0/MOCC.pdf?forcedownload=1>

**Exemplo 1 – MOOC.pdf (corretamente formatado)**

|  |
| --- |
| /\* exemplo 1  fatorial versão recursiva \*/  int fact(int);  void main(void);  int fact(int k) {  if (k <= 1)  return 1;  else  return k \* fact(k - 1);  }  void main(void) {  int n;  writes("Introduza inteiro: ");  n = read();  write(fact(n));  } |

****



**O Exemplo 2 e Exemplo 3 do MOOC.pdf são os mesmos do casos\_sucesso.pdf , mais acima**

**Seguem também todos os testes unitários executados, divididos em tipo Sucesso (casos de sucesso) e Falha (casos de falha propositada, para ver casos limite), numa tabela com a informação resumida, seguida da execução e árvore para cada exemplo.**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **N.ª** | **Descrição** |  | **Tipo** | **Estado** |
| **1.1** | **Teste 1 – Diretivas Pré-processuais**  **Regra: Não pode haver diretivas (ex.: “#include”) no código.** | Sem diretivas | **Sucesso** | **OK** |
| **1.2** | **Contém diretiva proibida “**#include” | **Falha** | **OK** |
| **2.1** | **Teste 2 – Ordem – Protótipos Antes de Declarações**  **Regra: Todos os protótipos devem vir antes de qualquer declaração ou definição.** | Protótipos declarados no início | **Sucesso** | **OK** |
| **2.2** | Variável declarada antes dos protótipos | **Falha** | **OK** |
| **3.1** | **Teste 3 – Declaração de Variáveis: Tipos e Sintaxe**  **Regra: Apenas os tipos "int" e "double" e “void” são permitidos.** | **Uso de tipos permitidos (int, double e void)** | **Sucesso** | **OK** |
| **3.2** | **Uso de tipo não permitido: (float)** | **Falha** | **OK** |
| **4.1** | **Teste 4 – Inicialização de Variáveis**  **Regra: As variáveis podem ser inicializadas com expressões aritméticas ou com valores lidos.** | **Inicialização Aritmética e com leitura** | **Sucesso** | **OK** |
| **4.2** | **Erro de sintaxe: falta parênteses na chamada de read** | **Falha** | **OK** |
| **5.1** | **Teste 5 – Funções de Leitura e Escrita**  **Regra: As funções de leitura devem ser chamadas com parênteses: read(), readc(), reads().**  **Para escrita, usamos: write() para variáveis simples, writev() para vetores, e writes() para strings.** | **Uso correcto de todos os write()** | **Sucesso** | **OK** |
| **5.2** | **Faltam parênteses e aspas ausentes em string Usado o write() incorrecto para strings** | **Falha** | **OK** |
| **6.1** | **Teste 6 – Blocos de Instrução: Obrigatoriedade de Chavetas**  **Regra: Todos os blocos de instrução (mesmo com uma única instrução) devem estar entre chavetas.** | **Blocos de instrução com chavetas** | **Sucesso** | **OK** |
| **6.2** | **Falta de chavetas no condicional e de escrita** | **Falha** | **OK** |
| **7.1** | **Teste 7 – Ciclo While**  **Regra: O ciclo while deve ter a forma “while (expressão) { bloco }”.** | **Ciclo while bem definido** | **Sucesso** | **OK** |
| **7.2** | **Ciclo while com parenteses em falta** | **Falha** | **OK** |
| **8.1** | **Teste 8 – Ciclo For**  **Regra: O ciclo for deve ter a forma “for (inicialização; condição; atualização) { bloco }” e não pode usar operadores não permitidos (como i++).** | **Ciclo for bem definido** | **Sucesso** | **OK** |
| **8.2** | **Ciclo for com uso de i++** | **Falha** | **OK** |
| **9.1** | **Teste 9 – Condições Simplificadas**  **Regra: As condições devem ser simples (ou uma única expressão ou uma comparação simples “Expr op Expr”), sem composições ambíguas.** | **Condição simples de comparação** | **Sucesso** | **OK** |
| **9.2** | **Condição complexa com 3 variáveis** | **Falha** | **OK** |
| **10.1** | **Teste 10 – Função main com Protótipo**  **Regra: A função main (ponto de entrada) deve ter o seu protótipo declarado antes de sua definição.** | **Função main com protótipo declarado** | **Sucesso** | **OK** |
| **10.2** | **Função main sem protótipo declarado** | **Falha** | **OK** |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |

|  |
| --- |
| Exemplo de Sucesso 1.1:  int foo(int);  void main(void);  int foo(int a) { return a; }  void main(void) { }  Exemplo de Falha 1.2:  #include <stdio.h> // <-- Isto deve causar erro  int foo(int);  void main(void);  int foo(int a) { return a; }  void main(void) { } |

|  |
| --- |
| Exemplo de Sucesso 2.1:  int foo(int);  void main(void);  int m, n; // <-- Declaração de variáveis após os protótipos  int foo(int a) { return a; }  void main(void) { }  Exemplo de Falha 2.1:  int m, n; // <-- Não é permitido antes dos protótipos  int foo(int);  void main(void);  int foo(int a) { return a; }  void main(void) { } |

|  |
| --- |
| Exemplo de Sucesso 3.1:  int foo(int);  void main(void);  int a, b, v[10];  double x, y, z;  int foo(int a){ return a; }  void main(void){ }  Exemplo de Falha 3.2:  int foo(int);  void main(void);  float a; // <-- 'float' não está definido na gramática  int foo(int a){ return a; }  void main(void){ } |

|  |
| --- |
| Exemplo de Sucesso 4.1:  int foo(int);  void main(void);  int m = 1, n = 2 \* m;  double x = 3.14, y = x / 2;  int a = read();  int foo(int a){ return a; }  void main(void){ }  Exemplo de Falha 4.2:  int foo(int);  void main(void);  int m = 1, n = 2 \* m;  double x = 3.14, y = x / 2;  int a = read; // <-- Falta os parênteses  int foo(int a){ return a; }  void main(void){ } |

|  |
| --- |
| Exemplo de Sucesso 5.1:  int foo(int);  void main(void);  int c = read();  int s[] = reads();  int foo(int a){ return a; }  void main(void){  int v[3] = {97,98,99};  write(v[0]); // para número  writec(v[0]); // para carácter  writev(v); // para vetor  writes("abc"); // para string  }  Exemplo de Falha 5.2:  int foo(int);  void main(void);  // Uso incorreto: faltam parênteses e aspas ausentes em string  int c = read; // <-- Falta parênteses  int s[] = reads; // <-- Falta parênteses  int foo(int a){ return a; }  void main(void){  int v[3] = {97,98,99};  write("abc"); // <-- Para escrever string, deve usar writes()  } |

|  |
| --- |
| Exemplo de Sucesso 6.1:  int foo(int);  void main(void);  int foo(int a){  if(a < 0) {  return b;  } else {  return a;  }  }  void main(void){  if(1 < 2) {  writes("ok");  }  }  Exemplo de Falha 6.2:  int foo(int);  void main(void);  int foo(int a){  if(a < 0)  return b; // <-- Falta chavetas no if  else  return a;  }  void main(void){  if(1 < 2)  writes("ok"); // <-- Falta chavetas  } |

|  |
| --- |
| Exemplo de Sucesso 7.1:  int foo(int);  void main(void);  int foo(int a){ return a; }  void main(void){  int x = 10;  while(x > 0) {  x = x - 1;  }  }  Exemplo de Falha 7.2:  int foo(int);  void main(void);  int foo(int a){ return a; }  void main(void){  int x = 10;  while x > 0 { // <-- Faltam os parênteses  x = x - 1;  }  } |

|  |
| --- |
| Exemplo de Sucesso 8.1:  int foo(int);  void main(void);  int foo(int a){ return a; }  void main(void){  int i, s;  for(i = 0; i < 10; i = i + 1) {  s = s + i;  }  }  Exemplo de Falha 8.2:  int foo(int);  void main(void);  int foo(int a){ return a; }  void main(void){  int i, s;  for(i = 0; i < 10; i++) { // <-- Uso de i++ ou i++ shorthand não é permitido  s = s + i;  }  } |

|  |
| --- |
| Exemplo de Sucesso 9.1:  int foo(int);  void main(void);  int foo(int a){ return a; }  void main(void){  int x = 5, y = 10;  if (x < y) {  writes("ok");  }  }  Exemplo de Falha 9.2:  int foo(int);  void main(void);  int foo(int a){ return a; }  void main(void){  int x = 5, y = 10, z = 15;  if (x < y < z) { // <-- Expressão "x < y < z" não é permitida  writes("erro");  }  } |

|  |
| --- |
| Exemplo de Sucesso:  int fact(int);  void main(void);  int fact(int k) {  if (k <= 1) {  return 1;  } else {  return k \* fact(k - 1);  }  }  void main(void) {  int n;  writes("Introduza inteiro: ");  n = read();  write(fact(n));  }  Exemplo de Falha:  int fact(int);  // --> Falta o protótipo de main    int fact(int k) {  if (k <= 1) {  return 1;  } else {  return k \* fact(k - 1);  }  }  void main(void) { // main definida, mas sem protótipo  int n;  writes("Introduza inteiro: ");  n = read();  write(fact(n));  } |