

**UNIDADE CURRICULAR: Compilação**

**CÓDIGO: 21018**

**DOCENTE: Constantino Martins**

**TUTOR: Rúdi Gualter**

**Trabalho realizado pelos alunos (grupo: QUALQUER TOKEN):**

**Nome:** Andreia Romão – **Nº Estudante:** 1702430

**Nome:** Cátia Santos – **Nº Estudante:** 1702194

**Nome:** Rui Menino – **Nº Estudante:** 1103425

**Nome:** Luís Tavares – **Nº Estudante:** 1803237

**Nome:** José Augusto Azevedo – **Nº Estudante:** 2200655

**CURSO:** Licenciatura em Engenharia Informática

**DATA DE ENTREGA:** 14 de Maio de 2025

**TRABALHO / RESOLUÇÃO:**

**Grupo Qualquer Token:**

1702430 - Andreia Romão

1702194 - Cátia Santos

2200655 - José Augusto Azevedo

1803237 - Luís Tavares

1103425 - Rui Menino

O presente relatório documenta o progresso e a implementação relativos ao desenvolvimento de um compilador para a linguagem MOC, dando seguimento ao trabalho iniciado por ambos os grupos Qualquer e Token no efólio A da UC de Compilação. Nessa fase inicial do projeto, foi efectuada a análise léxica e sintática da linguagem MOC, onde se construiu uma gramática formal da linguagem MOC, destinada a ser utilizada pelo analisador sintático (Parser) e onde os grupos desenvolveram as componentes de frontend (lexer e parser) necessárias para validar a estrutura de um programa MOC, utilizando ANTRL4.

Nesta nova fase, correspondente ao efólio B, o grupo Qualquer e o grupo Token, decidiram juntar-se para continuar na criação de um compilador para a linguagem MOC. Efectuamos um processo de análise e avaliação comparativa dos trabalhos desenvolvidos por ambos os grupos no E-fólio A. O objetivo era identificar a base de trabalho mais sólida e promissora para servir como fundação para as fases subsequentes e mais complexas da implementação do compilador.

Após ponderação conjunta das soluções de análise léxica e sintática apresentadas por cada grupo, foi tomada a decisão de prosseguir o desenvolvimento utilizando como ponto de partida o projecto previamente elaborado pelo "Grupo Qualquer". Esta escolha estratégica visou capitalizar sobre a abordagem considerada mais avançada para facilitar a implementação das fases seguintes. Deste modo, o trabalho descrito no presente documento detalha a continuação da implementação do compilador MOC, utilizando Python, focando-se agora em três componentes cruciais do compilador: a análise semântica, a geração de código intermédio, a gestão da tabela de símbolos e a optimização de código intermédio.

O desenvolvimento foi modularizado em ficheiros distintos (VisitorSemantico.py, VisitorTAC.py, OtimizadorTAC.py), cujas implementações e funcionalidades são descritas nas secções seguintes. O objetivo final desta fase foi transformar a representação sintática de um programa MOC válido num código intermédio optimizado, preparando o caminho para a geração de código final.

**Análise Semântica (VisitorSemantico.py)**

A análise semântica constitui a primeira etapa após a validação sintática, sendo responsável por verificar a coerência e o significado do programa MOC, para além da sua estrutura gramatical. O objetivo principal deste visitor é percorrer a árvore de análise sintática (Parse Tree) produzida pelo parser e realizar verificações de consistência e significado que vão além da estrutura gramatical, identificando erros semânticos comuns. Para tal, o visitor mantém um estado interno, nomeadamente uma estrutura para gestão de âmbitos(scopes) e na validação da declaração e uso correcto dos identificadores (variáveis e funções).

A gestão de âmbito é realizada através de uma lista (*self.escopos*) que funciona como uma pilha, onde cada elemento da lista é um conjunto (set) contendo os nomes dos identificadores declarados nesse âmbito específico. Um novo âmbito é empilhado (append) ao entrar numa definição de função (visitFuncao, visitFuncaoPrincipal), e os nomes dos parâmetros são imediatamente adicionados a esse novo âmbito. Ao visitar uma declaração de variáveis locais (visitDeclaracao), o visitor verifica primeiro se o nome já existe no âmbito atual (topo da pilha, self.escopos[-1]) para detetar declarações duplicadas; caso contrário, adiciona o novo identificador a esse âmbito. Crucialmente, ao terminar a visita de uma função, o âmbito correspondente é removido da pilha (pop), garantindo a correta gestão dos âmbitos léxicos.

As principais verificações semânticas implementadas centram-se na declaração antes da utilização. Ao encontrar um identificador em diferentes contextos de uso – como no lado esquerdo de uma atribuição (visitInstrucaoAtribuicao), num acesso a vetor (visitAcessoVetor), como argumento de WRITEV (visitInstrucaoEscrita), ou genericamente numa expressão (visitIdComPrefixo) – o visitor percorre a pilha de âmbitos (reversed(self.escopos)) para garantir que esse identificador foi declarado num âmbito acessível (o atual ou um superior). Se um identificador não for encontrado, uma exceção de erro semântico é levantada. O visitor também mantém um conjunto (self.funcoes\_declaradas) com os nomes das funções declaradas através de protótipos (visitPrototipo, visitPrototipoPrincipal), permitindo que visitIdComPrefixo distinga entre uma variável não declarada e uma chamada a uma função (embora não valide a chamada em si).

Para além destas verificações explícitas, o VisitorSemantico assegura a travessia recursiva das restantes estruturas da linguagem (expressões, blocos, estruturas de controlo if/else, while, for, return, etc.), garantindo que as validações de uso de identificadores são aplicadas em todos os contextos apropriados.

**No entanto, é importante notar as limitações da presente implementação: não realiza verificações de tipo (ex: compatibilidade de tipos em atribuições, operações aritméticas/lógicas, ou correspondência entre argumentos e parâmetros de funções), focando-se exclusivamente na gestão de nomes e âmbitos. Adicionalmente, o tratamento de erros atual baseia-se no lançamento de exceções (raise Exception), o que interrompe a análise no primeiro erro semântico encontrado, em vez de acumular uma lista de todos os erros.**

**3. Geração de Código Intermédio (VisitorTAC.py)**

Após a validação semântica do programa MOC, a fase seguinte consiste na tradução da árvore sintática (agora semanticamente correta) para uma representação intermédia mais próxima da máquina, mas ainda independente da arquitetura final. Para este projeto, optou-se pelo Código de Três Endereços (Three-Address Code - TAC), representado como uma sequência de quádruplas.

Foi implementado um outro Visitor (VisitorTAC.py). Este visitor percorre novamente a árvore de análise sintática, utilizando a informação previamente recolhida e validada para gerar a sequência de instruções TAC. A principal responsabilidade deste visitor é traduzir as construções sintáticas da linguagem MOC para uma representação mais próxima da máquina, mas ainda independente da arquitetura final, é utilizado o código de três endereços (Three-Address Code - TAC), especificamente num formato de quádruplas para a representação intermédia da linguagem.

As instruções são geradas num formato de quádruplas (operador, argumento1, argumento2, resultado). Internamente, o VisitorTAC gere uma lista (*tac\_quadruplos*) onde armazena a sequência de instruções TAC geradas. Cada instrução é representada como um dicionário com chaves "op", "arg1", "arg2", e "res". A classe também utiliza contadores para gerar nomes únicos para variáveis temporárias (*novo\_temp*) e rótulos (*nova\_label*), essenciais para representar resultados intermédios de expressões e para implementar o fluxo de controlo. Um método auxiliar *adicionar\_quadruplo* facilita a criação destas quádruplas, gerando automaticamente temporários para o resultado quando necessário. O acesso a informações semânticas, como variáveis declaradas, é feito através de estruturas herdadas ou mantidas pela própria classe.

A tradução das construções MOC para TAC é realizada através da sobrescrita dos métodos *visitNomeDaRegra*. As expressões aritméticas, lógicas e relacionais são decompostas em quádruplas simples, usando temporários para resultados intermédios. As atribuições são convertidas em instruções *ASSIGN* ou *[]=* (para vetores, incluindo cálculo de offset). Estruturas de controlo como *if-else*, *while* e *for* são implementadas usando rótulos e instruções de salto condicional (*ifFalse*) e incondicional (*goto*). As chamadas a funções geram instruções *param* para os argumentos e *call* para a invocação, enquanto as funções de entrada/saída (*read, write, etc*.) são mapeadas para quádruplas específicas. É utilizada uma função utilitária, *gerar\_texto\_tac*, para formatar a lista de quádruplas numa representação textual legível. No final do processo de visita, a lista *tac\_quadruplos* contém a representação completa do programa MOC em Código de Três Endereços, pronta para a fase de otimização.

**4. Optimização do Código Intermédio (OtimizadorTAC.py)**

Com o código intermédio (TAC) gerado, foi implementado um módulo de otimização (OtimizadorTAC.py) com o objetivo de melhorar a eficiência do código antes da geração final, sem alterar a sua semântica. O optimizador recebe a lista de quádruplas TAC gerada na fase anterior e aplica um conjunto de técnicas, tornando o código potencialmente mais rápido e/ou mais curto, antes de prosseguir para a geração de código final.

Ao ser instanciada, a classe OtimizadorTAC criamos uma cópia profunda (*copy.deepcopy*) da lista de quádruplas recebida, garantindo assim que as optimizações não modificam a lista original. Também é possível definir um conjunto opcional *variaveis\_utilizador*, que especifica variáveis cujo valor deve ser preservado mesmo que a análise indique que não são usadas posteriormente (útil para resultados finais ou variáveis com significado externo). Na classe implementamos vários métodos, cada um correspondendo a uma técnica de optimização específica.

As principais optimizações implementadas são:

* **Eliminação de Código Morto** (*eliminar\_codigo\_morto*): Esta técnica utiliza uma análise de vivacidade (liveness analysis) percorrendo os quádruplos de trás para a frente. Identificamos as variáveis cujo valor é efetivamente utilizado mais tarde ("*variáveis vivas*") e removemos as instruções cujo resultado (variável de destino *res*) não está no conjunto de variáveis vivas no ponto seguinte e que não possuem efeitos colaterais observáveis (como chamadas a funções, operações de I/O, ou manipulação de memória/arrays). Instruções sem resultado (labels, gotos) ou com efeitos colaterais são preservadas.
* **Propagação de Cópias** (*propagacao\_copias*): Analisamos as instruções de atribuição simples (ex: *t2 = t1*). Se uma variável (*t2*) recebe o valor de outra (*t1*) e essa variável de destino (*t2*) não é redefinida posteriormente de forma complexa (ex: dentro de ciclos ou múltiplas vezes), o optimizador substitui as utilizações subsequentes de *t2* diretamente por *t1*, eliminando a cópia redundante.
* **Dobramento de Constantes** (*constant\_folding*): Procuramos por operações aritméticas cujos operandos são ambos constantes numéricas conhecidas em tempo de compilação (ex: *t1 = 3 + 5*). Avaliamos essas operações e substituimos a instrução original por uma atribuição direta do resultado calculado (ex: *t1 = 8*).
* **Eliminação de Subexpressões Comuns** (*eliminar\_subexpressoes\_comuns\_CSE*): Mantémos um registo das expressões já calculadas e das versões actuais das variáveis. Se encontramos uma expressão (ex: *a + b*) que já foi calculada anteriormente com os operandos a e b tendo os mesmos valores (mesma versão), em vez de gerar o código para recalcular, reutilizamos o resultado anterior (guardado num temporário), substituindo a computação redundante por uma cópia. Incluimos normalização para operações comutativas.
* **Eliminação de Código Inatingível** (*eliminar\_codigo\_inatingivel*): Removemos sequências de instruções que nunca podem ser executadas porque seguem imediatamente uma instrução de salto incondicional (goto) ou um return, e não são o alvo de nenhuma label.
* **Movimento de Código Invariante de Ciclos** (*mover\_invariantes*): Identificamos loops (através de saltos para trás - back-edges). Dentro de cada loop, procuramos por instruções cujos operandos são definidos fora do loop (ou são constantes). Estas instruções "invariantes" são então movidas para imediatamente antes do ponto de entrada do loop, evitando que sejam recalculadas desnecessariamente a cada iteração.

Para orquestrar a aplicação destas técnicas, é fornecida a função *otimizar\_completo*. Esta função instancia o OtimizadorTAC e aplica as otimizações numa ordem específica, incluindo a execução repetida da eliminação de código morto até que nenhuma alteração adicional seja possível (atingindo um ponto fixo), pois outras otimizações podem gerar novo código morto. O resultado final desta função é a lista de quádruplas TAC otimizada.

Lista das Especificações da Linguagem MOC

1. **Comentários -** Delimitadores: /\* (início) e \*/ (fim).
2. **Estrutura do Programa**
   1. **Protótipos de funções**: Devem ser declarados antes de qualquer função ou variável.
   2. **Função main()**: Ponto de entrada obrigatório.
3. **Blocos de Código -** Delimitados por {}, mesmo para blocos com uma única instrução
4. **Instruções e Operadores**
   1. **Terminação**: Todas as instruções terminam com ;.
   2. **Operadores**:
      1. Aritméticos: +, -, \*, /, %.
      2. Relacionais: ==, !=, >, <, >=, <=.
      3. Lógicos: && (E), || (OU), ! (NÃO).
      4. Atribuição: =.
   3. **Condições**: Formato restrito a Expr ou Expr OpCond Expr (e.g., x > 5 && y != 0).
5. **Estruturas de Controle**
   1. **Condicionais**:
      1. if (condicao){…} ou if(condicao){…}else{…}
      2. switch/case.
   2. **Loops**:
      1. for
      2. while.
6. **Declaração de Funções**
   1. Formato: tipo\_retorno nome(parâmetros) { ... }.
   2. Tipos de retorno: int, double, void, ou ausente (void implícito).
7. **Tipos de Dados**
   1. **Básicos**: int (inteiros), double (ponto flutuante).
   2. **Vetores**: Arrays de int ou double (e.g., int v[] = {1, 2, 3};).
   3. **Strings**: Vetores de int terminados em 0 (ASCII).
8. **Variáveis**
   1. **Declaração**:
      1. Sem inicialização: Valor padrão 0.
      2. Com inicialização: Usando expressões aritméticas (e.g., int c = 2 \* b;).
      3. Vetores: Tamanho automático se inicializados (e.g., int v[] = {1, 2};).
   2. **Âmbito**: Variáveis devem ser declaradas antes do uso.
9. **Entrada/Saída**
   1. **Entrada**:
      1. read(): Lê int ou double.
      2. readc(): Lê caracter (retorna valor ASCII).
      3. reads(): Lê string para vetor de int (termina em 0).
   2. **Saída**:
      1. write(x): Imprime valor de variável.
      2. writec(x): Imprime caracter (ASCII).
      3. writev(vetor): Imprime vetor no formato {1, 2, 0}.
      4. writes("texto"): Imprime string (com \n ao final).
10. **Conversão de Tipos**
    1. **Implícita**: int → double em operações mistas.
    2. **Explícita**: Usar (int) ou (double) (e.g., (int) 3.14 → 3).
11. **Regras Adicionais**
    1. **Strings Literais**: Usadas diretamente no writes("Olá").
    2. **Vetores como Strings**: int s[] = reads(); armazena códigos ASCII + 0.
    3. **Erros**: Uso de variáveis não declaradas é inválido.

**Bibliografia/Referências:**

* **Compilers: principles, techniques and tools, 2nd Ed., Aho, Lam, Setti, Ullman, Addison-Wesley, 2007**
* **Compiladores – Da Teoria à Prática, Pedro Reis Santos e Thibault Langlois. FCA, 2015.**
* **The ANTLR Mega Tutorial:** [**https://tomassetti.me/antlr-mega-tutorial/**](https://tomassetti.me/antlr-mega-tutorial/)
* **ANTLR Doc:** [**https://github.com/antlr/antlr4/blob/master/doc/index.md**](https://github.com/antlr/antlr4/blob/master/doc/index.md)
* **CD | INTRODUCTION | INTRODUCTION AND VARIOUS PHASES OF COMPILER | RAVINDRABABU RAVULA**

<https://youtu.be/Qkwj65l_96I?list=PL5UbMb0H_A9hs6Z_myVW_tqRpFipkzniD>

* **EECS4302 ANTLR4 PARSER GENERATOR TUTORIAL**

<https://youtu.be/-FdD_xzNFL4?list=PL5UbMb0H_A9hs6Z_myVW_tqRpFipkzniD>

* **COMPILADORES DE JUDSON SANTIAGO**

<https://www.youtube.com/playlist?list=PLX6Nyaq0ebfhI396WlWN6WlBm-tp7vDtV>

* **COMPILADORES DE PROF. JOSÉ RUI**

<https://www.youtube.com/playlist?list=PLqlIQgAFrQ14VmHe8VbIVUkBv5Hziv86->

* **COMPILADORES (CC3001) — 2022/2023 — Professor Pedro Vasconcelos, 2022.**

**Faculdade de Ciências da Universidade do Porto**

<https://www.dcc.fc.up.pt/~pbv/aulas/compiladores/teoricas/>