

**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:** 19 de Maio de 2025

**TRABALHO / RESOLUÇÃO:**

O presente relatório documenta o progresso e a implementação relativos ao desenvolvimento de um compilador para a linguagem MOC. Previamente, ambos os grupos Qualquer e Token no E-fólio A da UC de Compilação tinham efetuado a implementação da linguagem MOC, através da construção de uma gramática formal MOC.g4 utilizada pelo analisador sintático (parser) e léxico (lexer). Tendo também sido desenvolvidos componentes de apoio, como um listener customizado de formar a ajudar a validar a estrutura de um programa utilizando ANTRL4.

Nesta nova fase, correspondente ao E-fólio B, o grupo Qualquer e o grupo Token, decidiram juntar-se para continuar a expandir este mesmo compilador. Como tal, foi iniciado um processo de análise e avaliação de ambos os trabalhos desenvolvidos que tinha como objetivo com a identificação do candidato mais indicado, que iria servir de base 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 projeto 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 quatro componentes cruciais do compilador: a análise semântica, a geração de código intermédio, a gestão da tabela de símbolos e a otimização de código intermédio.

O desenvolvimento foi modularizado em ficheiros distintos (VisitorSemantico.py, VisitorTAC.py, TabelaSimbolos.py e 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 otimizado, preparando o caminho para a geração de código final.

**Análise Semântica (VisitorSemantico.py)**

A análise semântica é a etapa após a validação sintática, sendo responsável por verificar a coerência lógica e o significado correto do programa, indo além da verificação de estrutura gramatical. O objetivo principal deste visitor é percorrer a árvore de análise sintática (Parse Tree) produzida pelo parser e aplicar regras de consistência que vão além da estrutura gramatical, identificando erros semânticos comuns. Para tal, o visitor mantém um estado interno durante a análise, nomeadamente uma pilha de contextos léxicos, para a gestão de contextos(scopes) de forma a validar situações tais como, a declaração e uso correto dos identificadores (variáveis e funções) e compatibilidade de tipos.

A gestão de contextos léxicos é realizada através de uma lista (*self.* contexto) que funciona como uma pilha, onde cada elemento da lista representa um contexto local – um conjunto (*set*) que contém os nomes dos identificadores declarados nesse contexto específico. Um novo contexto é empilhado (*append*) ao iniciar a definição de uma função (*visitFuncao*, *visitFuncaoPrincipal*), e os nomes dos parâmetros são imediatamente adicionados a esse novo contexto. Durante a visita a declarações de variáveis locais (*visitDeclaracao*), é verificado se o nome já existe no contexto atual (topo da pilha, *self.contexto[-1])* para detetar duplicações. Não existindo conflito, o novo identificador é inserido nesse contexto. Para finalizar, a análise da função, o contexto correspondente é removido da pilha (*pop*), assegurando a correta delimitação dos contexto léxicos e a integridade da nossa análise semântica.

As principais verificações semânticas implementadas focam-se na garantia de que todos os identificadores são declarados antes de utilizados. Sempre que um identificador é encontrado em diferentes contextos de uso – como no lado esquerdo de uma atribuição (*visitInstrucaoAtribuicao*), num acesso a vetor (*visitAcessoVetor*), como argumento de escrita (*visitInstrucaoEscrita*), ou expressões genéricas (*visitIdComPrefixo*) – o visitor percorre a pilha de contextos pela ordem reversa (*reversed(self.contexto*)) de forma a garantir que esse identificador foi previamente declarado num contexto acessível (o actual ou um superior). Se um identificador não for encontrado, uma excepção de erro semântico é levantada. Adicionalmente, o visitor mantém um conjunto (*self.funcoes\_declaradas*) que contém os nomes das funções declaradas através de protótipos (*visitPrototipo*, *visitPrototipoPrincipal*), permitindo distinguir entre uma variável não declarada e uma chamada a uma função (*visitIdComPrefixo)* (embora não valide os argumentos das chamadas nesta fase).

Para além destas verificações explícitas, sobre declarações e uso de identificadores, o nosso *VisitorSemantico* percorre recursivamente as restantes estruturas da linguagem (expressões, blocos de código, instruções de controlo if/else, while, for, return, etc...), garantindo que as validações semânticas básicas são aplicadas em todos os contextos apropriados.

As acções semânticas implementadas são:

**Verificação de declarações**, relativamente às variáveis, verifica se os identificadores foram declarados antes do uso e verifica a redeclaração no mesmo contexto; relativamente às funções/protótipos, valida os tipos de retorno e os parâmetros entre os protótipos e as definições; (Figura 1 - Verificação de declarações - variáveis e Figura 2 - Verificação de declarações - Funções/Protótipos)

**Gestão de contextos**, verifica a entrada de saída de contextos, onde os blocos, funções e estruturas de controle criam novos contextos;

**Verificação de tipos**, nas atribuições é garantida a compatibilidade entre ambos os operadores; nas expressões os operandos são validados em operações aritméticas/lógicas; nas funções é verificado se o tipo de return corresponde ao declarado.

**Tratamento de estruturas de controle**; nas estruturas condicionais (if/else) é verificado se a condição é booleana; nos ciclos (while,for) são analisadas as expressões de término e inicialização.

**Arrays**, nas declarações de vectores, validação das dimensões e inicializações; no acesso a índices, é impedido o acesso a não-vectores.

**Funções**, nas chamadas de funções, é verificada a existência e paridade dos parâmetros; nos protótipos e definições é garantida a consistência entre declarações e implementações.

No VisitorSemantico implementamos uma análise semântica robusta, onde analisamos as declarações (variáveis, funções, arrays); os tipos (atribuições, expressões, retornos), os contextos (hierarquia de blocos) e o fluxo de controle (loops e condicionais).

**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.

**ANEXOS**

**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.

**Testes efectuados**

Uma imagem com texto, captura de ecrã, Tipo de letra

Os conteúdos gerados por IA poderão estar incorretos.

Figura - Verificação de declarações - variáveis

Uma imagem com texto, captura de ecrã, Tipo de letra

Os conteúdos gerados por IA poderão estar incorretos.

Figura - Verificação de declarações - Funções/Protótipos

**Lista de Estruturas Auxiliares Utilizadas nos Métodos do OtimizadorTAC**

Em cada método de otimização do OtimizadorTAC são usadas estruturas auxiliares essências na obtenção dos resultados pretendidos. Essas estruturas são:

**1. Constant Folding**

**Estruturas Auxiliares:**

* constantes\_resolvidas (Dicionário Dict[str, int|float|bool]):
  + Mapeia **nomes de variáveis/temporários** para seus **valores constantes conhecidos**.
  + Exemplo: Se t1 = 5 + 3 é resolvido para t1 = 8, armazena {"t1": 8}.
* **Função** resolve\_operand():
  + Determina se um operando é constante (literal ou variável com valor conhecido).
  + Retorna (valor, é\_constante).

**Fluxo:**

1. Percorre os quadruplos buscando operações com operandos constantes.
2. Usa constantes\_resolvidas para substituir variáveis por valores conhecidos.
3. Calcula o resultado da operação em tempo de compilação e atualiza o dicionário.

**2. Propagação de Cópias**

**Estruturas Auxiliares:**

* Substituições (Dicionário Dict[str, str]):
  + Mapeia **variáveis destino** para **variáveis origem** em cópias diretas (ex: {"b": "a"} para b = a).
* Atribuições (Dicionário Dict[str, int]):
  + Conta quantas vezes cada variável foi atribuída (ex: {"a": 1, "b": 2}).
* modificadas\_em\_ciclos (Conjunto Set[str]):
  + Armazena variáveis modificadas em loops/desvios (não seguras para propagação).

**Fluxo:**

1. Conta atribuições com atribuicoes.
2. Marca variáveis em loops com modificadas\_em\_ciclos.
3. Substitui usos de variáveis copiadas usando substituições (ex: substitui b por a se b = a).

**3. Eliminação de Subexpressões Comuns (CSE)**

**Estruturas Auxiliares:**

* expressoes\_vistas (Dicionário Dict[Tuple, str]):
  + Chave: Tupla (operação, arg1, versão\_arg1, arg2, versão\_arg2) (ex: ("+", "a", 1, "b", 1)).
  + Valor: Nome do temporário que armazenou o resultado (ex: "t1").
* versao\_vars (Dicionário Dict[str, int]):
  + Rastreia **versões** de variáveis (incrementadas a cada atribuição).
  + Evita falsos positivos (ex: se a muda, a + b não é mais a mesma subexpressão).

**Fluxo:**

1. Para cada operação, gera uma chave única baseada nos operandos e suas versões.
2. Se a chave existe em expressoes\_vistas, reutiliza o resultado armazenado.
3. Caso contrário, registra a nova expressão no dicionário.

**4. Loop-Invariant Code Motion (LICM)**

**Estruturas Auxiliares:**

* label\_idx (Dicionário Dict[str, int]):
  + Mapeia **labels** para **índices** no TAC (ex: {"L1": 5}).
* Defs (Conjunto Set[str]):
  + Variáveis modificadas dentro do loop.
* Invariantes (Lista List[Tuple[int, int, Quadrupla]]):
  + Armazena instruções invariantes e suas posições originais.

**Fluxo:**

1. Identifica loops (saltos para trás) usando label\_idx.
2. Coleta variáveis modificadas no loop (defs).
3. Move instruções cujos operandos não estão em defs para fora do loop.

**5. Eliminação de Código Morto**

**Estruturas Auxiliares:**

* vivas (Conjunto Set[str]):
  + Variáveis cujos valores são lidos posteriormente.
* usados (Conjunto Set[str]):
  + Variáveis usadas como operandos em qualquer instrução.
* variaveis\_utilizador (Conjunto Set[str]):
  + Variáveis explicitamente marcadas como não removíveis (ex: saídas do programa).

**Fluxo:**

1. Identifica variáveis usadas (usados) em uma primeira passagem.
2. Na segunda passagem (de trás para frente), atualiza vivas com variáveis lidas.
3. Remove instruções cujos resultados não estão em vivas.

**6. Eliminação de Código Inatingível**

**Estruturas Auxiliares:**

* lideres (Conjunto Set[int]):
  + Índices de instruções que iniciam blocos básicos (primeira instrução, alvos de saltos, etc.).
* blocos (Lista List[BlocoBasico]):
  + Blocos básicos do CFG, com sucessores e predecessores.
* ids\_blocos\_alcancaveis (Conjunto Set[int]):
  + Blocos alcançáveis a partir do bloco inicial (via BFS no CFG).

**Fluxo:**

1. Constrói o CFG com lideres e blocos.
2. Executa BFS para encontrar blocos alcançáveis (ids\_blocos\_alcancaveis).
3. Remove blocos não alcançáveis e salta para labels inexistentes.

**7. Construção do Grafo de Fluxo de Controle (CFG)**

**Estruturas Auxiliares:**

* mapa\_labels\_para\_id\_bloco (Dicionário Dict[str, int]):
  + Relaciona **labels** a **IDs de blocos** (ex: {"L1": 0}).
* mapa\_idx\_lider\_para\_id\_bloco (Dicionário Dict[int, int]):
  + Mapeia **índices de líderes** no TAC para **IDs de blocos**.

**Fluxo:**

1. Identifica líderes e cria blocos básicos.
2. Conecta blocos via sucessores com base em goto, ifFalse, etc.
3. Usa os mapas para resolver saltos durante otimizações.

**Resumo das Estruturas por Método**

| **Método** | **Estruturas Auxiliares** | **Finalidade** |
| --- | --- | --- |
| Constant Folding | constantes\_resolvidas,  resolve\_operand | Avaliar expressões constantes e substituir por resultados. |
| Propagação de Cópias | substituicoes,  atribuicoes,  modificadas\_em\_ciclos | Eliminar atribuições redundantes (x = y → substituir x por y). |
| CSE | expressoes\_vistas,  versao\_vars | Evitar recálculo de expressões idênticas. |
| LICM | label\_idx,  defs, invariantes | Mover código invariante para fora de loops. |
| Código Morto | vivas,  usados, variaveis\_utilizador | Remover instruções com resultados não utilizados. |
| Código Inatingível | lideres,  blocos, ids\_blocos\_alcancaveis | Eliminar blocos nunca executados. |
| CFG | mapa\_labels\_para\_id\_bloco,  mapa\_idx\_lider\_para\_id\_bloco | Modelar fluxo do programa para análises. |

Estas estruturas garantem que as otimizações são **precisas** e **eficientes**, evitando efeitos colaterais indesejados.

**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/>