

**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.

**1. 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 contextos 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 atual ou um superior). Se um identificador não for encontrado, uma exceção de erro semântico é levantada. Adicionalmente, 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 todas as estruturas da linguagem (expressões, blocos de código, instruções de controlo *if/else*, *while*, *for*, *return*, entre outras), garantindo que as validações semânticas fundamentais são aplicadas em todos os contextos apropriados. As ações semânticas implementadas são:

**Verificação de declarações**, garante que identificadores e variáveis foram declarados antes do uso e impede a sua redeclaração no mesmo contexto. Para funções e protótipos, valida a consistência dos tipos de retorno e da lista de parâmetros no momento da sua definição; (Figura 1 e Figura 2).

**Gestão de contextos**, Salvaguarda que cada bloco de código, função ou estrutura de controlo inicia e termina corretamente um novo contexto através de uma pilha de contextos na tabela de símbolos. Assegurando assim, que declarações feitas num determinado contexto não interferem em outros, garantindo isolamento semântico. (Figura 3)

**Verificação de tipos**, nas instruções de atribuição é verificada a compatibilidade entre o tipo da variável e o valor que está a ser atribuído, no caso das expressões os operandos são analisados conforme o operador em questão (aritméticas, relacionais ou lógico). Finalmente, nas funções é verificado se o tipo de *return* corresponde ao tipo que foi declarado. (Figura 4)

**Tratamento de estruturas de controle**; nas estruturas condicionais (*if, 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 vetores, validamos as dimensões e o número de elementos fornecidos na lista de inicialização, adicionalmente, no acesso a índices, impedimos o uso de [] em variáveis não declaradas como vetores, prevenindo acessos inválidos.

**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, nomeadamente no tipo de retorno e na lista de tipos dos parâmetros, detetando redefinições inválidas ou incompatibilidades entre protótipo e definição.

Em geral, tivemos como objetivo implementar uma análise semântica robusta, onde analisamos as declarações (variáveis, funções, vetores); os tipos (atribuições, expressões, retornos), os contextos (hierarquia de blocos) e o fluxo de controle (ciclos e condicionais). Nos anexos são apresentados os testes efetuados na deteção de erros semânticos assim como uma lista dos erros semânticos detetados pelo compilador.

**2. 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.

Foi implementado um outro Visitor (VisitorTAC.py), responsável por percorrer novamente a árvore de análise sintática, utilizando a informação previamente recolhida e validada para gerar a sequência de instruções de código intermédio. A principal responsabilidade deste visitor é traduzir as construções da linguagem MOC para uma representação mais próxima do nível de 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*) e armazenadas internamente numa lista (*tac\_quadruplos*). Cada quádrupla é representada como um dicionário com as chaves "*op*", "*arg1*", "*arg2*", e "*res*". Adicionalmente, a classe utiliza também contadores para gerar nomes únicos para variáveis temporárias (*novo\_temp*) e rótulos (*nova\_label*), essenciais para representar resultados intermédios de expressões e fluxos de controlo. Existem também, um método auxiliar *adicionar\_quadruplo* facilita a criação destas quádruplas, abstraindo a geração de registos temporários e organização dos operandos. O acesso a informações semânticas, como variáveis ou funções declaradas, é feito através de estruturas mantidas ou herdadas pelo próprio visitor.

A tradução das construções da linguagem MOC para TAC é realizada através da sobrescrita dos métodos *visitNomeDaRegra*. As expressões aritméticas, lógicas e relacionais são decompostas em quádruplas elementares, recorrendo a variáveis temporárias para armazenar os resultados intermédios. As atribuições simples são convertidas em instruções do tipo *assign =* ou no caso de vetores, em *[]=* (com cálculo explícito de offset). Estruturas de controlo como *if-else*, *while* e *for* são traduzidas utilizando rótulos (*label*) e instruções de salto condicional (*ifFalse*) ou incondicionais (*goto*). Chamadas a funções são tratadas como instruções *param* para os argumentos e *call* para a invocação, enquanto as funções de entrada/saída (*read, write, etc*.) são mapeadas diretamente para quádruplas específicas. É utilizada uma função utilitária, *gerar\_texto\_tac*, para converter e formatar a lista de quádruplas numa representação textual legível do nosso código intermédio. No final do processo de visita, a lista *tac\_quadruplos* contém a representação TAC completa do programa MOC, pronta para a fase de otimização.

**3. Otimização do Código Intermédio (OtimizadorTAC.py)**

Com o código intermédio (TAC) gerado, foi desenvolvido um módulo de otimização (*OtimizadorTAC.py)* com o objetivo de aplicar melhorarias sobre a sequência de quádruplas produzidas, antes da geração final, sem alterar a semântica do programa. O otimizador recebe a lista de quádruplas gerada na fase anterior e aplica um conjunto de transformações que visam tornar o código potencialmente mais rápido e/ou mais curto, aumentando a sua eficiência, preparando-o para a etapa de geração de código final.

Ao ser instanciada, a classe *OtimizadorTAC* cria uma cópia profunda (*copy.deepcopy*) da lista de quádruplas recebida, garantindo assim que as otimizações não modificam o código 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). Esta classe implementa múltiplos métodos, cada um correspondendo a uma técnica de otimização distinta, que descrevemos abaixo com mais detalhe:

* **Eliminação de Código Morto** (*eliminar\_codigo\_morto*): Esta técnica utiliza uma análise de vivacidade (liveness analysis) percorrendo os quádruplos de trás para a frente com objetivo de identificar as variáveis cujo valor é efetivamente utilizado mais tarde ("*variáveis vivas*"). Instruções cujo resultado (variável de destino *res*) não pertence a esse 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) são consideradas “código morto” e consequentemente eliminadas. Instruções sem resultado (labels, gotos) ou com efeitos colaterais são preservadas.
* **Propagação de Cópias** (*propagacao\_copias*): São analisadas 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 otimizador substitui as utilizações subsequentes de *t2* diretamente por *t1*, eliminando a cópia redundante.
* **Dobramento de Constantes** (*constant\_folding*): Consiste em procurar por operações aritméticas cujos operandos são ambos constantes numéricas conhecidas em tempo de compilação (ex: *t1 = 3 + 5*). Essas operações são avaliadas e substituídas por uma atribuição direta do resultado calculado (ex: *t1 = 8*), sendo um processo iterativo visto que uma substituição poder propagar novas constantes para as instruções seguintes e necessitar de dobramentos adicionais.
* **Eliminação de Subexpressões Comuns** (*eliminar\_subexpressoes\_comuns\_CSE*): Mantém um registo das expressões já calculadas e das versões atuais das variáveis. Caso encontre uma expressão (*ex: a + b*) que já tenha sido calculada anteriormente com os mesmos operandos (com os mesmos valores e versões), em vez de gerar o código para recalcular, reutiliza o resultado anterior (guardado num temporário), substituindo a computação redundante por uma cópia. Incluímos também uma normalização para operações comutativas, ou seja, *a+b* é tratado como equivalente a *b+a.*
* **Eliminação de Código Inatingível** (*eliminar\_codigo\_inatingivel*): Remove sequências de instruções que nunca podem ser executadas, normalmente por seguirem imediatamente uma instrução de salto incondicional (*goto*) ou um *return*, e não serem alvo de nenhuma *label*. De forma a alcançar esta otimização é criado um CFG (grafo de fluxo de controlo) após identificarmos as instruções “líderes” no TAC, são criados os blocos básicos associados e por fim efetuada a análise de alcançabilidade dos blocos de código. Isto permite detetar blocos inatingíveis e assim remover os mesmos do código intermédio.
* **Movimento de Código Invariante de Ciclos** (*mover\_invariantes*) (**Loop-Invariant Code Motion – LICM)**: Identificamos ciclo (através de saltos para trás - back-edges). Dentro de cada ciclo, procuramos por instruções cujos operandos são definidos fora do ciclo (ou que são valores constantes). Estas instruções "invariantes" são movidas para imediatamente antes do ponto de entrada do ciclo, evitando que sejam recalculadas desnecessariamente a cada iteração.

De forma a orquestrar a aplicação destas técnicas, é implementada a função *otimizar\_completo*. Esta função instância o *OtimizadorTAC* e aplica as otimizações numa ordem específica, incluindo a execução repetida da eliminação de código morto até um ponto fixo, ou seja, até que nenhuma alteração adicional seja possível, dado que outras otimizações podem gerar novo código redundante. O resultado final desta função é a lista de quádruplas TAC otimizada, pronta para a geração de código final.

Concluindo, o nosso compilador da linguagem MOC evoluiu, desde compreender as palavras e estruturas do código (análise lexical e sintática), passando pela verificação de coerência lógica (análise semântica), até á geração de uma versão intermédia mais simples e próxima da máquina: o código intermédio (TAC). Ao mesmo, foram aplicadas várias técnicas de otimização, de forma a garantir o aproveitamento de constantes, eliminação de código inatingível e redução de cálculos redundantes, permitindo um código intermédio mais limpo e eficiente, e que pronto para as seguintes fases da compilação.

Nos anexos apresentamos inúmeros testes efetuados ao longo do desenvolvimento de cada uma das etapas.

**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.

**Exemplo, testes efetuados e análise**

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

Os conteúdos gerados por IA poderão estar incorretos.

Figura 1 - Verificação de declarações - variáveis

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

Os conteúdos gerados por IA poderão estar incorretos.

Figura 2 - Verificação de declarações - Funções/Protótipos

A screenshot of a computer screen

AI-generated content may be incorrect.

A screenshot of a computer

AI-generated content may be incorrect.

Figura - Criação de contextos no início de função, início do main e início de bloco

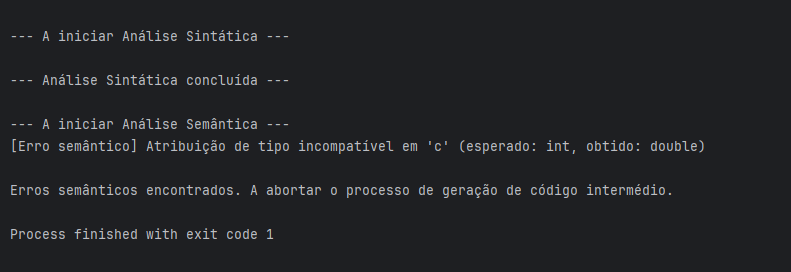
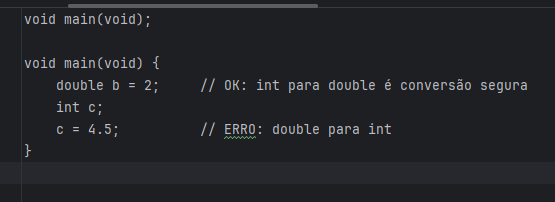


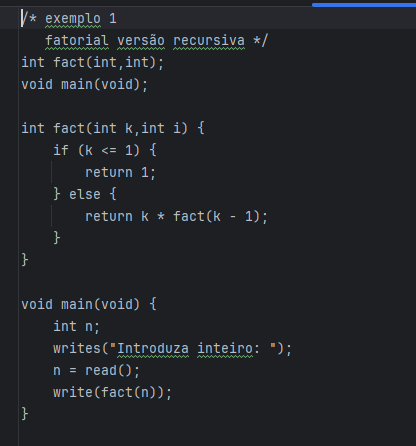
Figura 4 - Input e Output de Verificação de compatibilidade de tipos

A screenshot of a computer program

AI-generated content may be incorrect.

Figura - Verificação da deteção dos erros semânticos

**Input:**



**Output:**

python.exe main.py .\test\_examples\exemplo1.moc

--- A iniciar Análise Sintática ---

--- Análise Sintática concluída ---

--- A iniciar Análise Semântica ---

--- Análise Semântica concluída ---

--- A iniciar Geração de Código Intermédio ---

--- Geração de Código Intermédio concluída ---

==== CÓDIGO TAC GERADO ====

fact:

k = param1

i = param2

t1 = k <= 1

ifFalse t1 goto L2

L1:

return 1

goto L3

L2:

t2 = k - 1

param t2

t3 = call fact

t4 = k \* t3

return t4

L3:

end\_fact:

main:

writes "Introduza inteiro: "

t5 = call read

n = t5

param n

t6 = call fact

write t6

halt

end\_main:

==== CÓDIGO TAC OTIMIZADO ====

fact:

t1 = param1 <= 1

ifFalse t1 goto L2

L2:

t2 = param1 - 1

param t2

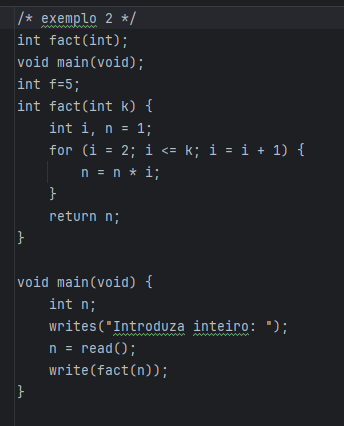
t3 = call fact

t4 = param1 \* t3

return t4

Process finished with exit code 0

**Input:**



**Output:**

python.exe main.py .\test\_examples\exemplo2.moc

--- A iniciar Análise Sintática ---

--- Análise Sintática concluída ---

--- A iniciar Análise Semântica ---

--- Análise Semântica concluída ---

--- A iniciar Geração de Código Intermédio ---

--- Geração de Código Intermédio concluída ---

==== CÓDIGO TAC GERADO ====

f = 5

fact:

k = param1

n = 1

i = 2

L1:

t1 = i <= k

ifFalse t1 goto L3

L2:

t2 = n \* i

n = t2

t3 = i + 1

i = t3

goto L1

L3:

return n

end\_fact:

main:

writes "Introduza inteiro: "

t4 = call read

n = t4

param n

t5 = call fact

write t5

halt

end\_main:

==== CÓDIGO TAC OTIMIZADO ====

fact:

n = 1

i = 2

n = 2

i = 3

L1:

t1 = i <= param1

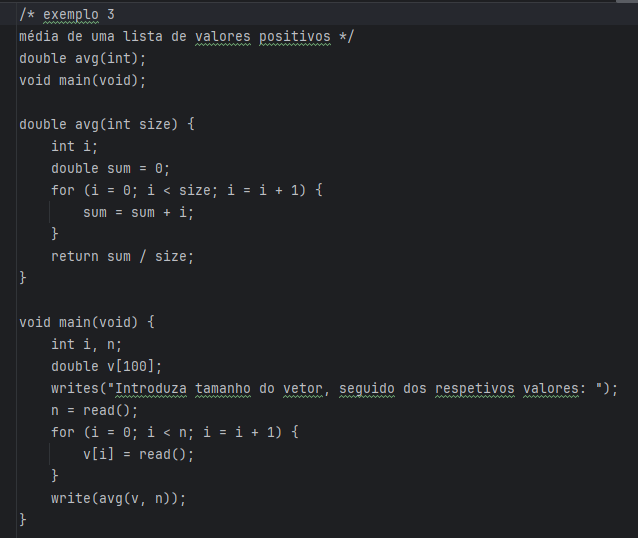
ifFalse t1 goto L3

L3:

return n

Process finished with exit code 0

**Input:**



**Output:**

python.exe main.py .\test\_examples\exemplo3.moc

--- A iniciar Análise Sintática ---

--- Análise Sintática concluída ---

--- A iniciar Análise Semântica ---

--- Análise Semântica concluída ---

--- A iniciar Geração de Código Intermédio ---

--- Geração de Código Intermédio concluída ---

==== CÓDIGO TAC GERADO ====

avg:

size = param1

sum = 0

i = 0

L1:

t1 = i < size

ifFalse t1 goto L3

L2:

t2 = sum + i

sum = t2

t3 = i + 1

i = t3

goto L1

L3:

t4 = sum / size

return t4

end\_avg:

main:

alloc v, 100

writes "Introduza tamanho do vetor, seguido dos respetivos valores: "

t5 = call read

n = t5

i = 0

L4:

t6 = i < n

ifFalse t6 goto L6

L5:

t7 = i \* 4

t8 = call read

v[t7] = t8

t9 = i + 1

i = t9

goto L4

L6:

param v

param n

t10 = call avg

write t10

halt

end\_main:

==== CÓDIGO TAC OTIMIZADO ====

avg:

sum = 0

i = 0

i = 1

L1:

t1 = i < param1

ifFalse t1 goto L3

L3:

t4 = sum / param1

return t4

Process finished with exit code 0

**Input:**

A screen shot of a computer program

AI-generated content may be incorrect.

**Output:**

python.exe main.py .\test\_examples\exemplo4.moc

--- A iniciar Análise Sintática ---

--- Análise Sintática concluída ---

--- A iniciar Análise Semântica ---

--- Análise Semântica concluída ---

--- A iniciar Geração de Código Intermédio ---

--- Geração de Código Intermédio concluída ---

==== CÓDIGO TAC GERADO ====

main:

x = 10

write x

halt

end\_main:

==== CÓDIGO TAC OTIMIZADO ====

main:

write 10

halt

end\_main:

Process finished with exit code 0

**Input:**

A screen shot of a computer program

AI-generated content may be incorrect.

**Output:**

python.exe main.py .\test\_examples\exemplo5.moc

--- A iniciar Análise Sintática ---

--- Análise Sintática concluída ---

--- A iniciar Análise Semântica ---

--- Análise Semântica concluída ---

--- A iniciar Geração de Código Intermédio ---

--- Geração de Código Intermédio concluída ---

==== CÓDIGO TAC GERADO ====

main:

x = 10

y = x

t1 = x == 10

ifFalse t1 goto L2

L1:

writes "10"

goto L3

L2:

t2 = x + 25

x = t2

L3:

halt

end\_main:

==== CÓDIGO TAC OTIMIZADO ====

main:

x = 10

L1:

writes "10"

goto L3

L3:

halt

end\_main:

Process finished with exit code 0

**Input:**

A screenshot of a computer program

AI-generated content may be incorrect.

**Output:**

python.exe main.py .\test\_examples\exemplo6.moc

--- A iniciar Análise Sintática ---

--- Análise Sintática concluída ---

--- A iniciar Análise Semântica ---

--- Análise Semântica concluída ---

--- A iniciar Geração de Código Intermédio ---

--- Geração de Código Intermédio concluída ---

==== CÓDIGO TAC GERADO ====

main:

a = 10

t1 = b + 10

c = t1

halt

end\_main:

==== CÓDIGO TAC OTIMIZADO ====

main:

t1 = b + 10

halt

end\_main:

Process finished with exit code 0

**Input:**

A screenshot of a computer program

AI-generated content may be incorrect.

**Output:**

python.exe main.py .\test\_examples\exemplo7.moc

--- A iniciar Análise Sintática ---

--- Análise Sintática concluída ---

--- A iniciar Análise Semântica ---

--- Análise Semântica concluída ---

--- A iniciar Geração de Código Intermédio ---

--- Geração de Código Intermédio concluída ---

==== CÓDIGO TAC GERADO ====

main:

x = 10

y = 20

t1 = call read

n = t1

i = 0

L1:

t2 = i < n

ifFalse t2 goto L3

L2:

t3 = x + y

z = t3

write z

t4 = i + 1

i = t4

goto L1

L3:

halt

end\_main:

==== CÓDIGO TAC OTIMIZADO ====

main:

t1 = call read

i = 0

i = 1

L1:

t2 = i < t1

ifFalse t2 goto L3

L3:

halt

end\_main:

Process finished with exit code 0

**Input:**

A screenshot of a computer

AI-generated content may be incorrect.

**Output:**

A screen shot of a computer

AI-generated content may be incorrect.

**Input:**

A screen shot of a computer

AI-generated content may be incorrect.

**Output:**

A computer screen shot of a program

AI-generated content may be incorrect.

**Input:**

A computer screen with white text

AI-generated content may be incorrect.

**Output:**

python.exe main.py .\test\_examples\teste\_ciclo.moc

--- A iniciar Análise Sintática ---

--- Análise Sintática concluída ---

--- A iniciar Análise Semântica ---

--- Análise Semântica concluída ---

--- A iniciar Geração de Código Intermédio ---

--- Geração de Código Intermédio concluída ---

==== CÓDIGO TAC GERADO ====

main:

soma = 0

i = 0

L1:

t1 = i < 10

ifFalse t1 goto L3

L2:

t2 = i == 5

ifFalse t2 goto L5

L4:

write i

L5:

t3 = soma + i

soma = t3

write soma

t4 = i + 1

goto L1

L3:

write soma

return 0

halt

end\_main:

==== CÓDIGO TAC OTIMIZADO ====

main:

soma = 0

t2 = False

t3 = 0

L1:

L2:

ifFalse t2 goto L5

L5:

soma = t3

write soma

goto L1

Process finished with exit code 0

**Input:**

A screen shot of a computer program

AI-generated content may be incorrect.

**Output:**

A computer screen with white text

AI-generated content may be incorrect.

**Input:**

A screen shot of a computer program

AI-generated content may be incorrect.

**Output:**

python.exe main.py .c\test\_examples\teste\_geral.moc

--- A iniciar Análise Sintática ---

--- Análise Sintática concluída ---

--- A iniciar Análise Semântica ---

--- Análise Semântica concluída ---

--- A iniciar Geração de Código Intermédio ---

--- Geração de Código Intermédio concluída ---

==== CÓDIGO TAC GERADO ====

main:

t1 = 2 + 3

x = t1

y = x

z = y

t2 = 10 - 4

r = t2

t = r

a = 7

b = 3

t3 = a + b

soma = t3

t4 = (double) soma

d = t4

alloc v, 3

t5 = 0 \* 4

v[t5] = 1

t6 = 1 \* 4

v[t6] = 2

t7 = 2 \* 4

v[t7] = 3

i = 0

total = 0

L1:

t8 = i < 3

ifFalse t8 goto L3

L2:

t9 = i \* 4

t10 = v[t9]

t11 = total + t10

total = t11

t12 = i + 1

i = t12

goto L1

L3:

t13 = total > 5

ifFalse t13 goto L5

L4:

write total

goto L6

L5:

write 0

L6:

copia = total

resultado = copia

write resultado

t14 = call read

t15 = t14

t16 = call readc

t17 = t16

t18 = call reads

t19 = t18

writes "fim"

halt

end\_main:

==== CÓDIGO TAC OTIMIZADO ====

main:

t4 = (double) 10

alloc v, 3

v[0] = 1

v[4] = 2

v[8] = 3

total = 0

t9 = 0

L1:

L2:

t10 = v[t9]

t11 = total + t10

total = t11

goto L1

Process finished with exit code 0

**Input:**

A computer screen shot of a program

AI-generated content may be incorrect.

**Output:**

A screen shot of a computer

AI-generated content may be incorrect.

**Input:**

A screenshot of a computer program

AI-generated content may be incorrect.

**Output:**

python.exe main.py .\test\_examples\Testes\_optimizador01.moc

--- A iniciar Análise Sintática ---

--- Análise Sintática concluída ---

--- A iniciar Análise Semântica ---

--- Análise Semântica concluída ---

--- A iniciar Geração de Código Intermédio ---

--- Geração de Código Intermédio concluída ---

==== CÓDIGO TAC GERADO ====

main:

x = 5

a = 2.5

halt

end\_main:

==== CÓDIGO TAC OTIMIZADO ====

main:

halt

end\_main:

Process finished with exit code 0

**Input:**

A computer screen shot of a black screen

AI-generated content may be incorrect.

**Output:**

python.exe main.py .\test\_examples\Testes\_optimizador02.moc

--- A iniciar Análise Sintática ---

--- Análise Sintática concluída ---

--- A iniciar Análise Semântica ---

--- Análise Semântica concluída ---

--- A iniciar Geração de Código Intermédio ---

--- Geração de Código Intermédio concluída ---

==== CÓDIGO TAC GERADO ====

main:

t1 = 0 \* 4

x[t1] = 1

t2 = 1 \* 4

x[t2] = 2

t3 = 2 \* 4

x[t3] = 3

t4 = 0 \* 4

t5 = x[t4]

t6 = 1 \* 4

t7 = x[t6]

t8 = t5 + t7

y = t8

write y

halt

end\_main:

==== CÓDIGO TAC OTIMIZADO ====

main:

x[0] = 1

x[4] = 2

x[8] = 3

t4 = 0

t5 = x[t4]

t7 = x[4]

t8 = t5 + t7

write t8

halt

end\_main:

Process finished with exit code 0

**Input:**

A screen shot of a computer

AI-generated content may be incorrect.

**Output:**

python.exe main.py .\test\_examples\Testes\_optimizador03.moc

--- A iniciar Análise Sintática ---

--- Análise Sintática concluída ---

--- A iniciar Análise Semântica ---

--- Análise Semântica concluída ---

--- A iniciar Geração de Código Intermédio ---

--- Geração de Código Intermédio concluída ---

==== CÓDIGO TAC GERADO ====

main:

a = 5

b = 10

t1 = a + b

t2 = t1 \* 2

c = t2

t3 = c / 5.0

d = t3

write d

halt

end\_main:

==== CÓDIGO TAC OTIMIZADO ====

main:

write 6.0

halt

end\_main:

Process finished with exit code 0

**Input:**

A screen shot of a computer

AI-generated content may be incorrect.

**Output:**

python.exe main.py .\test\_examples\Testes\_optimizador04.moc

--- A iniciar Análise Sintática ---

--- Análise Sintática concluída ---

--- A iniciar Análise Semântica ---

--- Análise Semântica concluída ---

--- A iniciar Geração de Código Intermédio ---

--- Geração de Código Intermédio concluída ---

==== CÓDIGO TAC GERADO ====

main:

a = 50

b = 10

t1 = a + b

c = t1

t2 = 3 \* b

d = t2

t3 = c - d

e = t3

write e

halt

end\_main:

==== CÓDIGO TAC OTIMIZADO ====

main:

write 30

halt

end\_main:

Process finished with exit code 0

**Input:**

A screen shot of a computer program

AI-generated content may be incorrect.

**Output:**

python.exe main.py .\test\_examples\Testes\_optimizador05.moc

--- A iniciar Análise Sintática ---

--- Análise Sintática concluída ---

--- A iniciar Análise Semântica ---

--- Análise Semântica concluída ---

--- A iniciar Geração de Código Intermédio ---

--- Geração de Código Intermédio concluída ---

==== CÓDIGO TAC GERADO ====

main:

a = 5

b = 10

t1 = b \* 2

t2 = a + t1

c = t2

t3 = b + 1

d = t3

write d

halt

end\_main:

==== CÓDIGO TAC OTIMIZADO ====

main:

write 11

halt

end\_main:

Process finished with exit code 0

**Input:**

A screen shot of a computer program

AI-generated content may be incorrect.

**Output:**

python.exe main.py .\test\_examples\Testes\_optimizador06.moc

--- A iniciar Análise Sintática ---

--- Análise Sintática concluída ---

--- A iniciar Análise Semântica ---

--- Análise Semântica concluída ---

--- A iniciar Geração de Código Intermédio ---

--- Geração de Código Intermédio concluída ---

==== CÓDIGO TAC GERADO ====

main:

a = 5

b = 10

t1 = b \* 2.0

t2 = a + t1

c = t2

t3 = c / 5.0

d = t3

write d

t4 = call read

f = t4

t5 = 2 \* b

g = t5

t6 = 0 \* 4

x[t6] = 1

t7 = 1 \* 4

x[t7] = 2

t8 = 2 \* 4

x[t8] = 3

t9 = 0 \* 4

t10 = x[t9]

t11 = 1 \* 4

t12 = x[t11]

t13 = t10 + t12

y = t13

write y

halt

end\_main:

==== CÓDIGO TAC OTIMIZADO ====

main:

write 5.0

t4 = call read

x[0] = 1

x[4] = 2

x[8] = 3

t9 = 0

t10 = x[t9]

t12 = x[4]

t13 = t10 + t12

write t13

halt

end\_main:

Process finished with exit code 0

**Input:**

A computer screen with white text

AI-generated content may be incorrect.

**Output:**

python.exe main.py .\test\_examples\Testes\_optimizador07.moc

--- A iniciar Análise Sintática ---

--- Análise Sintática concluída ---

--- A iniciar Análise Semântica ---

--- Análise Semântica concluída ---

--- A iniciar Geração de Código Intermédio ---

--- Geração de Código Intermédio concluída ---

==== CÓDIGO TAC GERADO ====

main:

x = 15

y = x

t1 = x == 10

ifFalse t1 goto L2

L1:

writes "10"

goto L3

L2:

t2 = x + 25

x = t2

L3:

halt

end\_main:

==== CÓDIGO TAC OTIMIZADO ====

main:

x = 15

t1 = False

ifFalse t1 goto L2

L2:

x = 40

L3:

halt

end\_main:

Process finished with exit code 0

**Input:**

A computer screen with white text and numbers

AI-generated content may be incorrect.

**Output:**

python.exe main.py .\test\_examples\Testes\_optimizador08.moc

--- A iniciar Análise Sintática ---

--- Análise Sintática concluída ---

--- A iniciar Análise Semântica ---

--- Análise Semântica concluída ---

--- A iniciar Geração de Código Intermédio ---

--- Geração de Código Intermédio concluída ---

==== CÓDIGO TAC GERADO ====

main:

x = 10

y = x

t1 = x == 10

ifFalse t1 goto L2

L1:

writes "10"

goto L3

L2:

t2 = x + 25

x = t2

L3:

halt

end\_main:

==== CÓDIGO TAC OTIMIZADO ====

main:

x = 10

L1:

writes "10"

goto L3

L3:

halt

end\_main:

Process finished with exit code 0

**Input:**

A screen shot of a computer program

AI-generated content may be incorrect.

**Output:**

python.exe main.py .\test\_examples\Testes\_optimizador09.moc

--- A iniciar Análise Sintática ---

--- Análise Sintática concluída ---

--- A iniciar Análise Semântica ---

--- Análise Semântica concluída ---

--- A iniciar Geração de Código Intermédio ---

--- Geração de Código Intermédio concluída ---

==== CÓDIGO TAC GERADO ====

obter\_valor:

valor = 42

return valor

t1 = valor + 10

valor = t1

write valor

end\_obter\_valor:

main:

t2 = call obter\_valor

resultado = t2

write resultado

return 0

halt

end\_main:

==== CÓDIGO TAC OTIMIZADO ====

obter\_valor:

valor = 42

return valor

Process finished with exit code 0

**Input:**

A computer screen shot of a program

AI-generated content may be incorrect.

**Output:**

python.exe main.py .\test\_examples\Testes\_optimizador10.moc

--- A iniciar Análise Sintática ---

--- Análise Sintática concluída ---

--- A iniciar Análise Semântica ---

--- Análise Semântica concluída ---

--- A iniciar Geração de Código Intermédio ---

--- Geração de Código Intermédio concluída ---

==== CÓDIGO TAC GERADO ====

main:

soma = 0

i = 0

L1:

t1 = i < 10

ifFalse t1 goto L3

L2:

t2 = i == 5

ifFalse t2 goto L5

L4:

write i

L5:

t3 = soma + i

soma = t3

write soma

t4 = i + 1

goto L1

L3:

write soma

return 0

halt

end\_main:

==== CÓDIGO TAC OTIMIZADO ====

main:

soma = 0

t2 = False

t3 = 0

L1:

L2:

ifFalse t2 goto L5

L5:

soma = t3

write soma

goto L1

Process finished with exit code 0

**Input:**

A computer screen shot of a computer code

AI-generated content may be incorrect.

**Output:**

python.exe main.py .\test\_examples\Testes\_optimizador11.moc

--- A iniciar Análise Sintática ---

--- Análise Sintática concluída ---

--- A iniciar Análise Semântica ---

--- Análise Semântica concluída ---

--- A iniciar Geração de Código Intermédio ---

--- Geração de Código Intermédio concluída ---

==== CÓDIGO TAC GERADO ====

main:

a = 10

t1 = b + 10

c = t1

halt

end\_main:

==== CÓDIGO TAC OTIMIZADO ====

main:

t1 = b + 10

halt

end\_main:

Process finished with exit code 0

**Input:**

A screen shot of a computer

AI-generated content may be incorrect.

**Output:**

python.exe main.py .\test\_examples\Testes\_optimizador12.moc

--- A iniciar Análise Sintática ---

--- Análise Sintática concluída ---

--- A iniciar Análise Semântica ---

--- Análise Semântica concluída ---

--- A iniciar Geração de Código Intermédio ---

--- Geração de Código Intermédio concluída ---

==== CÓDIGO TAC GERADO ====

main:

a = 100

write a

b = a

a = 10

t1 = b + 20

c = t1

write c

halt

end\_main:

==== CÓDIGO TAC OTIMIZADO ====

main:

a = 100

write a

a = 10

write 120

halt

end\_main:

Process finished with exit code 0

**Lista de erros semânticos detetados**

Para se entender a natureza dos diferentes erros semânticos que podem ocorrer durante a análise temos os seguintes erros detetados:

**1. Erros de declaração, redeclaração e definição:**

Erros relacionados à forma como variáveis, funções, parâmetros e protótipos são declarados ou definidos, incluindo conflitos de nomes e redefinições.

[Erro semântico] Conflito na declaração de '{nome\_funcao}' (já declarado como {simbolo})

[Erro semântico] Erro ao declarar função '{nome\_funcao}'

[Erro semântico] Erro ao declarar variável '{nome\_var}'

[Erro semântico] Erro inesperado ao declarar função principal '{nome\_funcao}'

[Erro semântico] Erro inesperado ao declarar protótipo '{nome\_funcao}'

[Erro semântico] Parâmetro '{param.nome}' já foi declarado.

[Erro semântico] Parâmetro '{param['nome']}' redeclarado

[Erro semântico] Protótipo '{nome\_funcao}' redeclarado identicamente

[Erro semântico] Redefinição da função principal '{nome\_funcao}' (já declarada na linha {nrlinha})

[Erro semântico] Redefinição inválida de '{nome\_funcao}' (já declarado como {simbolo})

[Erro semântico] Variável '{nome\_var}' já declarada neste contexto

**2. Erros de tipo:**

Erros que ocorrem quando há incompatibilidade entre os tipos de dados esperados e os tipos de dados fornecidos (em atribuições, inicializações, parâmetros de função, retornos de função).

[Erro semântico] Atribuição incompatível em '{nome\_variavel}' (esperado: {simbolo.tipo}, obtido: {tipoexp})

[Erro semântico] Inicialização com tipo incompatível para '{nome}' (esperado: {tipo\_var}, obtido: {tipo\_expr})

[Erro semântico] Parâmetros incompatíveis para '{nome\_funcao}'

[Erro semântico] Tipo de retorno incompatível para '{funcao}' (esperado: {tiporetorno}, obtido: {tiporetorno})

**3. Uso de Identificadores não declarados:**

Erros que acontecem ao tentar usar uma variável, função ou vetor que não foi previamente declarado no escopo atual.

[Erro semântico] Função '{nome}' não declarada

[Erro semântico] Identificador '{nome}' não declarado (linha {linha})

[Erro semântico] Variável '{nome\_variavel}' não declarada"

[Erro semântico] Variável '{nome}' não declarada

[Erro semântico] Vetor '{nome}' usado antes de ser declarado.

**4. Uso indevido de identificadores:**

Erros relacionados ao uso de um identificador de forma incorreta para o seu tipo (por exemplo, tentar aceder a um índice de algo que não é um vetor).

[Erro semântico] Acesso a índice em não-vetor '{nome}'

[Erro semântico] Índice aplicado a não-vetor '{nome\_variavel}'

**5. Erros específicos de vetor/array:**

Erros que se aplicam exclusivamente à definição ou manipulação de vetores (arrays).

[Erro semântico] Tamanho de array deve ser positivo

[Erro semântico] Tamanho de array inválido

**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/>
* ANTLR Doc: <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/>