A white rectangular sign with red text

AI-generated content may be incorrect.

**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:** 25 de Junho de 2025

**TRABALHO / RESOLUÇÃO:**

Este relatório documenta a fase final de desenvolvimento do compilador para a linguagem MOC. O objetivo principal do projeto consistiu na criação de um compilador capaz de traduzir código fonte MOC para uma linguagem de baixo nível, tendo sido selecionado o **Assembly P3** como a arquitetura de destino.

Nesta fase final, o foco esteve na implementação do gerador de código, a última etapa do processo de compilação que ocorre após as fases de análise (léxica, sintática, semântica) e optimização da linguagem intermédia (Código de Três Endereços - TAC). Adicionalmente, foram corrigidos dois erros críticos detetados durante a implementação, um na análise semântica e outro na optimização de código, que foram cruciais para garantir a robustez e correção do compilador.

**Correções de Erros Implementadas**

No decurso da implementação, foram identificados e corrigidos dois erros significativos que afetavam a funcionalidade do compilador.

**1. Correção na Análise Semântica de acesso a Arrays**

* **Problema:** Foi detetado um erro na validação de tipos durante a análise semântica. Ao aceder a um elemento de um array do tipo double (ex: v[1]), o compilador indicava, incorretamente, que o índice do array também deveria ser do tipo double. Um índice de array deve ser sempre um valor inteiro.
* **Solução:** A regra de verificação de tipos para operações de acesso a arrays foi revista e corrigida. A nova lógica garante que o tipo de dado da expressão utilizada como índice seja sempre validado como inteiro, independentemente do tipo de dados armazenado no array. Esta correção assegura que o compilador aplica corretamente as restrições da linguagem.

**2. Correção na Otimização de Código Morto**

* **Problema:** O algoritmo de otimização, especificamente na etapa de eliminação de código morto (*dead code elimination*), apresentava uma falha na identificação do ponto de entrada do programa. O método partia do princípio de que a primeira função declarada no código era o ponto de entrada. Consequentemente, em programas onde a função main não era a primeira, esta e todas as funções por ela chamadas eram incorretamente eliminadas como código morto.
* **Solução:** Foi desenvolvido um método específico para identificar o bloco de código correspondente à função main antes de iniciar a análise de código morto. O algoritmo de optimização foi modificado para usar este bloco como a raiz da sua análise, garantindo que main e todo o código alcançável a partir dela sejam preservados.

**Implementação do Gerador de Código P3 Assembly**

A etapa final do projeto foi a criação da classe GeradorP3Assembly, responsável por traduzir a representação intermédia otimizada (Quádruplos TAC) para código Assembly P3 executável.A classe GeradorP3Assembly adopta uma abordagem de duas fases para garantir que todas as variáveis e estruturas de dados são devidamente declaradas antes da geração do código executável.

**Pré-Análise (\_pre\_scan\_quadruplos):** Antes de iniciar a tradução, o gerador realiza uma passagem completa pela lista de quádruplos. Nesta fase, identifica todos os identificadores que necessitam de alocação de memória:

* + Variáveis e temporários.
  + Arrays (através da instrução alloc), declarando-os com o pseudocódigo TAB.
  + Literais de string (usados na instrução writes), que são adicionados à secção de dados com o pseudocódigo STR. Esta análise prévia permite construir uma secção de dados completa e organizada.

**Tradução de Instruções (translate\_tac\_instruction):** Após a pré-análise, o gerador percorre novamente a lista de quádruplos e traduz cada instrução TAC para uma ou mais instruções Assembly P3.

**Mapeamento de Instruções TAC para Assembly P3**

A seguir, são detalhados alguns exemplos da lógica de tradução implementada:

* **Operações Aritméticas (+, -, \*, /, %):** Uma operação TAC como res = arg1 + arg2 é decomposta num padrão que utiliza registos. Os operandos são primeiro movidos para registos (e.g., R1, R2), a operação P3 correspondente (ADD, SUB, MUL, DIV) é executada, e o resultado é finalmente movido da localização do registo para a variável de destino. *Exemplo de tradução para res = a + b:*

MOV R1, M[a]

MOV R2, M[b]

ADD R1, R2

MOV M[res], R1

* **Operações Relacionais e Saltos (==, <, IFGOTO):** As comparações são traduzidas usando a instrução CMP do P3, que modifica os *flags* de estado. De seguida, uma instrução de salto condicional (e.g., JMP.Z para igualdade, JMP.N para menor que) é usada para desviar o fluxo de controlo. *Exemplo de tradução para if a == b goto L1:*

MOV R1, M[a]

MOV R2, M[b]

CMP R1, R2

JMP.Z L1

* **Acesso a Arrays ([] e []=):** A tradução de acesso a arrays considera a diferença entre o endereçamento em bytes (comum no TAC, onde cada elemento ocupa 4 bytes) e o endereçamento em palavras do P3 (2 bytes por palavra). O *offset* em bytes é convertido para um *offset* em palavras através de uma operação de deslocamento à direita (SHR R1, #1). O endereço final é calculado somando o endereço base do array com o *offset* em palavras, e o acesso à memória é feito através de endereçamento indireto por registo (M[R2]).
* **Chamadas de Funções (PARAM, CALL, RETURN):** O mecanismo de chamada de função foi implementado utilizando a pilha (stack):
  + PARAM arg1: O argumento é movido para um registo e depois colocado na pilha com a instrução PUSH.
  + CALL func: A instrução CALL do P3 é utilizada, que guarda o endereço de retorno na pilha e salta para a etiqueta da função.
  + RETURN val: Por convenção, o valor de retorno é colocado no registo R1 antes da instrução RET, que restaura o fluxo do programa a partir do endereço guardado na pilha.
* **Entrada e Saída (writes, writec):** As operações de escrita foram implementadas como sub-rotinas P3 reutilizáveis (WRITES, WRITEC). Quando uma instrução writes é encontrada, o gerador emite uma chamada (CALL) para a rotina WRITES, passando o endereço da string literal através da pilha. Esta abordagem modulariza o código e evita a duplicação de lógica complexa de I/O.

**Estrutura do Ficheiro Assembly de Saída**

O método final, generate\_from\_tac\_list, monta a string completa do código Assembly, organizando-a numa estrutura clara e funcional:

1. **Secção de Dados:** Iniciada no endereço 8000h (ORIG 8000h), contém todas as declarações de variáveis (WORD), arrays (TAB) e literais de string (STR). O endereço do topo da pilha (SP\_ADDRESS) é também definido aqui.
2. **Secção de Código:** Iniciada no endereço 0000h (ORIG 0000h), começa com um salto (JMP \_start) para o ponto de entrada principal.
3. **Sub-rotinas:** As funções auxiliares, como WRITES e WRITEC, são inseridas após o salto inicial.
4. **Programa Principal:** O código começa na etiqueta \_start, onde o Stack Pointer (SP) é inicializado. Segue-se o código traduzido da função main.
5. **Fim do Programa:** A execução termina com um loop infinito (Fim: BR Fim), uma prática comum para deter a execução no simulador P3, que não possui uma instrução HALT nativa.

**Conclusão**

A fase final do projeto do compilador MOC foi concluída com sucesso. A implementação do gerador de código GeradorP3Assembly demonstrou ser capaz de traduzir eficientemente a linguagem intermédia otimizada para um código Assembly P3 funcional e bem estruturado.

As correções nos módulos de análise semântica e de otimização de código foram fundamentais para aumentar a fiabilidade e o âmbito de aplicação do compilador. O projeto, no seu todo, atingiu os seus objetivos, resultando numa ferramenta de compilação robusta que cobre todas as etapas essenciais, desde a análise do código fonte até à geração de código de máquina executável.

**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 efetuados**

**Input exemplo 1:**

**void main(void);  
void main(void) {  
 writes("le valor maximo: ");  
 int x = read();  
 write(x);  
 writes("le salto: ");  
 int y = read();  
 write(y);  
 int i;  
 for (i = 1; i <= x; i = i + y) {  
 write(i);  
 }  
 writes ("fim");  
}**

**Ouput simulador P3 (**[**https://p3js.goncalomb.com/**](https://p3js.goncalomb.com/)**) exemplo 1:**

**Uma imagem com captura de ecrã, texto

Os conteúdos gerados por IA podem estar incorretos.**

**Input exemplo 2:**

**void main(void);  
void main(void) {  
 writes("le inteiro 1: ");  
 int x = read();  
 write(x);  
 writes("le inteiro 2: ");  
 int y= read();  
 write(y);  
 if (x>10 && y>10){  
 writes("ambos superiores a 10");  
 } else {  
 writes("um numero ou ambos menor ou igual a 10");  
 }  
 writes ("fim");  
}**

**Ouput simulador P3 (**[**https://p3js.goncalomb.com/**](https://p3js.goncalomb.com/)**) exemplo 2:**

**Uma imagem com texto, captura de ecrã, software, Software de multimédia

Os conteúdos gerados por IA podem estar incorretos.**

**Uma imagem com texto, captura de ecrã, software

Os conteúdos gerados por IA podem estar incorretos.**

**Uma imagem com texto, captura de ecrã, software, Software de multimédia

Os conteúdos gerados por IA podem estar incorretos.**

**Input exemplo 3:**

**void main(void);  
void main(void) {  
 writes("le inteiro 1: ");  
 int x = read();  
 write(x);  
 if (!(x>10)){  
 writes("numero nao e superior a 10");  
 } else {  
 writes("numero e superior a 10");  
 }  
 writes ("fim");  
}**

**Ouput simulador P3 (**[**https://p3js.goncalomb.com/**](https://p3js.goncalomb.com/)**) exemplo 3:**

**Uma imagem com texto, captura de ecrã, software, Software de multimédia

Os conteúdos gerados por IA podem estar incorretos.**

**Uma imagem com texto, captura de ecrã, software, Software de multimédia

Os conteúdos gerados por IA podem estar incorretos.**

**Codigo AS exemplo 1:**

;============== Região de Dados (inicia no endereço 8000h)   
 ORIG 8000h   
  
STR\_LIT\_1 STR 'l','e',' ','v','a','l','o','r',' ','m','a','x','i','m','o',':',' ',0; 'le valor maximo: '  
STR\_LIT\_2 STR 'l','e',' ','s','a','l','t','o',':',' ',0; 'le salto: '  
STR\_LIT\_3 STR 'f','i','m',0 ; 'fim'  
VAR\_1 WORD 0 ; variável 'main'  
VAR\_10 WORD 0 ; variável 'i'  
VAR\_11 WORD 0 ; variável 'L1'  
VAR\_12 WORD 0 ; variável 't3'  
VAR\_13 WORD 0 ; variável 'L3'  
VAR\_14 WORD 0 ; variável 'L2'  
VAR\_15 WORD 0 ; variável 't4'  
VAR\_16 WORD 0 ; variável 'fim'  
VAR\_17 WORD 0 ; variável 't5'  
VAR\_2 WORD 0 ; variável 'le valor maximo: '  
VAR\_3 WORD 0 ; variável 'read'  
VAR\_4 WORD 0 ; variável 't1'  
VAR\_5 WORD 0 ; variável 'x'  
VAR\_6 WORD 0 ; variável 'le salto: '  
VAR\_7 WORD 0 ; variável 't2'  
VAR\_8 WORD 0 ; variável 'y'  
VAR\_9 WORD 1 ; 1  
  
;-------------- Definições de Constantes de sistema   
SP\_ADDRESS EQU FDFFh   
CTRL\_PORT EQU FFFDh ; Porto de controlo do teclado  
IN\_PORT EQU FFFFh ; Porto de entrada de texto (teclado)  
OUT\_PORT EQU FFFEh ; Porto de saída (consola)  
LINEFEED EQU 10 ; Código ASCII da tecla enter na consola (LF)  
  
;============== Região de Código (inicia no endereço 0000h)   
 ORIG 0000h   
 JMP \_start ; jump to main  
  
;-------------- Rotinas   
  
; ----- Função writes("texto"): Imprime string  
WRITES: NOP   
 ; Guarda os registos usados na função   
 PUSH R1   
 PUSH R2   
  
 MOV R1, M[SP+4] ; Endereço da string passado via pilha  
WRITES\_L1: MOV R2, M[R1] ; Lê o carater apontado por R1  
 CMP R2, R0 ; Compara com o terminador  
 JMP.Z WRITES\_LF ; Se for zero, salta para o fim  
 MOV M[OUT\_PORT], R2 ; Escreve o carater no endereço de saída  
 INC R1 ; Avança para o próximo carater  
 JMP WRITES\_L1 ; Repete o ciclo  
WRITES\_LF: MOV R2, LINEFEED ; Muda de linha  
 MOV M[OUT\_PORT], R2   
 ; Restaura os registos usados na função   
 POP R2   
 POP R1   
  
WRITES\_END: RET   
  
; READ: Le inteiro da consola.  
; Return o inteiro em R1.  
READ: NOP   
 PUSH R1 ; Guarda os registos usados na função  
 PUSH R2 ; Guarda os registos usados na função  
; PUSH R3 ; Guarda os registos usados na função  
; PUSH R4 ; Guarda os registos usados na função  
; PUSH R5 ; Guarda os registos usados na função  
; PUSH R6 ; Guarda os registos usados na função  
; PUSH R7 ; Guarda os registos usados na função  
 MOV R4, 0 ; armazena numero  
 MOV R7, 1 ; armazena sinal (1 positivo, -1 negativo)  
READ\_WAIT: NOP   
 MOV R2, M[CTRL\_PORT]; Verifica se há tecla disponível  
 CMP R2, R0   
 BR.Z READ\_WAIT ; Espera enquanto não houver tecla  
 MOV R1, M[IN\_PORT] ; Lê o carácter para R1   
 CMP R1, '-' ; verifica se é sinal  
 JMP.NZ READ\_CONT ; Nao e '-', continua  
 MOV R7, -1 ; armazena sinal (-1 negativo)   
READ\_CONT: NOP   
 ;CMP R1, LINEFEED ; verifica se foi o enter   
 ;BR.Z READ\_RET ; label muito longe!!!   
 ; verificar se é um número entre 0 e 9   
 MOV R2, 30h ; Load ASCII '0'- 30 dec - 1Eh  
 CMP R1, R2 ; Compara R2 ('0') with R1 (char)  
 BR.N READ\_WAIT ; se menor '0', le novamente   
 MOV R2, 39h ; Load ASCII '9' - 39 dec - 27h  
 CMP R2, R1 ; Compara R1 (char) with R2 ('9')  
 BR.N READ\_WAIT ; se maior '9', le novamente   
 MOV R2, 30h ; Load ASCII '0'  
 ; R4 contém o número a ser multiplicado por 10   
 SUB R1, R2 ; R1 tem o valor inteiro digitado  
 MOV R5, R4 ; Copia o valor original para R5 (será X \* 2)  
 SHL R5, 1 ; R5 = R5 \* 2 (desloca R5 1 bit para a esquerda)  
 MOV R6, R4 ; Copia o valor original para R6 (será X \* 8)  
 SHL R6, 3 ; R6 = R6 \* 8 (desloca R6 3 bits para a esquerda)  
 ADD R5, R6 ; R5 = (X \* 2) + (X \* 8) = X \* 10  
 ; O resultado da multiplicação por 10 está agora em R5   
 MOV R4, R1 ; Armazena em R4 numero digitado  
 ADD R4, R5 ; Adiciona R4 com R5 (numero anterior \*10)  
 MOV R5, 0 ; Reset R5  
 MOV R6, 0 ; Reset R6  
READ\_NEXT: MOV R2, M[CTRL\_PORT]; Verifica se há tecla disponível  
 CMP R2, R0   
 BR.Z READ\_NEXT ; Espera enquanto não houver tecla  
 MOV R1, M[IN\_PORT] ; Lê o carácter para R1   
 CMP R1, LINEFEED ; verifica se foi o enter   
 BR.Z READ\_RET ; termina   
 JMP READ\_CONT ; le outro numero   
READ\_RET: NOP   
 CMP R7, 0 ; Se negativo o numero e negativo  
 JMP.NN READ1\_END ; Jump positivo  
 NEG R4 ; Negamos o numero  
READ1\_END: MOV R1, R4 ; Colocamos em R1 o numero  
 MOV M[SP+4], R1 ; Escreve o valor de retorno no espaço do stack  
 ; Restaura os registos usados na função   
; POP R7   
; POP R6   
; POP R5   
; POP R4   
; POP R3   
 POP R2   
 POP R1   
READ\_END: RET   
  
; ----- Função write(x): Imprime valor de variável.  
WRITE: NOP   
 ; Guarda os registos usados na função   
 PUSH R1   
 PUSH R2   
 PUSH R3   
 PUSH R4   
 PUSH R6   
 PUSH R7   
  
 MOV R1, M[SP+8] ; R1 = valor a imprimir  
 MOV R1, M[R1] ; R1 = valor a imprimir  
 MOV R0, 0 ; Tratamento de números negativos  
 CMP R1, R0 ; Compara o número com zero  
 BR.NN WRITE\_POSITIVE ; Se R1 for Não Negativo (>= 0), salta para imprimir.  
 MOV R2, '-' ; Sinal negativo para imprimir.  
 MOV M[OUT\_PORT], R2 ; Se R1 for negativo, imprime o sinal de menos  
 NEG R1 ; Converte R1 para seu valor absoluto (positivo)  
WRITE\_POSITIVE: NOP   
 MOV R7, 10000 ; Divisor inicial (10^4)  
 MOV R6, R0 ; Flag: dígito já impresso (0 = ainda não)  
  
WRITE\_L1: MOV R2, R1 ; R2 = valor atual  
 MOV R3, R7 ; R3 = divisor  
 DIV R2, R3 ; R2 = quociente (dígito), R3 = resto  
 CMP R6, R0 ; Já imprimimos algum dígito?  
 BR.NZ WRITE\_L2 ; Se sim, imprime sempre  
 CMP R2, R0   
 BR.Z WRITE\_L3 ; Se dígito é 0 e nada impresso, salta  
  
WRITE\_L2: ADD R2, 48 ; Converte para ASCII  
 MOV M[OUT\_PORT], R2 ; Escreve dígito  
 MOV R6, 1 ; Marca que começámos a imprimir  
  
WRITE\_L3: MOV R1, R3 ; Atualiza valor com o resto  
 MOV R4, 10   
 DIV R7, R4 ; R7 = R7 / 10 (próximo divisor)  
 CMP R7, R0   
 BR.NZ WRITE\_L1   
 ; Caso número seja 0 imprime '0'   
 CMP R6, R0   
 BR.NZ WRITE\_LF   
 MOV R1, '0'   
 MOV M[OUT\_PORT], R1   
WRITE\_LF: MOV R2, LINEFEED ; Muda de linha  
 MOV M[OUT\_PORT], R2   
 ; Restaura os registos usados na função   
 POP R7   
 POP R6   
 POP R4   
 POP R3   
 POP R2   
 POP R1   
  
WRITE\_END: RET   
  
;-------------- Programa Principal   
\_start: NOP   
 MOV R7, SP\_ADDRESS   
 MOV SP, R7 ; Define o Stack Pointer  
  
main: NOP   
; writes STR\_LIT\_1 -------------------------  
 PUSH STR\_LIT\_1 ; Endereço da string passado via pilha  
 CALL WRITES ; Chama a rotina  
 POP R0   
  
; call read -------------------------  
 PUSH R0 ; Reserva espaço para retorno  
 CALL READ ; Chama a rotina  
 POP M[VAR\_4] ; Atribui o valor à variável  
  
 MOV R1, M[VAR\_4]   
 MOV M[VAR\_5], R1   
; write VAR\_4 -------------------------  
 PUSH VAR\_4 ; Endereço do valor passado via pilha  
 CALL WRITE ; Chama a rotina  
 POP R0 ; Limpa a pilha  
  
; writes STR\_LIT\_2 -------------------------  
 PUSH STR\_LIT\_2 ; Endereço da string passado via pilha  
 CALL WRITES ; Chama a rotina  
 POP R0   
  
; call read -------------------------  
 PUSH R0 ; Reserva espaço para retorno  
 CALL READ ; Chama a rotina  
 POP M[VAR\_7] ; Atribui o valor à variável  
  
 MOV R1, M[VAR\_7]   
 MOV M[VAR\_8], R1   
; write VAR\_7 -------------------------  
 PUSH VAR\_7 ; Endereço do valor passado via pilha  
 CALL WRITE ; Chama a rotina  
 POP R0 ; Limpa a pilha  
  
 MOV R1, M[VAR\_9]   
 MOV M[VAR\_10], R1   
L1: NOP   
 MOV R1, M[VAR\_10]   
 MOV R2, M[VAR\_4]   
 CMP R1, R2 ; ZCNO flags affected  
 JMP.P L3   
L2: NOP   
; write VAR\_10 -------------------------  
 PUSH VAR\_10 ; Endereço do valor passado via pilha  
 CALL WRITE ; Chama a rotina  
 POP R0 ; Limpa a pilha  
  
 MOV R1, M[VAR\_10]   
 MOV R2, M[VAR\_7]   
 ADD R1, R2 ; ZCNO flags affected  
 MOV M[VAR\_15], R1   
 MOV R1, M[VAR\_15]   
 MOV M[VAR\_10], R1   
 JMP L1   
L3: NOP   
; writes STR\_LIT\_3 -------------------------  
 PUSH STR\_LIT\_3 ; Endereço da string passado via pilha  
 CALL WRITES ; Chama a rotina  
 POP R0   
  
; halt -------------------------   
 BR Fim ; Fim com loop infinito  
  
Fim: BR Fim

**Codigo AS exemplo 2:**

;============== Região de Dados (inicia no endereço 8000h)   
 ORIG 8000h   
  
STR\_LIT\_1 STR 'l','e',' ','i','n','t','e','i','r','o',' ','1',':',' ',0; 'le inteiro 1: '  
STR\_LIT\_2 STR 'l','e',' ','i','n','t','e','i','r','o',' ','2',':',' ',0; 'le inteiro 2: '  
STR\_LIT\_3 STR 'a','m','b','o','s',' ','s','u','p','e','r','i','o','r','e','s',' ','a',' ','1','0',0; 'ambos superiores a 10'  
STR\_LIT\_4 STR 'u','m',' ','n','u','m','e','r','o',' ','o','u',' ','a','m','b','o','s',' ','m','e','n','o','r',' ','o','u',' ','i','g','u','a','l',' ','a',' ','1','0',0; 'um numero ou ambos menor ou igual a 10'  
STR\_LIT\_5 STR 'f','i','m',0 ; 'fim'  
VAR\_1 WORD 0 ; variável 'main'  
VAR\_10 WORD 0 ; variável 't3'  
VAR\_11 WORD 0 ; variável 'L4'  
VAR\_12 WORD 0 ; variável 't4'  
VAR\_13 WORD 1 ; 1  
VAR\_14 WORD 0 ; variável 't5'  
VAR\_15 WORD 0 ; variável 'L5'  
VAR\_16 WORD 0 ; 0  
VAR\_17 WORD 0 ; variável 'L2'  
VAR\_18 WORD 0 ; variável 'L1'  
VAR\_19 WORD 0 ; variável 'ambos superiores a 10'  
VAR\_2 WORD 0 ; variável 'le inteiro 1: '  
VAR\_20 WORD 0 ; variável 'L3'  
VAR\_21 WORD 0 ; variável 'um numero ou ambos menor ou igual a 10'  
VAR\_22 WORD 0 ; variável 'fim'  
VAR\_23 WORD 0 ; variável 't6'  
VAR\_3 WORD 0 ; variável 'read'  
VAR\_4 WORD 0 ; variável 't1'  
VAR\_5 WORD 0 ; variável 'x'  
VAR\_6 WORD 0 ; variável 'le inteiro 2: '  
VAR\_7 WORD 0 ; variável 't2'  
VAR\_8 WORD 0 ; variável 'y'  
VAR\_9 WORD 10 ; 10  
  
;-------------- Definições de Constantes de sistema   
SP\_ADDRESS EQU FDFFh   
CTRL\_PORT EQU FFFDh ; Porto de controlo do teclado  
IN\_PORT EQU FFFFh ; Porto de entrada de texto (teclado)  
OUT\_PORT EQU FFFEh ; Porto de saída (consola)  
LINEFEED EQU 10 ; Código ASCII da tecla enter na consola (LF)  
  
;============== Região de Código (inicia no endereço 0000h)   
 ORIG 0000h   
 JMP \_start ; jump to main  
  
;-------------- Rotinas   
  
; ----- Função writes("texto"): Imprime string  
WRITES: NOP   
 ; Guarda os registos usados na função   
 PUSH R1   
 PUSH R2   
  
 MOV R1, M[SP+4] ; Endereço da string passado via pilha  
WRITES\_L1: MOV R2, M[R1] ; Lê o carater apontado por R1  
 CMP R2, R0 ; Compara com o terminador  
 JMP.Z WRITES\_LF ; Se for zero, salta para o fim  
 MOV M[OUT\_PORT], R2 ; Escreve o carater no endereço de saída  
 INC R1 ; Avança para o próximo carater  
 JMP WRITES\_L1 ; Repete o ciclo  
WRITES\_LF: MOV R2, LINEFEED ; Muda de linha  
 MOV M[OUT\_PORT], R2   
 ; Restaura os registos usados na função   
 POP R2   
 POP R1   
  
WRITES\_END: RET   
  
; READ: Le inteiro da consola.  
; Return o inteiro em R1.  
READ: NOP   
 PUSH R1 ; Guarda os registos usados na função  
 PUSH R2 ; Guarda os registos usados na função  
; PUSH R3 ; Guarda os registos usados na função  
; PUSH R4 ; Guarda os registos usados na função  
; PUSH R5 ; Guarda os registos usados na função  
; PUSH R6 ; Guarda os registos usados na função  
; PUSH R7 ; Guarda os registos usados na função  
 MOV R4, 0 ; armazena numero  
 MOV R7, 1 ; armazena sinal (1 positivo, -1 negativo)  
READ\_WAIT: NOP   
 MOV R2, M[CTRL\_PORT]; Verifica se há tecla disponível  
 CMP R2, R0   
 BR.Z READ\_WAIT ; Espera enquanto não houver tecla  
 MOV R1, M[IN\_PORT] ; Lê o carácter para R1   
 CMP R1, '-' ; verifica se é sinal  
 JMP.NZ READ\_CONT ; Nao e '-', continua  
 MOV R7, -1 ; armazena sinal (-1 negativo)   
READ\_CONT: NOP   
 ;CMP R1, LINEFEED ; verifica se foi o enter   
 ;BR.Z READ\_RET ; label muito longe!!!   
 ; verificar se é um número entre 0 e 9   
 MOV R2, 30h ; Load ASCII '0'- 30 dec - 1Eh  
 CMP R1, R2 ; Compara R2 ('0') with R1 (char)  
 BR.N READ\_WAIT ; se menor '0', le novamente   
 MOV R2, 39h ; Load ASCII '9' - 39 dec - 27h  
 CMP R2, R1 ; Compara R1 (char) with R2 ('9')  
 BR.N READ\_WAIT ; se maior '9', le novamente   
 MOV R2, 30h ; Load ASCII '0'  
 ; R4 contém o número a ser multiplicado por 10   
 SUB R1, R2 ; R1 tem o valor inteiro digitado  
 MOV R5, R4 ; Copia o valor original para R5 (será X \* 2)  
 SHL R5, 1 ; R5 = R5 \* 2 (desloca R5 1 bit para a esquerda)  
 MOV R6, R4 ; Copia o valor original para R6 (será X \* 8)  
 SHL R6, 3 ; R6 = R6 \* 8 (desloca R6 3 bits para a esquerda)  
 ADD R5, R6 ; R5 = (X \* 2) + (X \* 8) = X \* 10  
 ; O resultado da multiplicação por 10 está agora em R5   
 MOV R4, R1 ; Armazena em R4 numero digitado  
 ADD R4, R5 ; Adiciona R4 com R5 (numero anterior \*10)  
 MOV R5, 0 ; Reset R5  
 MOV R6, 0 ; Reset R6  
READ\_NEXT: MOV R2, M[CTRL\_PORT]; Verifica se há tecla disponível  
 CMP R2, R0   
 BR.Z READ\_NEXT ; Espera enquanto não houver tecla  
 MOV R1, M[IN\_PORT] ; Lê o carácter para R1   
 CMP R1, LINEFEED ; verifica se foi o enter   
 BR.Z READ\_RET ; termina   
 JMP READ\_CONT ; le outro numero   
READ\_RET: NOP   
 CMP R7, 0 ; Se negativo o numero e negativo  
 JMP.NN READ1\_END ; Jump positivo  
 NEG R4 ; Negamos o numero  
READ1\_END: MOV R1, R4 ; Colocamos em R1 o numero  
 MOV M[SP+4], R1 ; Escreve o valor de retorno no espaço do stack  
 ; Restaura os registos usados na função   
; POP R7   
; POP R6   
; POP R5   
; POP R4   
; POP R3   
 POP R2   
 POP R1   
READ\_END: RET   
  
; ----- Função write(x): Imprime valor de variável.  
WRITE: NOP   
 ; Guarda os registos usados na função   
 PUSH R1   
 PUSH R2   
 PUSH R3   
 PUSH R4   
 PUSH R6   
 PUSH R7   
  
 MOV R1, M[SP+8] ; R1 = valor a imprimir  
 MOV R1, M[R1] ; R1 = valor a imprimir  
 MOV R0, 0 ; Tratamento de números negativos  
 CMP R1, R0 ; Compara o número com zero  
 BR.NN WRITE\_POSITIVE ; Se R1 for Não Negativo (>= 0), salta para imprimir.  
 MOV R2, '-' ; Sinal negativo para imprimir.  
 MOV M[OUT\_PORT], R2 ; Se R1 for negativo, imprime o sinal de menos  
 NEG R1 ; Converte R1 para seu valor absoluto (positivo)  
WRITE\_POSITIVE: NOP   
 MOV R7, 10000 ; Divisor inicial (10^4)  
 MOV R6, R0 ; Flag: dígito já impresso (0 = ainda não)  
  
WRITE\_L1: MOV R2, R1 ; R2 = valor atual  
 MOV R3, R7 ; R3 = divisor  
 DIV R2, R3 ; R2 = quociente (dígito), R3 = resto  
 CMP R6, R0 ; Já imprimimos algum dígito?  
 BR.NZ WRITE\_L2 ; Se sim, imprime sempre  
 CMP R2, R0   
 BR.Z WRITE\_L3 ; Se dígito é 0 e nada impresso, salta  
  
WRITE\_L2: ADD R2, 48 ; Converte para ASCII  
 MOV M[OUT\_PORT], R2 ; Escreve dígito  
 MOV R6, 1 ; Marca que começámos a imprimir  
  
WRITE\_L3: MOV R1, R3 ; Atualiza valor com o resto  
 MOV R4, 10   
 DIV R7, R4 ; R7 = R7 / 10 (próximo divisor)  
 CMP R7, R0   
 BR.NZ WRITE\_L1   
 ; Caso número seja 0 imprime '0'   
 CMP R6, R0   
 BR.NZ WRITE\_LF   
 MOV R1, '0'   
 MOV M[OUT\_PORT], R1   
WRITE\_LF: MOV R2, LINEFEED ; Muda de linha  
 MOV M[OUT\_PORT], R2   
 ; Restaura os registos usados na função   
 POP R7   
 POP R6   
 POP R4   
 POP R3   
 POP R2   
 POP R1   
  
WRITE\_END: RET   
  
;-------------- Programa Principal   
\_start: NOP   
 MOV R7, SP\_ADDRESS   
 MOV SP, R7 ; Define o Stack Pointer  
  
main: NOP   
; writes STR\_LIT\_1 -------------------------  
 PUSH STR\_LIT\_1 ; Endereço da string passado via pilha  
 CALL WRITES ; Chama a rotina  
 POP R0   
  
; call read -------------------------  
 PUSH R0 ; Reserva espaço para retorno  
 CALL READ ; Chama a rotina  
 POP M[VAR\_4] ; Atribui o valor à variável  
  
 MOV R1, M[VAR\_4]   
 MOV M[VAR\_5], R1   
; write VAR\_4 -------------------------  
 PUSH VAR\_4 ; Endereço do valor passado via pilha  
 CALL WRITE ; Chama a rotina  
 POP R0 ; Limpa a pilha  
  
; writes STR\_LIT\_2 -------------------------  
 PUSH STR\_LIT\_2 ; Endereço da string passado via pilha  
 CALL WRITES ; Chama a rotina  
 POP R0   
  
; call read -------------------------  
 PUSH R0 ; Reserva espaço para retorno  
 CALL READ ; Chama a rotina  
 POP M[VAR\_7] ; Atribui o valor à variável  
  
 MOV R1, M[VAR\_7]   
 MOV M[VAR\_8], R1   
; write VAR\_7 -------------------------  
 PUSH VAR\_7 ; Endereço do valor passado via pilha  
 CALL WRITE ; Chama a rotina  
 POP R0 ; Limpa a pilha  
  
 MOV R1, M[VAR\_4]   
 MOV R2, 10   
 CMP R1, R2 ; ZCNO flags affected  
 JMP.N L4   
 JMP.Z L4   
 MOV R1, M[VAR\_7]   
 MOV R2, 10   
 CMP R1, R2 ; ZCNO flags affected  
 JMP.N L4   
 JMP.Z L4   
 MOV R1, M[VAR\_13]   
 MOV M[VAR\_14], R1   
 JMP L5   
L4: NOP   
 MOV R1, M[VAR\_16]   
 MOV M[VAR\_14], R1   
L5: NOP   
 JMP.N L2   
 JMP.Z L2   
L1: NOP   
; writes STR\_LIT\_3 -------------------------  
 PUSH STR\_LIT\_3 ; Endereço da string passado via pilha  
 CALL WRITES ; Chama a rotina  
 POP R0   
  
 JMP L3   
L2: NOP   
; writes STR\_LIT\_4 -------------------------  
 PUSH STR\_LIT\_4 ; Endereço da string passado via pilha  
 CALL WRITES ; Chama a rotina  
 POP R0   
  
L3: NOP   
; writes STR\_LIT\_5 -------------------------  
 PUSH STR\_LIT\_5 ; Endereço da string passado via pilha  
 CALL WRITES ; Chama a rotina  
 POP R0   
  
; halt -------------------------   
 BR Fim ; Fim com loop infinito  
  
Fim: BR Fim

**Codigo AS exemplo 3:**

;============== Região de Dados (inicia no endereço 8000h)   
 ORIG 8000h   
  
STR\_LIT\_1 STR 'l','e',' ','i','n','t','e','i','r','o',' ','1',':',' ',0; 'le inteiro 1: '  
STR\_LIT\_2 STR 'n','u','m','e','r','o',' ','n','a','o',' ','e',' ','s','u','p','e','r','i','o','r',' ','a',' ','1','0',0; 'numero nao e superior a 10'  
STR\_LIT\_3 STR 'n','u','m','e','r','o',' ','e',' ','s','u','p','e','r','i','o','r',' ','a',' ','1','0',0; 'numero e superior a 10'  
STR\_LIT\_4 STR 'f','i','m',0 ; 'fim'  
VAR\_1 WORD 0 ; variável 'main'  
VAR\_10 WORD 0 ; variável 'L1'  
VAR\_11 WORD 0 ; variável 'numero nao e superior a 10'  
VAR\_12 WORD 0 ; variável 'L3'  
VAR\_13 WORD 0 ; variável 'numero e superior a 10'  
VAR\_14 WORD 0 ; variável 'fim'  
VAR\_15 WORD 0 ; variável 't4'  
VAR\_2 WORD 0 ; variável 'le inteiro 1: '  
VAR\_3 WORD 0 ; variável 'read'  
VAR\_4 WORD 0 ; variável 't1'  
VAR\_5 WORD 0 ; variável 'x'  
VAR\_6 WORD 10 ; 10  
VAR\_7 WORD 0 ; variável 't2'  
VAR\_8 WORD 0 ; variável 't3'  
VAR\_9 WORD 0 ; variável 'L2'  
  
;-------------- Definições de Constantes de sistema   
SP\_ADDRESS EQU FDFFh   
CTRL\_PORT EQU FFFDh ; Porto de controlo do teclado  
IN\_PORT EQU FFFFh ; Porto de entrada de texto (teclado)  
OUT\_PORT EQU FFFEh ; Porto de saída (consola)  
LINEFEED EQU 10 ; Código ASCII da tecla enter na consola (LF)  
  
;============== Região de Código (inicia no endereço 0000h)   
 ORIG 0000h   
 JMP \_start ; jump to main  
  
;-------------- Rotinas   
  
; ----- Função writes("texto"): Imprime string  
WRITES: NOP   
 ; Guarda os registos usados na função   
 PUSH R1   
 PUSH R2   
  
 MOV R1, M[SP+4] ; Endereço da string passado via pilha  
WRITES\_L1: MOV R2, M[R1] ; Lê o carater apontado por R1  
 CMP R2, R0 ; Compara com o terminador  
 JMP.Z WRITES\_LF ; Se for zero, salta para o fim  
 MOV M[OUT\_PORT], R2 ; Escreve o carater no endereço de saída  
 INC R1 ; Avança para o próximo carater  
 JMP WRITES\_L1 ; Repete o ciclo  
WRITES\_LF: MOV R2, LINEFEED ; Muda de linha  
 MOV M[OUT\_PORT], R2   
 ; Restaura os registos usados na função   
 POP R2   
 POP R1   
  
WRITES\_END: RET   
  
; READ: Le inteiro da consola.  
; Return o inteiro em R1.  
READ: NOP   
 PUSH R1 ; Guarda os registos usados na função  
 PUSH R2 ; Guarda os registos usados na função  
; PUSH R3 ; Guarda os registos usados na função  
; PUSH R4 ; Guarda os registos usados na função  
; PUSH R5 ; Guarda os registos usados na função  
; PUSH R6 ; Guarda os registos usados na função  
; PUSH R7 ; Guarda os registos usados na função  
 MOV R4, 0 ; armazena numero  
 MOV R7, 1 ; armazena sinal (1 positivo, -1 negativo)  
READ\_WAIT: NOP   
 MOV R2, M[CTRL\_PORT]; Verifica se há tecla disponível  
 CMP R2, R0   
 BR.Z READ\_WAIT ; Espera enquanto não houver tecla  
 MOV R1, M[IN\_PORT] ; Lê o carácter para R1   
 CMP R1, '-' ; verifica se é sinal  
 JMP.NZ READ\_CONT ; Nao e '-', continua  
 MOV R7, -1 ; armazena sinal (-1 negativo)   
READ\_CONT: NOP   
 ;CMP R1, LINEFEED ; verifica se foi o enter   
 ;BR.Z READ\_RET ; label muito longe!!!   
 ; verificar se é um número entre 0 e 9   
 MOV R2, 30h ; Load ASCII '0'- 30 dec - 1Eh  
 CMP R1, R2 ; Compara R2 ('0') with R1 (char)  
 BR.N READ\_WAIT ; se menor '0', le novamente   
 MOV R2, 39h ; Load ASCII '9' - 39 dec - 27h  
 CMP R2, R1 ; Compara R1 (char) with R2 ('9')  
 BR.N READ\_WAIT ; se maior '9', le novamente   
 MOV R2, 30h ; Load ASCII '0'  
 ; R4 contém o número a ser multiplicado por 10   
 SUB R1, R2 ; R1 tem o valor inteiro digitado  
 MOV R5, R4 ; Copia o valor original para R5 (será X \* 2)  
 SHL R5, 1 ; R5 = R5 \* 2 (desloca R5 1 bit para a esquerda)  
 MOV R6, R4 ; Copia o valor original para R6 (será X \* 8)  
 SHL R6, 3 ; R6 = R6 \* 8 (desloca R6 3 bits para a esquerda)  
 ADD R5, R6 ; R5 = (X \* 2) + (X \* 8) = X \* 10  
 ; O resultado da multiplicação por 10 está agora em R5   
 MOV R4, R1 ; Armazena em R4 numero digitado  
 ADD R4, R5 ; Adiciona R4 com R5 (numero anterior \*10)  
 MOV R5, 0 ; Reset R5  
 MOV R6, 0 ; Reset R6  
READ\_NEXT: MOV R2, M[CTRL\_PORT]; Verifica se há tecla disponível  
 CMP R2, R0   
 BR.Z READ\_NEXT ; Espera enquanto não houver tecla  
 MOV R1, M[IN\_PORT] ; Lê o carácter para R1   
 CMP R1, LINEFEED ; verifica se foi o enter   
 BR.Z READ\_RET ; termina   
 JMP READ\_CONT ; le outro numero   
READ\_RET: NOP   
 CMP R7, 0 ; Se negativo o numero e negativo  
 JMP.NN READ1\_END ; Jump positivo  
 NEG R4 ; Negamos o numero  
READ1\_END: MOV R1, R4 ; Colocamos em R1 o numero  
 MOV M[SP+4], R1 ; Escreve o valor de retorno no espaço do stack  
 ; Restaura os registos usados na função   
; POP R7   
; POP R6   
; POP R5   
; POP R4   
; POP R3   
 POP R2   
 POP R1   
READ\_END: RET   
  
; ----- Função write(x): Imprime valor de variável.  
WRITE: NOP   
 ; Guarda os registos usados na função   
 PUSH R1   
 PUSH R2   
 PUSH R3   
 PUSH R4   
 PUSH R6   
 PUSH R7   
  
 MOV R1, M[SP+8] ; R1 = valor a imprimir  
 MOV R1, M[R1] ; R1 = valor a imprimir  
 MOV R0, 0 ; Tratamento de números negativos  
 CMP R1, R0 ; Compara o número com zero  
 BR.NN WRITE\_POSITIVE ; Se R1 for Não Negativo (>= 0), salta para imprimir.  
 MOV R2, '-' ; Sinal negativo para imprimir.  
 MOV M[OUT\_PORT], R2 ; Se R1 for negativo, imprime o sinal de menos  
 NEG R1 ; Converte R1 para seu valor absoluto (positivo)  
WRITE\_POSITIVE: NOP   
 MOV R7, 10000 ; Divisor inicial (10^4)  
 MOV R6, R0 ; Flag: dígito já impresso (0 = ainda não)  
  
WRITE\_L1: MOV R2, R1 ; R2 = valor atual  
 MOV R3, R7 ; R3 = divisor  
 DIV R2, R3 ; R2 = quociente (dígito), R3 = resto  
 CMP R6, R0 ; Já imprimimos algum dígito?  
 BR.NZ WRITE\_L2 ; Se sim, imprime sempre  
 CMP R2, R0   
 BR.Z WRITE\_L3 ; Se dígito é 0 e nada impresso, salta  
  
WRITE\_L2: ADD R2, 48 ; Converte para ASCII  
 MOV M[OUT\_PORT], R2 ; Escreve dígito  
 MOV R6, 1 ; Marca que começámos a imprimir  
  
WRITE\_L3: MOV R1, R3 ; Atualiza valor com o resto  
 MOV R4, 10   
 DIV R7, R4 ; R7 = R7 / 10 (próximo divisor)  
 CMP R7, R0   
 BR.NZ WRITE\_L1   
 ; Caso número seja 0 imprime '0'   
 CMP R6, R0   
 BR.NZ WRITE\_LF   
 MOV R1, '0'   
 MOV M[OUT\_PORT], R1   
WRITE\_LF: MOV R2, LINEFEED ; Muda de linha  
 MOV M[OUT\_PORT], R2   
 ; Restaura os registos usados na função   
 POP R7   
 POP R6   
 POP R4   
 POP R3   
 POP R2   
 POP R1   
  
WRITE\_END: RET   
  
;-------------- Programa Principal   
\_start: NOP   
 MOV R7, SP\_ADDRESS   
 MOV SP, R7 ; Define o Stack Pointer  
  
main: NOP   
; writes STR\_LIT\_1 -------------------------  
 PUSH STR\_LIT\_1 ; Endereço da string passado via pilha  
 CALL WRITES ; Chama a rotina  
 POP R0   
  
; call read -------------------------  
 PUSH R0 ; Reserva espaço para retorno  
 CALL READ ; Chama a rotina  
 POP M[VAR\_4] ; Atribui o valor à variável  
  
 MOV R1, M[VAR\_4]   
 MOV M[VAR\_5], R1   
; write VAR\_4 -------------------------  
 PUSH VAR\_4 ; Endereço do valor passado via pilha  
 CALL WRITE ; Chama a rotina  
 POP R0 ; Limpa a pilha  
  
 MOV R1, M[VAR\_4]   
 MOV R2, 10   
 CMP R1, R2 ; ZCNO flags affected  
 CMP R2, R1   
 JMP.N L2   
 JMP.Z L2   
L1: NOP   
; writes STR\_LIT\_2 -------------------------  
 PUSH STR\_LIT\_2 ; Endereço da string passado via pilha  
 CALL WRITES ; Chama a rotina  
 POP R0   
  
 JMP L3   
L2: NOP   
; writes STR\_LIT\_3 -------------------------  
 PUSH STR\_LIT\_3 ; Endereço da string passado via pilha  
 CALL WRITES ; Chama a rotina  
 POP R0   
  
L3: NOP   
; writes STR\_LIT\_4 -------------------------  
 PUSH STR\_LIT\_4 ; Endereço da string passado via pilha  
 CALL WRITES ; Chama a rotina  
 POP R0   
  
; halt -------------------------   
 BR Fim ; Fim com loop infinito  
  
Fim: BR Fim

**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/>