

Introdução à Software Básico: Montadores - parte 1

Departamento de Ciência da Computação
Instituto de Ciências Exatas
Universidade de Brasília

Montadores

- 1 Montadores - Características adicionais / Funções avançadas
- 2 Algoritmos utilizados na construção de um Montador
- 3 Código Relocável

Reserva de espaço para variáveis

- A diretiva SPACE apresentada antes não possui operando e sempre reserva uma palavra de memória.
- É possível imaginar esta mesma diretiva recebendo como parâmetro o número de palavras de memória a serem reservadas. Por exemplo:
XY: SPACE 10
- Reserva 10 palavras consecutivas de memória. O símbolo "XY" é associado ao endereço da primeira palavra reservada.

Reserva de espaço para variáveis

- Os montadores comerciais suportam expressões aritméticas no lugar dos operandos.
- Estas expressões são calculadas em tempo de montagem e o resultado é incluído no código de máquina.
- Por exemplo:
LOAD $XY + 2$
- Carrega para o acumulador o conteúdo da segunda palavra de memória após o rótulo XY

Contadores de posição

- A maioria dos montadores oferece uma diretiva para alterar o valor do contador de posições.
- Por exemplo, considere um microcomputador com EPROM nos primeiros 32K de memória e RAM nos restantes 32K (para esse microcomputador é necessário separar as instruções das variáveis).
- Nesse caso, todo programa teria a seguinte forma:

ORG 0

instruções

...

ORG 32768

variáveis

...

Definição de sinônimos

- A maioria dos montadores permite que o programador defina sinônimos para valores numéricos ou mesmo simbólicos.
- Uma diretiva normalmente oferecida é o EQU ("equate"), que cria um sinônimo textual para um símbolo.
- Por exemplo,
TAM: EQU 10
VAL: CONST TAM

Conditional assembly

- O montador “condicional” permite que se determine, durante o processo de montagem, se uma determinada parte do código será incluída ou não
- Isso pode ser feito de maneira simples através de uma diretiva, como mostra abaixo
- A diretiva *IF* instruirá o montador a incluir a linha seguinte somente se a expressão

IF expressão

FLAG EQU 1

...

IF FLAG

JMP XXX

- O uso de *IF* aqui é semelhante ao *#ifdef* que usamos em C

Macros são construções semelhantes a subrotinas

- 1 Elas associam um nome a um trecho de código,
- 2 Permitem que esse trecho seja referenciado pelo nome (quando for necessário a sua execução) e
- 3 Permitem a passagem de parâmetros.

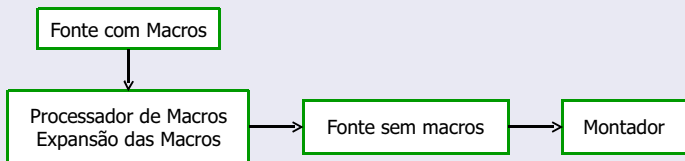
Macros são construções semelhantes a subrotinas no sentido de que

- 1 Em uma subrotina o código aparece na memória do computador somente uma vez e cada chamada desvia a execução para esse código.
- 2 Em uma macro, o código aparece na memória tantas vezes quantas a macro é chamada, isto é, em cada chamada é inserido no programa o código da macro.

Salomon, Assemblers and Loaders, Elsevier Science, 1993,
<http://www.davidsalomon.name/assem.advertis/AssemAd.html>, cap.4

- O fato de o código da macro ser inserido no lugar da chamada faz com que não exista economia de memória como no caso da subrotina.
- Entretanto, uma macro é mais rápida que uma subrotina, pois:
 - não existe necessidade de salvar endereço de retorno,
 - transferir execução e
 - retornar (resgatar o endereço salvo)
- Nos primórdios da computação, a economia de memória era a motivação principal para o uso de subrotinas. Hoje, a organização do código (em módulos) é provavelmente o motivo maior para seu uso.

- O processamento de macros é uma etapa anterior à primeira passagem do montador (pode ser visto como o pré-processador em C)



- Na prática, o processador de macros pode ser embutido na primeira passagem do montador.
- Isso evita que o conjunto processador de macros + montador faça um total de 3 passagens sobre o programa fonte.

Exemplo

Algoritmo 1 Exemplo 1

```
1: TROCA: MACRO
2:     COPY A, TEMP
3:     COPY B, A
4:     COPY TEMP, B
5:     ENDMACRO
6:     . . . ; parte sem macros 2
7:     TROCA
8:     . . . ; parte sem macros 3
9:     TROCA
```

- A macro “TROCA” para implementar uma operação tipo “permuta” entre duas posições de memória.
- Duas novas diretivas são introduzidas: *MACRO* e *ENDMACRO*.

- Toda vez que o processador de macros encontra uma chamada de macro ele realiza a sua expansão, que é a substituição da chamada pelo corpo da macro.
- O programa fonte que o montador recebe é o código original, com exceção das definições de macros que são eliminadas e das chamadas que são expandidas.

```
TROCA:  MACRO
        COPY A, TEMP
        COPY B, A
        COPY TEMP, B
        ENDMACRO
        . . . ; parte sem macros 2
TROCA
        . . . ; parte sem macros 3
TROCA
```

```
        . . .
        . . . ; parte sem macros 2
        COPY A, TEMP
        COPY B, A
        COPY TEMP, B
        . . . ; parte sem macros 3
        COPY A, TEMP
        COPY B, A
        COPY TEMP, B
```

Características adicionais dos montadores - MACROS

- Tipicamente, os processadores de macros exigem que uma macro seja definida antes de ser chamada.
- Essa restrição permite que o processador de macros realize seu trabalho com apenas uma passagem sobre o programa fonte.
- A macro TROCA, como definida antes, sempre troca os valores das posições A e B, usando TEMP como variável auxiliar.
- Uma macro que trabalha somente com posições fixas é pouco útil, por isso as macros admitem a utilização de parâmetros.

MACROS – Com parâmetros

```
SWAP:  MACRO  &A, &B, &T
        COPY  &A, &T
        COPY  &B, &A
        COPY  &T, &B
        ENDMACRO

        . . .
        SWAP X, Y, Z

        . . .
        SWAP R, S, T

        . . .
```

Tabelas usadas por um processador de macros

- O processador de macros utiliza duas tabelas:
 - **Macro Name Table (MNT)** – contém os nomes das macros definidas no programa
 - **Macro Definition Table (MDT)** – contém os corpos das macros
- Para cada macro, a MNT informa o número de argumentos e o local da MDT onde está o seu corpo.
- Na MDT (no corpo da macro) os argumentos formais são substituídos por marcadores de índice da forma *#i*.

<u>MNT:</u>	<u>MDT:</u>
.
(linha i) SWAP 3 25	(linha 25) COPY #1, #3
	COPY #2, #1
. . .	COPY #3, #2
	ENDMACRO

- ❶ Ler próxima linha do código fonte
- ❷ Se é MACRO, ler a definição completa da Macro e armazena na MDT. Ir para 1.
- ❸ Se é uma outra diretiva de pré-processamento, executar e ir para 1.
- ❹ Se é o nome de uma macro, expandi-la a partir da definição na MDT, substituindo parâmetros e escrever um novo código fonte. Ir para 1.
- ❺ Em qualquer outro caso, escreve a linha no novo código fonte. Ir para 1.
- ❻ Se eof, fim da passagem 0.

Fluxograma - Passagem 0 do Montador

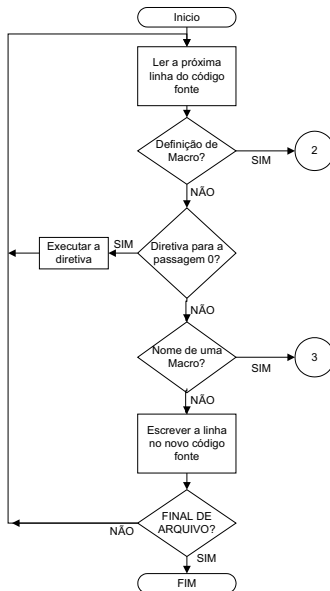


Figura: Passagem 0

Fluxograma - Passagem 0 do Montador

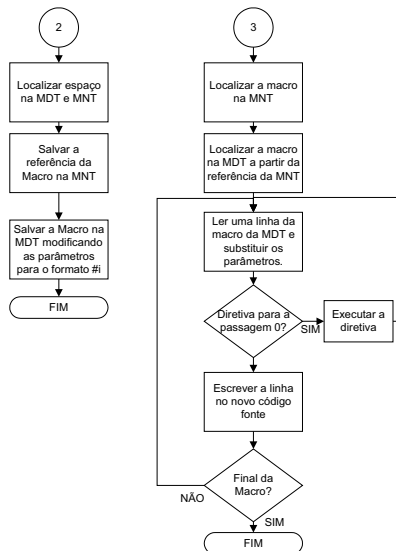


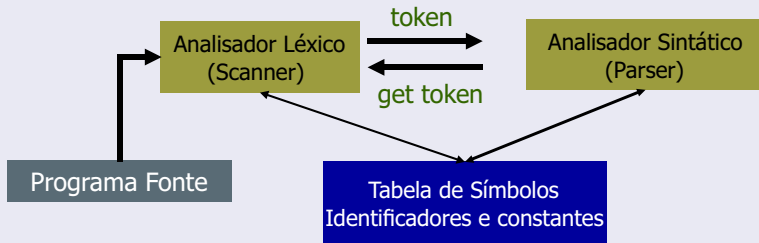
Figura: Passagem 0

Construção de um Montador

- Como já vimos, os tradutores (incluídos aí, os montadores, montadores-cruzados, compiladores, etc) possuem uma estrutura básica que é independente da linguagem a ser traduzida ou do programa objeto a ser gerado
- Desta forma, podemos ter idéia de como criar um montador
- Sabemos que ele terá duas partes principais
 - A análise e
 - A Síntese

Análise

- Inicialmente teremos que reconhecer os tokens, isto é, as partes do programa que fazem sentido, descartando os comentários, espaços desnecessários e linhas em branco – análise Léxica



Análise

- Os programas escritos na nossa linguagem montadora possuem uma estrutura que facilita a leitura do código
`<rótulo>: <operação> <operandos> ; <comentários>`
- Se utilizarmos o algoritmo de duas passagens, a primeira irá identificar os rótulos, o que torna-se fácil em virtude dos “:”, e incluí-los na tabela de símbolos. Devemos também identificar a instrução, porém com o único objetivo de atualizar o contador de posições.
- Na segunda passagem: ler a operação para verificar os formatos de instruções válidos:
 - Formato 1: Opcode
 - Formato 2: Opcode Endereço
 - Formato 3: Opcode Endereço_1 Endereço_2

Análise

- Caso o opcode ou endereços não estiverem definidos corretamente nas tabelas auxiliares, uma mensagem de erro será mostrada.
- Desta forma, a segunda passagem seria mais próxima da análise Sintática.
- Caso não exista erro, o nosso “montador” poderia então gerar o código intermediário (ou objeto, dependendo do caso)
- Dado o conjunto de instruções que temos, o nosso montador poderia ser um cross-assembler (montador cruzado) de forma que o código objeto gerado fosse de fato um código em formato objeto da máquina destino.
 - Essa última parte, estaria relacionada com a síntese.
 - Poderíamos gerar um “código intermediário” para então gerar o código para a máquina alvo.

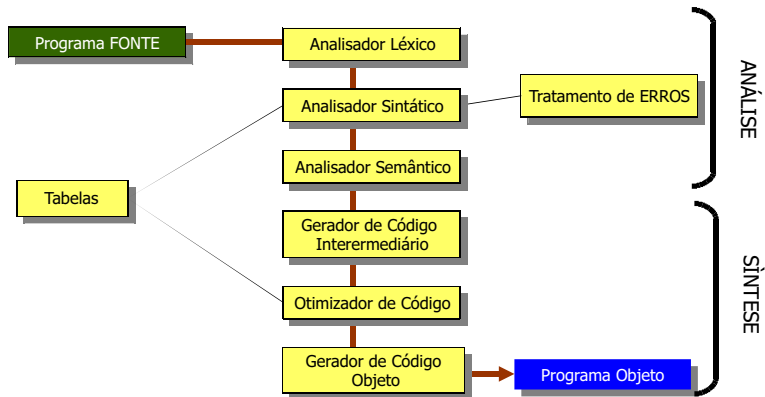
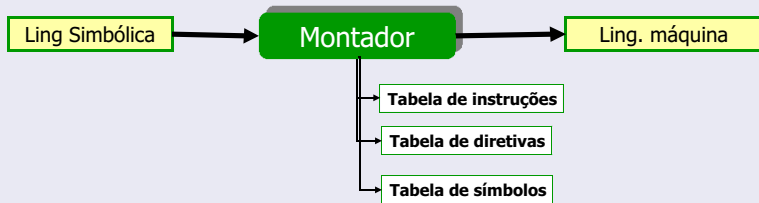


Figura: Estrutura de um Tradutor

- Como vimos, os montadores utilizam várias tabelas para auxiliar na verificação e geração do código objeto



- Em geral, os montadores/compiladores, gastam mais da metade do tempo buscando informações que estão contidas em tabelas!

- Certamente, algoritmos que realizem essas “buscas” de forma rápida são necessários.

Algoritmos de Busca

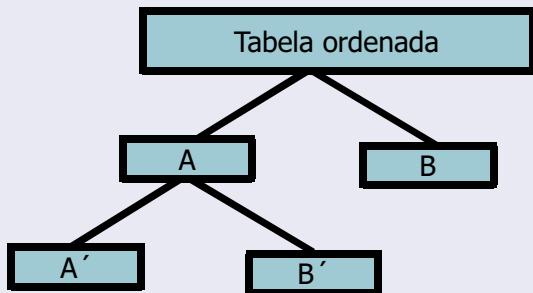
- O processo de busca normalmente recebe como entrada código (nome), e tenta encontrar na tabela,
- Retorna os valores associados ao código de busca quando encontrado ou informa que o mesmo não foi encontrado.

1 - Busca Sequencial

- Consiste em ler todos os itens da tabela (array), um a um, até que o item que estamos buscando, ou a fim da tabela, seja encontrado
- Em média, esse algoritmo faz $(N + 1)/2$ (complexidade $\Theta(n)$) comparações até encontrar o item que estamos buscando (N = número de itens na tabela)
- Se a tabela aumenta, aumenta também o tempo de busca!

2 - Busca Binária

- Necessita que a tabela esteja ordenada
 - Dividimos a tabela em duas partes A, e B
 - Comparamos o valor que estamos buscando com o ítem inicial da parte B
 - Se o item que estamos buscando for menor, então descartamos a parte B e repetimos o processo para a parte A
 - Caso contrário, descartamos a parte A, e repetimos o processo para a parte B.



2 - Busca Binária

- Na busca binária a complexidade do caso médio (e pior caso) é $O(\log_2 n)$.
- Suponha que a tabela que estamos pesquisando tenha 1000 itens.
- Se utilizarmos a busca sequencial, teremos em média 500 comparações até encontrar o valor que estamos buscando.
- Com a busca binária, o número de comparações é reduzido a 10 comparações neste caso.
- A única desvantagem é que temos que ordenar antes!

Ordenação

- Existem vários algoritmos de ordenação e, dependendo do caso, alguns algoritmos podem ser mais rápidos do que outros.
- Um simples algoritmo de ordenação pode ser construído da seguinte forma (ordenação por seleção – selection sort)

Algoritmo 2 Selection Sort

```
for  $k \leftarrow 0$  to  $N - 2$  do  
   $posMenor \leftarrow k$   
  for  $i \leftarrow k + 1$  to  $N - 1$  do {Percorre todo o vetor}  
    if  $numero[i] < numero[posMenor]$  then  
       $posMenor \leftarrow i$   
    end if  
  end for  
  if  $posMenor \neq k$  then  
     $aux \leftarrow numero[posMenor]$   
     $numero[posMenor] \leftarrow numero[k]$   
     $numero[k] \leftarrow aux$   
  end if  
end for
```



SelectionSort

- Identificamos o menor (ou maior) elemento no segmento do vetor que contém os elementos ainda não selecionados.
- Trocamos o elemento identificado com o primeiro elemento do segmento.
- Atualizamos o tamanho do segmento (diminuímos uma posição).
- Interrompemos o processo quando o segmento contiver apenas um elemento.
- Eficiência:
 - Pior caso $O(n^2)$.
 - Caso médio $O(n^2)$.

Quicksort

- Arrays grandes precisam de algoritmos de ordenação mais eficientes que o *selection sort*. Por exemplo o *Quicksort*.
 - Eficiência no pior caso: $O(n^2)$
 - Eficiência no caso médio: $O(n \log_2 n)$

STEP 1: choose a pivot



STEP 2: lesser values go to the left, greater values go to the right

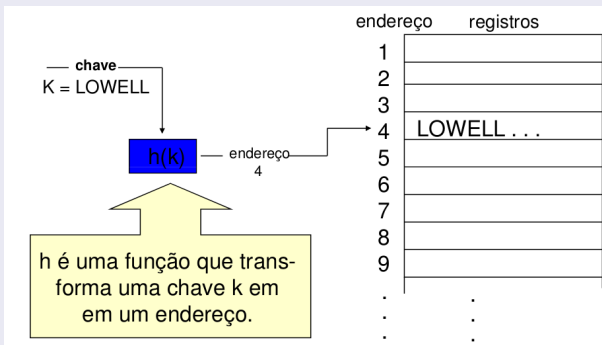


STEP 3: repeat from step 1 with the two sub-lists



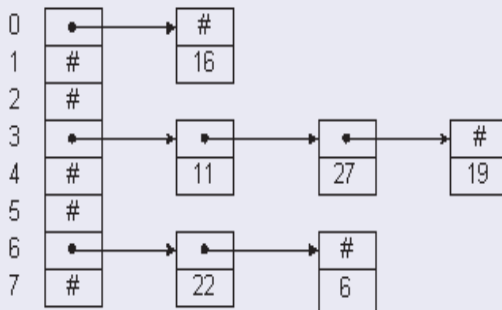
Hashing

- Porém, seria melhor se nossa tabela estivesse organizada usando *Hashing*.
- Suponha que temos uma coleção de N itens e uma função que gera um código único para cada item



Hashing

- A vantagem da tabela hashing é que o tempo de procura é $O(1)$.
- Porém, é muito difícil achar uma função hash que gere um hashing sem colisões.
 - Suponha que temos uma coleção de N itens e uma função que gera um código único para cada item
 - Cada elemento apontará para um lista encadeada contendo os itens daquela chave
 - A função hash para este exemplo é *chave mod 8*.



Hashing

- A vantagem da tabela hashing encadeada é que o tempo de procura é $O(1)$, ou $O(n)$ no pior caso.
- Se tivermos uma boa função hash, isto é uma função que distribua os itens de forma uniforme, teremos aproximadamente o mesmo número em cada lista.
- Desta forma, a procura por cada item terá um número de comparações semelhantes

Código Relocável

- Até agora, o nosso montador que utilizamos nos exemplos gera instruções de máquina com endereços absolutos, isto é, com o endereço que eles irão ocupar na memória quando o código for executado.
- Ao desenvolver um programa maior, é conveniente montá-lo em partes, e armazenar cada uma destas partes separadamente e juntar (ligar) as mesmas antes da execução do código
- Porém, se todos as partes iniciam no endereço zero, como temos feito até agora, apenas uma parte poderá ser executada

Código Relocável

- Algumas partes do código objeto são independentes do local em que o programa é carregado na memória.
- Isto é, os valores não são alterados no processo de carga.

Exemplo

VETOR:	STOP
	SPACE 5
VAR:	CONST 10

Gera

end 0.	14
end 1.	??
end 2.	??
end 3.	??
end 4.	??
end 5.	??
end 6.	10

- O código acima pode ser carregado para qualquer outro endereço (diferente de zero), sem necessidade de alteração do código

Exemplo

- Os valores que não são alterados no processo de carga são ditos **absolutos**.
- Para facilitar a visualização, representaremos essa informação através da letra "a" colocada logo após cada valor absoluto:

```
STOP  
VETOR: SPACE 5  
VAR:  CONST 10
```

Gera

```
end 0. 14a  
end 1. ??a  
end 2. ??a  
end 3. ??a  
end 4. ??a  
end 5. ??a  
end 6. 10a
```

Código Relocável

- Os símbolos que representam endereços de memória são ditos **relativos**.
- Isto é, o valor de um símbolo relativo depende do local onde o programa é carregado.
- Representaremos essa informação através da letra “r” colocada logo após cada valor absoluto:

Exemplo

```
LOAD  VAR
ADD   K1
STORE VAR
OUTPUT VAR
STOP
VAR:  SPACE 1
K1:   CONST 1
```

Gera

```
end 0. 10a 09r
end 2. 01a 10r
end 4. 11a 09r
end 6. 13a 09r
end 8. 14a
end 9. ??a
end 10. 01a
```

Código Relocável

- Na maioria dos sistemas operacionais, o endereço de carga não é conhecido até o momento da execução do programa
- Se, em tempo de carga, a gerência de memória do sistema operacional determina que o programa seja carregado no endereço 200, basta somar o valor 200 a todos os campos relativos do programa.

Exemplo

Código fonte		
	LOAD	VAR
	ADD	K1
	STORE	VAR
	OUTPUT	VAR
	STOP	
VAR:	SPACE	1
K1:	CONST	1

Código objeto c/inf rel		
end	0. 10a	09r
end	2. 01a	10r
end	4. 11a	09r
end	6. 13a	09r
end	8. 14a	
end	9. ??a	
end	10. 01a	

Código relocado		
end	200. 10	209
end	202. 01	210
end	204. 11	209
end	206. 13	209
end	208. 14	
end	209. ??	
end	210. 01	

Próxima Aula

Carregador e Ligador