Introdução à Software Básico: Montadores - parte 1

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

Sumário

Montadores

Montador

- Nos primórdios da computação, os primeiros programas foram escritos em Linguagem de Máquina, isto é, as instruções eram armazenadas diretamente na memória, em formato binário
- Esse trabalho de programação era feito através de botões ou ligações por cabos.
- Já na segunda geração das linguagens de programação surgem os primeiros tradutores. Que traduzem uma linguagem formada por símbolos mnemônicos (linguagem de Montagem) e a linguagem binária da máquina:
  - os chamados Montadores (do inglês, "Assembler").

Montador

- Como vimos anteriormente, o processo de "montagem" recebe como entrada um arquivo texto contendo o código fonte do programa em linguagem de montagem e gera como saída um arquivo binário contendo o código de máquina e outras informações relevantes para a execução do código gerado.
- Para entendermos o processo de montagem, vamos precisar de algumas "ferramentas", as quais veremos a seguir.

Uma máquina hipotética

Descrição da máquina hipotética

- Para nos auxiliar no estudo de montadores (e também ligadores e carregadores), nós vamos definir uma máquina hipotética e um conjunto (reduzidíssimo) de instruções para ela.
- Embora o nosso conjunto de instruções seja bem reduzido, veremos que programas completos podem ser criados
- O objetivo aqui é apresentar a linguagem assembly de maneira informal, complementado posteriormente.

Descrição da máquina hipotética

- A nossa máquina hipotética terá:
  - 1 registrador acumulador de 16 bits. Chamaremos de **ACC**.
  - 1 registrador apontador de instruções de 16 bits. Chamaremos de **Program counter** ou **PC**.
  - Memória de 216 palavras de 16 bits;
  - A nossa máquina hipotética pode reconhecer 3 formatos de instruções:
    - Formato 1: Opcode
    - Formato 2: Opcode Endereço
    - Formato 3: Opcode Endereço_1 Endereço_2

Código de Operação ou OPCODE

- Identifica a operação a ser realizada pelo processador.
- É o campo da instrução cujo **valor binário** identifica a operação a ser realizada (é o código binário da instrução) .
- Este código é a entrada no **decodificador de instruções** na unidade de controle.
- Cada instrução deverá ter um código único que a identifique.

Uma máquina hipotética

mnemônico

código da máquina

espaço que a inst ocupará em mem

Opcode Simbólico	Opcode Numérico	Tamanho (palavras)	Ação
ADD	01	2	ACC ← ACC + mem(OP)
SUB	02	2	ACC ← ACC - mem(OP)
MUL	03	2	ACC ← ACC × mem(OP)
DIV	04	2	ACC ← ACC ÷ mem(OP)
JMP	05	2	PC ← OP
JMPN	06	2	Se ACC<0 então PC ← OP
JMPP	07	2	Se ACC>0 então PC ← OP
JMPZ	08	2	Se ACC=0 então PC ← OP
COPY	09	3	mem(OP2) ← mem(OP1)
LOAD	10	2	ACC ← mem(OP)
STORE	11	2	mem(OP) ← ACC
INPUT	12	2	mem(OP) ← entrada
OUTPUT	13	2	saída ← mem(OP)
STOP	14	1	Suspende a execução

Figura: Conjunto de instruções da máquina hipotética

Uma máquina hipotética

Conjunto de instruções

- O conjunto de instruções de alguns processadores, como por exemplo o Intel 8080, não possui instruções para multiplicação ou divisão
- Portanto, um programa em linguagem de máquina, nestes processadores, não pode fazer multiplicação ou divisão diretamente (somente através de combinação de outras instruções).
- Já um programa em linguagem de nível mais alto pode implementar comandos de multiplicação (ou divisão) que combinem uma série de instruções binárias do conjunto de instruções para fazer uma multiplicação (ou divisão) através de repetidas somas (subtrações) ou por deslocamento de bits (Ex. C).

Formato de uma linha fonte:

- O nosso programa fonte, em pseudo-assembler, utiliza o seguinte formato:
  <rótulo>: <operação> <operandos> ;<comentários>

Exemplo

Algoritmo 1 Exemplo 1

```
1: ROT: INPUT N1
2:          COPY N2,N1 ;isso é um comentário
3: N1: SPACE
4: N2: SPACE
```

Exemplo

O programa a seguir utiliza o conjunto de instruções definidos acima para ler dois números da entrada padrão (teclado) e imprimir a soma dos mesmos na saída padrão (vídeo):

- $N3 = N1 + N2;$

Algoritmo 2 Exemplo 2

```
INPUT     N1      ; Lê o primeiro número
INPUT     N2      ; Lê o segundo número
LOAD      N1      ; Carrega N1 no acumulador
ADD       N2      ; Soma está no acumulador
STORE     N3      ; Coloca soma N1+N2 em N3
OUTPUT    N3      ; Escreve o resultado
STOP              ; Termina a execução do programa
N1:       SPACE   ; Reserva espaço para o primeiro número
N2:       SPACE   ; Reserva espaço para o segundo número
N3:       SPACE   ; Reserva espaço para o resultado
```

Conjunto de instruções

- São instruções para o próprio montador, isto é, uma instrução da linguagem assembler para que o montador modifique o seu comportamento ou realize alguma tarefa.
- Exemplo:
  - diretivas podem ser usadas para alocar espaço para dados ou variáveis;
  - instruir o montador a incluir arquivos fontes;
  - estabelecer o ponto de ínicio do programa, etc.
- Inicialmente vamos utilizar duas diretivas que usaremos nos exemplos e nos nossos programas:
  - **CONST** → Instrui o montador a gerar dados em memória (leitura apenas)
  - **SPACE** → Reserva espaço para os dados (leitura e gravação)

Exemplo

Exemplo 3

- Crie um programa, utilizando o conjunto de instruções definidos da nossa máquina hipotética que compute a área de um triângulo.
- O programa deverá ler os dados com INPUT e mostrar o resultado (Sabemos que a área é dada por: $(\text{Base} \times \text{Altura})/2$)

Exemplo

Solução

Algoritmo 3 Exemplo 3

```
INPUT     B        ; Lê a base
INPUT     H        ; Lê a altura
LOAD      B        ; Carrega B no acumulador
MULT      H        ; ACC tem  BxH
DIV       DOIS     ; ACC tem o resultado de (B+H)/2
STORE     R        ; Coloca soma (B+H)/2 em R
OUTPUT    R        ; Escreve o resultado
STOP               ; Termina a execução do programa
B:        SPACE    ; Reserva espaço para a base
H:        SPACE    ; Reserva espaço para a altura
R:        SPACE    ; Reserva espaço para o resultado
DOIS:     CONST  2 ; Reserva espaço para uma constante
```

Nosso Pseudo-Asm

- A linguagem assembler que estamos utilizando é também chamada de linguagem simbólica pura, pois cada declaração gera exatamente uma instrução de máquina.

```
Código fonte:
INPUT   N1
INPUT   N2
LOAD    N1
ADD     N2
STORE   N3
OUTPUT  N3
STOP
N1:     SPACE
N2:     SPACE
N3:     SPACE
```

Código Gerado		
End 0	12	13
End 2	12	14
End 4	10	13
End 6	01	14
End 8	11	15
End 10	13	15
End 12	14	
End 13	xx	
End 14	xx	
End 15	xx	

Figura: Exemplo de Montagem

Nota

Valores numéricos aparecem em decimal e o valor "xx" representa um valor qualquer ocupando uma palavra de memória (16 bits).

Funções de um Montador

Tarefas básicas do montador

- Tradução de pseudo-instruções por *opcodes* numéricos e expansão de "macros"
- Representação do programa em linguagem de máquina: instruções e dados
- Reservar espaço para os dados
- Determinação dos endereços de memória referenciados pelo programa
- Registro das informações necessárias à **ligação** do programa:
  - Tabela de Definições - símbolos externos usados no módulo
  - Tabela de Uso – Símbolos exportados e seus atributos

O processo de Montagem (Assembler)

- Como vimos, um programa em linguagem *assembly* puro consiste de uma série de declarações de uma linha
- Desta forma, seria natural imaginar um montador que lesse o código fonte (*assembly*), linha a linha, e gerasse o código de máquina em um outro arquivo
- Esse processo poderia ser repetido até que todas as declarações do arquivo fonte fossem traduzidas em linguagem de máquina

O processo de Montagem (Assembler)

- Infelizmente, o processo não é tão simples.
- Para ver isso, imagine uma situação na qual temos um desvio (*jump*) no programa para um rótulo R dentro do programa.
- O montador não poderá gerar o código de máquina para essa instrução até que o endereço do rótulo R seja conhecido.
- O rótulo R pode estar definido no final do programa, o que leva o montador a ler o programa fonte inteiro para poder encontrar o rótulo (e então gerar o endereço)

O processo de Montagem (Assembler)

- Outro exemplo, e no exercício anterior onde os identificadores para as variáveis (os rótulos de N1,N2 e N3) foram declarados no final

```
Código fonte:
INPUT   N1
INPUT   N2
LOAD    N1
ADD     N2
STORE   N3
OUTPUT  N3
STOP
N1:     SPACE
N2:     SPACE
N3:     SPACE
```

	Código Gerado	
End 0	12	13
End 2	12	14
End 4	10	13
End 6	01	14
End 8	11	15
End 10	13	15
End 12	14	
End 13	xx	
End 14	xx	
End 15	xx	

O processo de Montagem (Assembler)

- O problema que vimos no slide anterior é conhecido como *forward reference problem* – ou problema de referência posterior
- Como resolver a questão de "referências posteriores"?
- Existem duas formas:
  1. Ler o código uma única vez utilizando mecanismos para resolver o problema de referências posteriores
     - Torna o montador mais complexo
  2. Ler o código duas vezes
     - A primeira leitura identifica os símbolos
     - A segunda, gera o código de máquina
     - Aumenta o Tempo de leitura. Necessitamos ler todo o programa fonte duas vezes

O processo de Montagem (Assembler)

- Por sua simplicidade, a maioria dos assembler utiliza o processo de ler o código fonte duas vezes
- Cada leitura do código fonte é chamada de **passagem**
- O tradutor que lê o código fonte duas vezes é chamado de tradutor de duas passagens (ou *two-pass translator*)
- Seguindo a mesma nomenclatura, um montador que lê o código fonte duas vezes é chamado de montador de duas passagens (*two-pass assembler*)

Estrutura Interna de um Montador

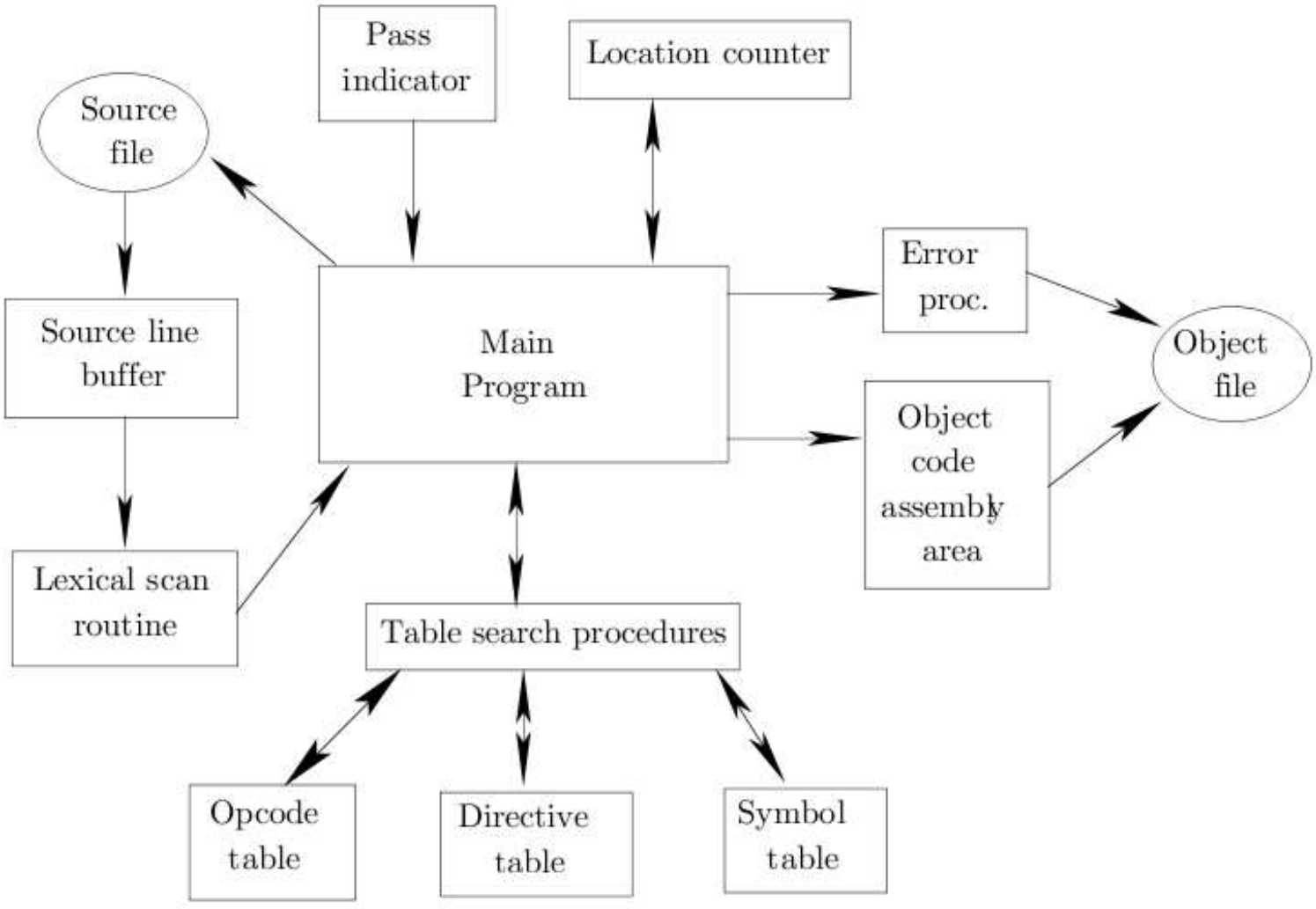


Figura: Estrutura Interna de um Montador

Montador de duas passagens

- Iniciaremos a nossa discussão do pelo algoritmo de duas passagens
- O processo executado pelo montador em cada passagem é descrito como segue:
- **Primeira Passagem**
  - Na primeira passagem, o montador coleta informações de definições de rótulos, símbolos, etc, e os armazena em uma tabela
- **Segunda Passagem**
  - Na segunda passagem, os valores (endereços) dos símbolos já são conhecidos e cada declaração pode, então, ser "montada"

Primeira Passagem

- A função principal da primeira passagem é a construção da Tabela de Símbolos
- **Tabela de símbolos (TS)**: contém todos os símbolos definidos no programa e seus atributos
- **Símbolo**: Um símbolo é um rótulo ou o nome de uma variável declarada com um nome simbólico

Exemplo

Algoritmo 4 Exemplo 4

```
1: ...
2: N1: CONST 0
3: N2: SPACE
4: L: OUTPUT X
```

Estruturas de Dados

- Além da tabela de símbolos, o montador utiliza as seguintes estruturas de dados:
- **Tabela de instruções**:
  - contém os opcodes simbólicos acompanhadas das informações para gerar código de máquina
  - Basicamente, esta tabela contém o formato da instrução e opcode numérico.
  - Esta tabela está pronta, isto é, faz parte do código do montador.

Estruturas de Dados

- **Contador de linhas**:
  - indica a linha do programa fonte que está sendo analisada no momento (útil para impressão de mensagens de erro)
- **Contador de posições (location counter)**:
  - indica a posição de memória a ser ocupada (preenchida) pelo código de máquina.
- **Tabela de símbolos (TS)**:
  - contém todos os símbolos definidos no programa e seus atributos. Essa tabela é criada na 1ª passagem e usada na 2ª
- **Tabela de diretivas**:
  - contém as diretivas da linguagem. Em geral, para cada diretiva existe uma rotina que implementa as ações correspondentes. Esta tabela também está pronta.
  - Nos exemplos, nós vamos assumir que as diretivas CONST and SPACE ocupam 1 palavra de memória (16 bits)

Algoritmo da primeira passagem:

```
contador_posição = 0
contador_linha = 1
Enquanto arquivo fonte não chegou ao fim,  faça:
    Obtém uma linha do fonte
    Separa os elementos da linha:
        rótulo, operação, operandos, comentários
    Ignora os comentários
    Se existe rótulo:
        Procura rótulo na TS (Tabela de Símbolos)
    Se achou: Erro → símbolo redefinido
    Senão:      Insere rótulo e contador_posição na TS
    Procura operação na  tabela de instruções
    Se achou:
        contador_posição = contador_posição + tamanho da instrução
    Senão:
        Procura operação na tabela de diretivas
        Se achou:
                chama subrotina que executa a diretiva
                contador_posição = valor retornado pela subrotina
        Senão: Erro, operação não identificada
        contador_linha = contador_linha + 1
```

Figura: Primeira Passagem

Exemplo Primeira Passagem 1

- **Código fonte:**

```
INPUT          N1
INPUT          N2
LOAD           N1
ADD            N2
STORE          N3
OUTPUT         N3
STOP
N1:    SPACE
N2:    SPACE
N3:    SPACE
```

- **Tabela de Símbolos**

```
N1:    13
N2:    14
N3:    15
```

Exemplo Primeira Passagem 2

- Considere o programa abaixo e assuma que a entrada seja 8.
- Para o programa você deverá:
  - Mostrar a saída
  - Tente identificar o que está sendo computado
  - Utilizar o algoritmo de primeira passada para criar a tabela de símbolos

```
         COPY            ZERO,   OLDER
         COPY            ONE,    OLD
         INPUT           LIMIT
         OUTPUT          OLD
FRONT:   LOAD            OLDER
         ADD             OLD
         STORE           NEW
         SUB             LIMIT
         JMPP            FINAL
         OUTPUT          NEW
         COPY            OLD,    OLDER
         COPY            NEW,    OLD
         JMP             FRONT
FINAL:   OUTPUT          LIMIT
         STOP
ZERO:    CONST           0
ONE:     CONST           1
OLDER:   SPACE
OLD:     SPACE
NEW:     SPACE
LIMIT:   SPACE
```

Exemplo

Solução

```
         COPY          ZERO,    OLDER
         COPY          ONE,     OLD
         INPUT         LIMIT
         OUTPUT        OLD
FRONT:   LOAD          OLDER
         ADD           OLD
         STORE         NEW
         SUB           LIMIT
         JMPP          FINAL
         OUTPUT        NEW
         COPY          OLD,     OLDER
         COPY          NEW,     OLD
         JMP           FRONT
FINAL:   OUTPUT        LIMIT
         STOP
ZERO:    CONST         0
ONE:     CONST         1
OLDER:   SPACE
OLD:     SPACE
NEW:     SPACE
LIMIT:   SPACE
```

Mostra a Série de Fibonacci
8 → 1,1,2,3,5,8

Símbolo	Valor
FRONT	10
FINAL	30
ZERO	33
ONE	34
OLDER	35
OLD	36
NEW	37
LIMIT	38

Exemplo

Exemplo Primeira Passagem 3

- Crie um programa que compute o fatorial de um número.
- Utilize o algoritmo da primeira passagem para gerar a tabela de símbolos.

Exemplo

Solução

- Esse programa computa o fatorial do valor lido pela entrada padrão

Código Fonte

```
        INPUT   N
        LOAD    N
FAT:    SUB     ONE
        JNPZ    FIM
        STORE   AUX
        MULT    N
        STORE   N
        LOAD    AUX
        JNP     FAT
FIM:    OUTPUT  N
        STOP
AUX:    SPACE
N:      SPACE
ONE:    CONST   1
```

Input→ 5

Output→ **120**

N	ACC	AUX
5	5	
5	5-1=4	4
20	4*5=20	4
20	4	4

N	ACC	AUX
20	4	4
20	4-1=3	3
60	3*20=60	3
60	3	3

N	ACC	AUX
60	3	3
60	3-1=2	2
120	2*60=120	2
120	2	2

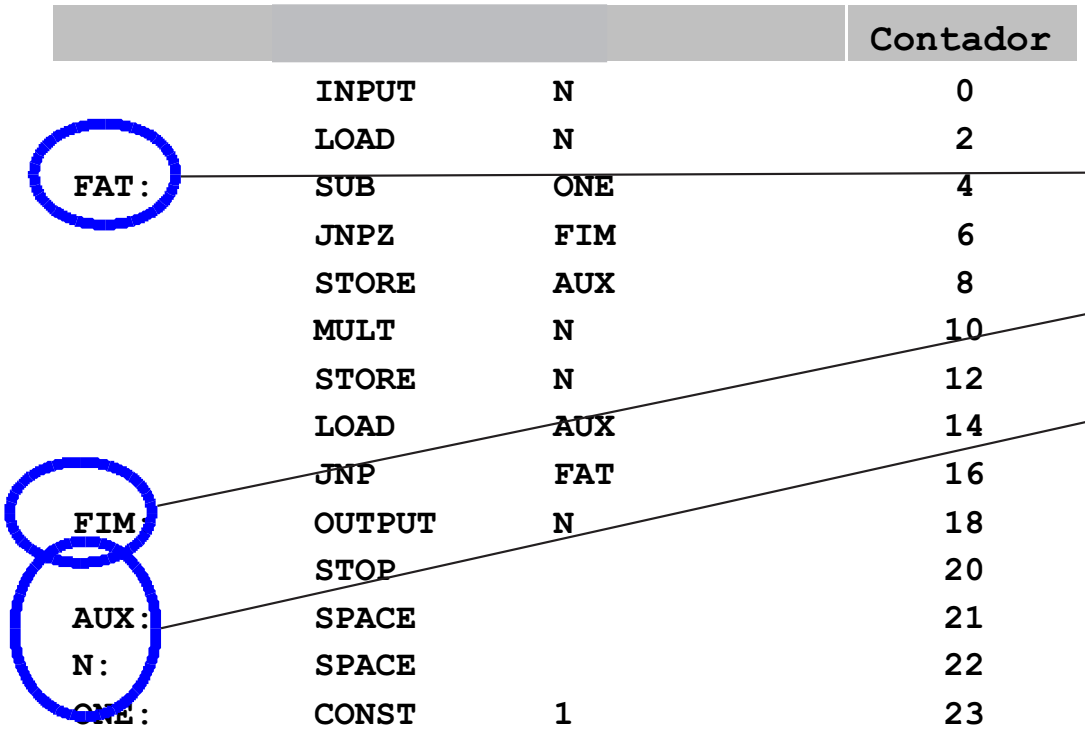
N	ACC	AUX
120	2	2
120	2-1=1	1
120	1*120=120	1
120	1	1

N	ACC	AUX
120	1	1
120	1-1=0	1

Exemplo

Solução

- Vamos ilustrar o algoritmo de duas passagens com um programa que computa o fatorial de um número lido do fluxo de entrada padrão



			Contador
	INPUT	N	0
	LOAD	N	2
FAT:	SUB	ONE	4
	JNPZ	FIM	6
	STORE	AUX	8
	MULT	N	10
	STORE	N	12
	LOAD	AUX	14
	JNP	FAT	16
FIM:	OUTPUT	N	18
	STOP		20
AUX:	SPACE		21
N:	SPACE		22
ONE:	CONST	1	23

Tabela de Símbolos	
Símbolo	**Valor**
FAT	4
FIM	18
AUX	21
N	22
ONE	23

Próxima Aula

Montador passagem única

Introdução à Software Básico: Montadores - parte 2

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

Sumário

Montadores

1. Montador de duas passagem: algoritmo da segunda passagem
2. Montador de passagem única

O Algoritmo de duas passagens

- Fluxo de informações para a geração da tabela de símbolos em um montador de duas passagens

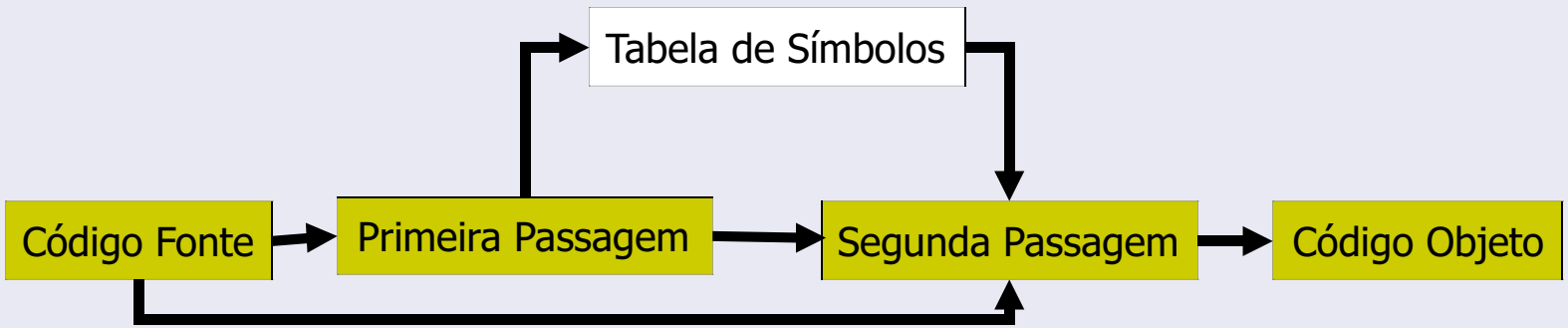


Figura: Fluxo do montador de duas passagens

O Algoritmo de duas passagens

- O processo executado pelo montador em cada passagem é descrito como segue:
- **Primeira Passagem**
  - Na primeira passagem, o montador coleta informações de definições de rótulos, símbolos, etc, e os armazena na tabela símbolos:
    - Símbolo, valor (endereço)
- **Segunda Passagem**
  - Na segunda passagem, os valores (endereços) dos símbolos já são conhecidos e as declarações podem então ser "montadas"

Algoritmo da segunda passagem:

```
Contador_posição = 0
Contador_linha = 1
Enquanto arquivo fonte não chegou ao fim,  faça:
   Obtém uma linha do fonte
   Separa os elementos da linha: rótulo,operação,operandos,comentários
   Ignora o rótulo e os comentários

   Para cada operando que é símbolo
   Procura operando na TS
        Se não achou: Erro, símbolo indefinido
   Procura operação na tabela de instruções
   Se achou:
        contador_posição = contador_posição + tamanho da instrução
        Se número e tipo dos operandos está correto então
                gera código objeto conforme formato da instrução
        Senão: Erro, operando inválido
   Senão:
        Procura operação na tabela de diretivas
        Se achou:
                Chama subrotina que executa a diretiva
                Contador_posição = valor retornado pela subrotina
        Senão: Erro, operação não identificada
   Contador_linha = contador_linha + 1
```

Figura: Algoritmo da segunda passagem

O Algoritmo de duas passagens

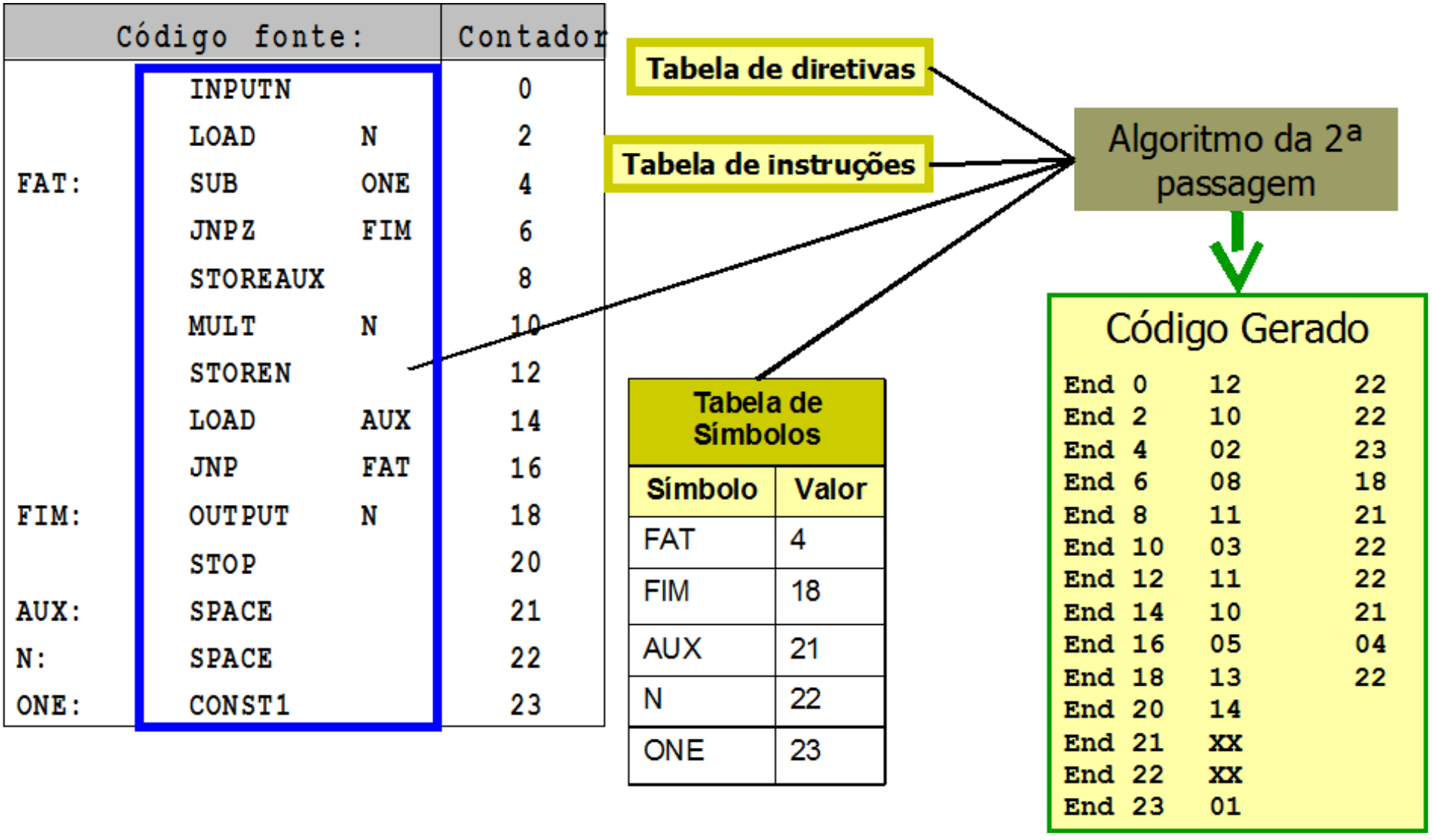


Figura: Algoritmo da segunda passagem

Exemplo

- Gere o código objeto do exercício anterior – série de Fibonacci

```
          COPY      ZERO,     OLDER
          COPY      ONE,      OLD
          INPUT     LIMIT
          OUTPUT    OLD
FRONT:    LOAD      OLDER
          ADD       OLD
          STORE     NEW
          SUB       LIMIT
          JMPP      FINAL
          OUTPUT    NEW
          COPY      OLD,      OLDER
          COPY      NEW,      OLD
          JMP       FRONT
FINAL:    OUTPUT    LIMIT
          STOP
ZERO:     CONST     0
ONE:      CONST     1
OLDER:    SPACE
OLD:      SPACE
NEW:      SPACE
LIMIT:    SPACE
```

TABELA DE SÍMBOLOS

Simbolo	Valor
FRONT	10
FINAL	30
ZERO	33
ONE	34
OLDER	35
OLD	36
NEW	37
LIMIT	38

O Algoritmo de duas passagens

Solução

```
     CÓDIGO GERADO
end.  0: 09 33 35
end.  3: 09 34 36
end.  6: 12 38
end.  8: 13 36
end. 10: 10 35
end. 12: 01 36
end. 14: 11 37
end. 16: 02 38
end. 18: 07 30
end. 20: 13 37
end. 22: 09 36 35
end. 25: 09 37 36
end. 28: 05 10
end. 30: 13 38
end. 32: 14
end. 33: 0
end. 34: 1
end. 35: xx
end. 36: xx
end. 37: xx
end. 38: xx
```

OBS: O valor "xx" representa um valor qualquer. Normalmente zero é usado nestes casos

Observações sobre os endereços das instruções

- O código foi gerado para o endereço zero de memória. Caso fosse necessário gerar o código para outro endereço bastaria alterar o valor inicial do contador de posições.
- Uma solução mais geral é gerar o código sempre para o endereço zero, mas informar também as posições (palavras) do código que contém endereços.
- A indicação das posições que contém endereços é conhecida como **informação de relocação**.
- Nesse caso, a tarefa de acertar os endereços em função do ponto de carga (isto é, a relocação do programa) fica para o **carregador**.

O Algoritmo de uma passagem

- O algoritmo de duas passagens soluciona o problema de referências posteriores de forma bastante simples
- Se todos os símbolos referidos na linha lida (instrução simbólica) já estão definidos, então a instrução de máquina é gerada diretamente, de forma completa
- Obviamente, um montador de um único passo nem sempre poderá determinar o endereço de um símbolo assim que encontrar o mesmo.
- **Como criar então um montador de uma única passagem?**

O Algoritmo de uma passagem

Como criar então um montador de uma única passagem?

- Um símbolo ainda não definido é inserido na tabela de símbolos como havíamos feito até agora
- A instrução de máquina também é gerada.
- Porém, o campo correspondente ao símbolo indefinido fica para ser preenchido **mais tarde**.

Análise das referências posteriores

- Quando uma referência é feita a um rótulo, o montador irá procurar o símbolo da tabela de símbolo da mesma forma que fizemos antes. Aqui, temos várias possibilidades.
  - Se todos os símbolos referidos na linha lida (instrução simbólica) já estão definidos, então a instrução de máquina é gerada diretamente, de forma completa. Neste caso o símbolo estará marcado como "definido=true" na Tabela de Símbolos (TS) e o valor (endereço) já é conhecido.
  - Se o símbolo não estiver na TS, então inserimos o mesmo e marcamos "definido = false". Um campo é criado na TS para apontar para uma lista de símbolos indefinidos.

O Algoritmo de uma passagem

Análise das referências posteriores

- Se o símbolo encontrado já estiver na TS mas ainda não estiver definido, então ele é inserido na lista especifica do símbolo e "ligado" com os itens que já se encontram na lista.
- Quando, finalmente, o símbolo aparece como rótulo, o seu valor (que é dado pelo valor do location counter) é usado para preencher os campos especificados na lista.
- Por último, esse valor é colocado na TS, com a indicação de "símbolo definido". Daqui para a frente, novas ocorrências do símbolo serão substituídas diretamente pelo valor que está na TS

O Algoritmo de uma passagem

Tabela de Símbolos			
Símbolo	Valor	Def	Lista
N	22	T	&
FAT	4	T	-1
ONE	23	T	&
FIM	18	T	&
AUX	21	T	&

Código fonte:			Contador
	INPUT	N	0
	LOAD	N	2
FAT:	SUB	ONE	4
	JNPZ	FIM	6
	STORE	AUX	8
	MULT	N	10
	STORE	N	12
	LOAD	AUX	14
	JNP	FAT	16
FIM:	OUTPUT	N	18
	STOP		20
AUX:	SPACE		21
N:	SPACE		22
ONE:	CONST	1	23

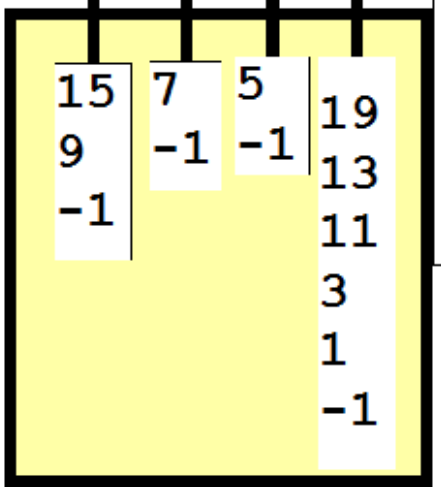


Figura: montador de passagem única

O Algoritmo de uma passagem

Tabela de Símbolos			
Símbolo	Valor	Def	Lista
N	22	T	&
FAT	4	T	-1
ONE	23	T	&
FIM	18	T	&
AUX	21	T	&

Código fonte:			Contador
	INPUT	N	0
	LOAD	N	2
FAT:	SUB	ONE	4
	JNPZ	FIM	6
	STORE	AUX	8
	MULT	N	10
	STORE	N	12
	LOAD	AUX	14
	JNP	FAT	16
FIM:	OUTPUT	N	18
	STOP		20
AUX:	SPACE		21
N:	SPACE		22
ONE:	CONST	1	23

15
9
-1

7
-1

5
-1

19
13
11
3
1
-1

Código Gerado
Em memória

End	0	12	22
End	2	10	22
End	4	02	-
End	6	08	-
End	8	11	-
End	10	03	22
End	12	11	22
End	14	10	22
End	16	05	4
End	18	13	-
End	20	14	22
End	21	XX	
End	22	XX	
End	23	01	

Figura: montador de passagem única

O Algoritmo de uma passagem

Exemplo

- Utilize o algoritmo de uma passagem para gerar a tabela de símbolos para o código objeto do programa mostrado abaixo (calcula area triângulo)

```
INPUT     B
INPUT     H
LOAD      B
MULT      H
DIV       DOIS
STORE     R
OUTPUT    R
STOP
B:        SPACE
H:        SPACE
R:        SPACE
DOIS:     CONST   2
```

O Algoritmo de uma passagem

Solução

```
INPUT    B       0
INPUT    H       2
LOAD     B       4
MULT     H       6
DIV      DOIS    8
STORE    R       10
OUTPUT   R       12
STOP             14
B:       SPACE   15
H:       SPACE   16
R:       SPACE   17
DOIS:    CONST 2 18
```

Tabela de Símbolos

Símbolo	Valor	Def	Lista
B	15	T	→5→1
H	16	T	→7→3
R	17	T	→13→11
DOIS	18	T	→9

O Algoritmo de uma passagem

Observação

- O uso de uma lista encadeada para resolver o problema de referências posteriores pode ser feita de forma mais "econômica" em termos de memória.
- Podemos armazenar a lista dentro do próprio código a ser gerado

O Algoritmo de uma passagem

Exemplo

- Vamos utilizar o mesmo código para gerar a lista no exemplo abaixo.

Código fonte:			Contador
	INPUT	N	0
	LOAD	N	2
FAT:	SUB	ONE	4
	JNPZ	FIM	6
	STORE	AUX	8
	MULT	N	10
	STORE	N	12
	LOAD	AUX	14
	JNP	FAT	16
FIM:	OUTPUT	N	18
	STOP		20
AUX:	SPACE		21
N:	SPACE		22
ONE:	CONST	1	23

```
End  0    12  -
End  2    10  -
End  4    02  -
End  6    08  -
End  8    11  -
End 10    03  -
End 12    11  -
End 14    10  -
End 16    05  4
End 18    13  -
End 20    14
End 21    XX
End 22    XX
End 23    01
```

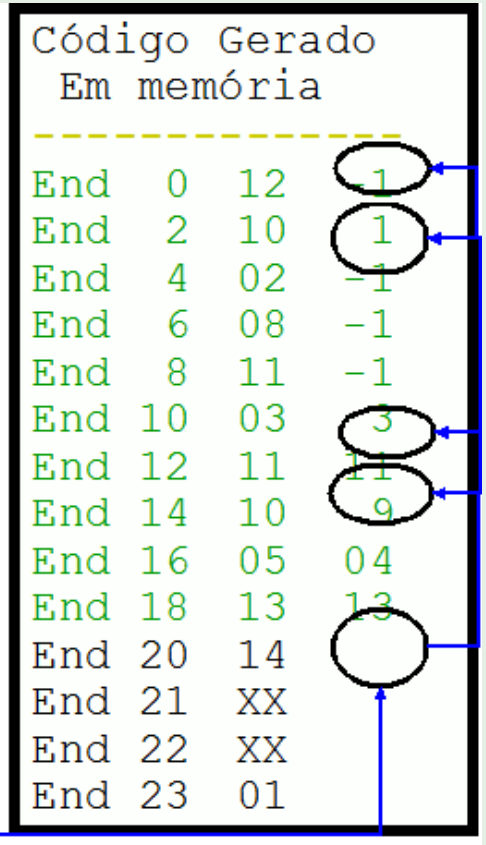
O Algoritmo de uma passagem

Exemplo

Código fonte:			Contador
	INPUT	N	0
	LOAD	N	2
FAT:	SUB	ONE	4
	JNPZ	FIM	6
	STORE	AUX	8
	MULT	N	10
	STORE	N	12
	LOAD	AUX	14
	JNP	FAT	16
FIM:	OUTPUT	N	18
	STOP		20
AUX:	SPACE		21
N:	SPACE		22
ONE:	CONST	1	23

Tabela de Símbolos

Símbolo	Valor	Def	Lista
N	22	T	19
FAT	4	T	-1
ONE	23	T	5
FIM	18	T	7
AUX	21	T	15

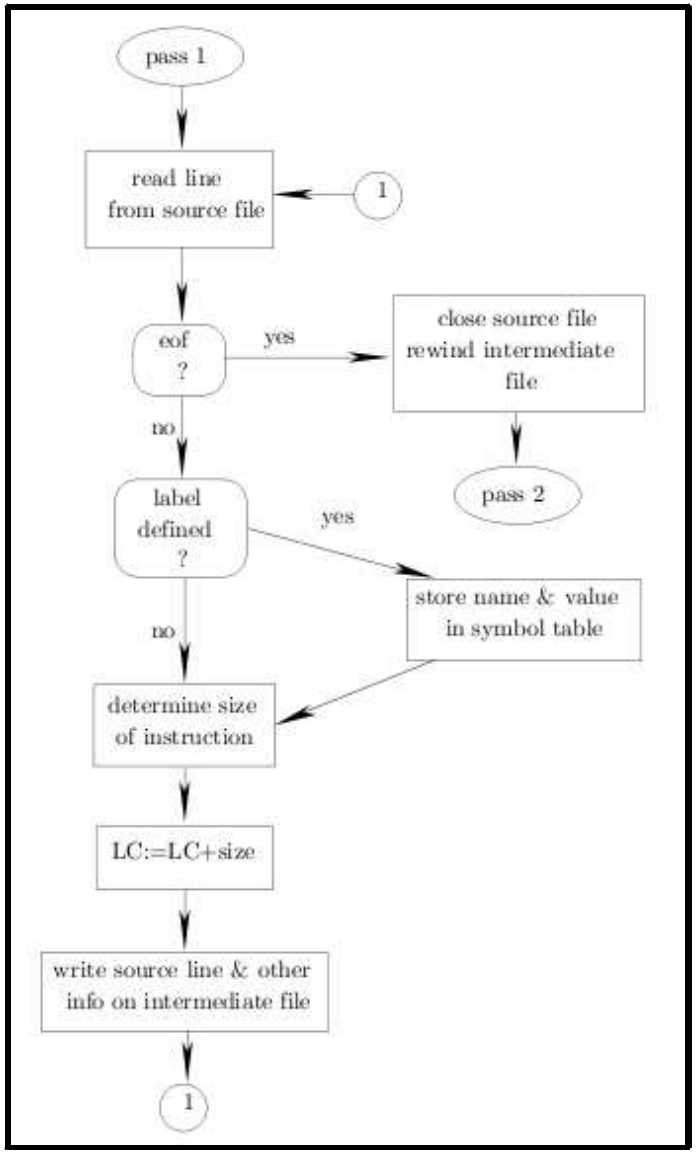


O Algoritmo de uma passagem

Vantagens/Desvantagens

- O algoritmo de duas passagems é mais simples de implementar e requer menos memória do algoritmo de passagem única.
- O algoritmo de passagem única economiza uma leitura completa do arquivo fonte.

Fluxogramas – Montador de 2 passagens



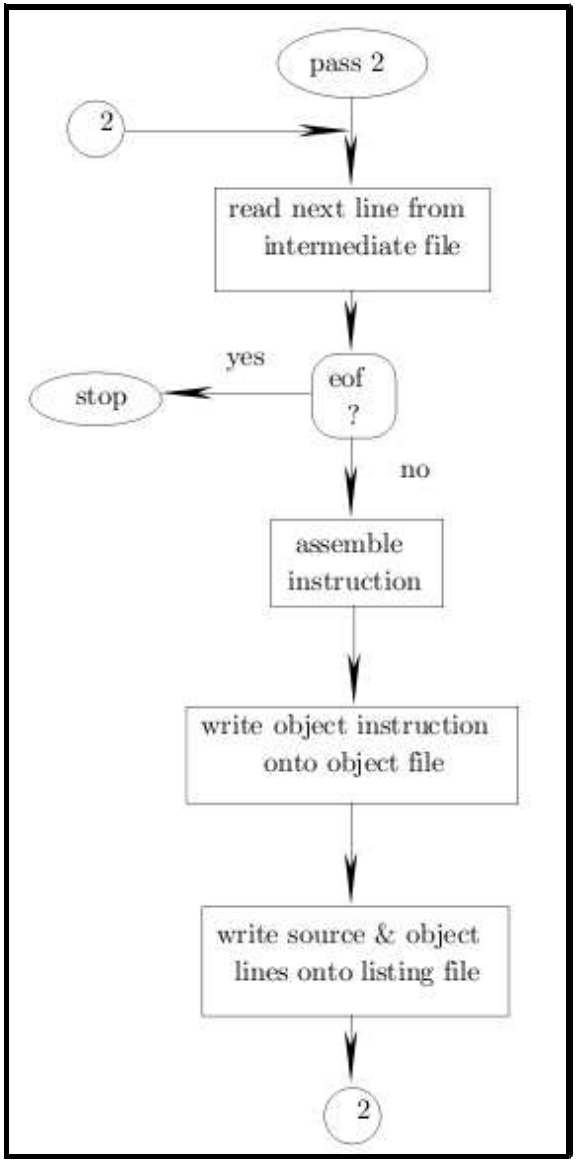


Figura: Fluxograma montador de duas passagens

Fluxogramas – Montador de passagem única

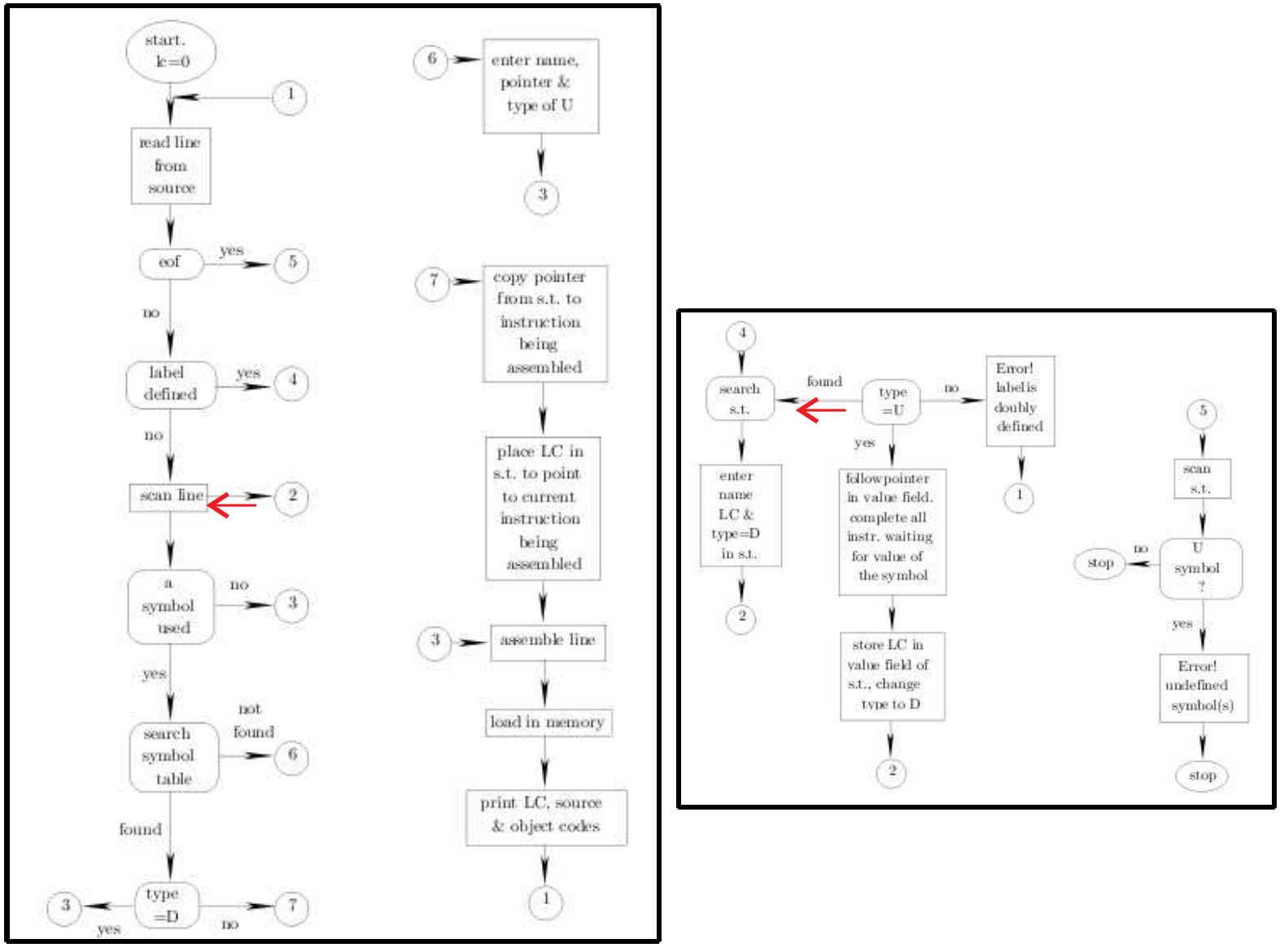


Figura: Fluxograma montador de duas passagens

Próxima Aula

Montador avançado

Introdução à Software Básico: Montadores - parte 1

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

Sumário

Montadores

Características adicionais dos montadores

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.

Características adicionais dos montadores

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
...
```

Características adicionais dos montadores

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
  IF expressão
- A diretiva *IF* instruirá o montador a incluir a linha seguinte somente se a expressão durante o tempo de montagem seja diferente de zero.
  FLAG EQU 1
  . . .
  IF FLAG
  JMP XXX
- O uso de *IF* aqui é semelhante ao *#ifdef* que usamos em C

Características adicionais dos montadores - MACROS

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.

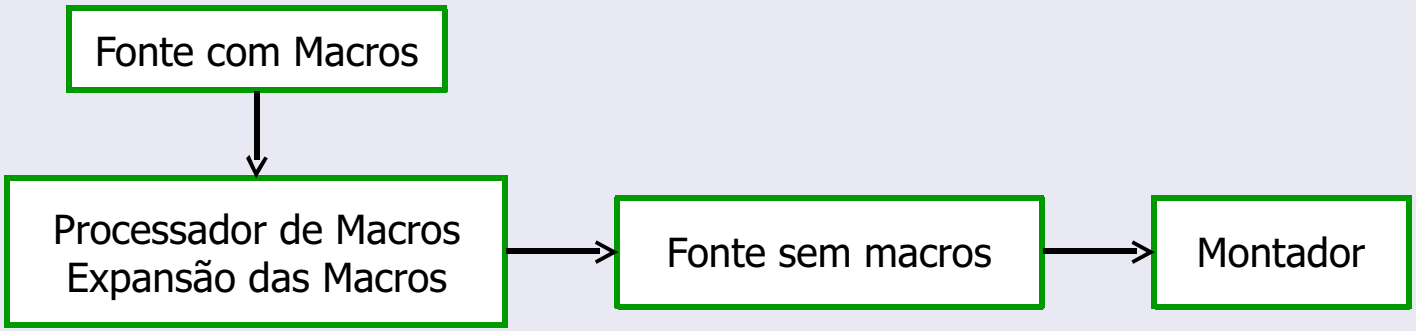
Salomon, Assemblers and Loaders, Elsevier Science, 1993, http://www.davidsalomon.name/assem.advertis/AssemAd.html, cap.4

Características adicionais dos montadores - MACROS

- 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.

Características adicionais dos montadores - MACROS

- 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

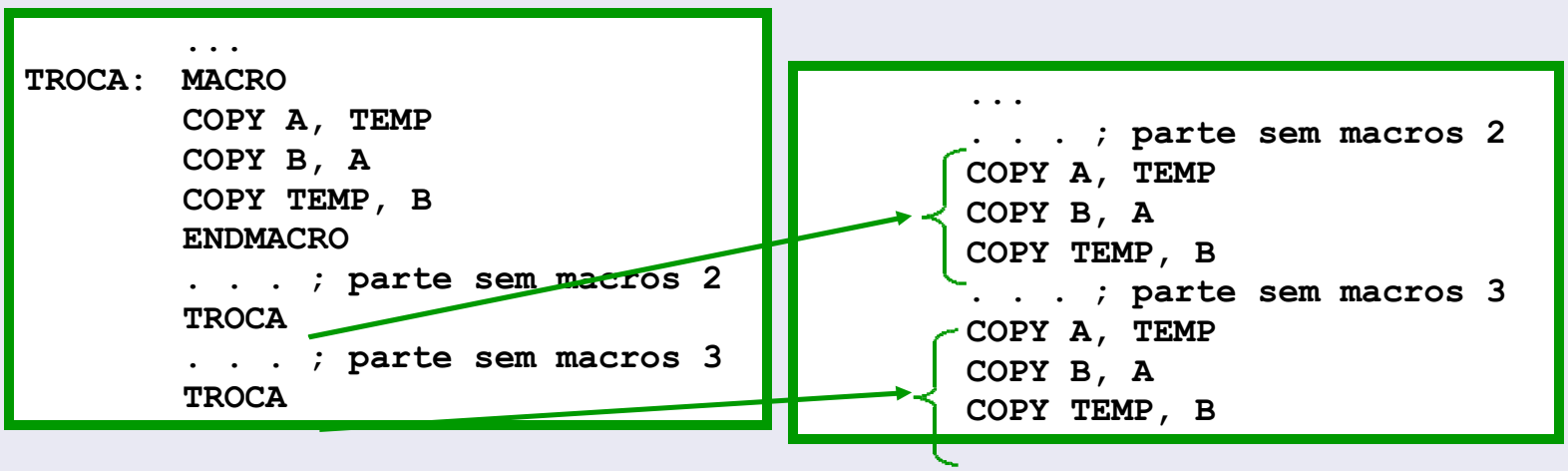
Algoritmo 1 Exemplo 1

```
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*.

Características adicionais dos montadores - MACROS

- 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.



Características adicionais dos montadores - MACROS

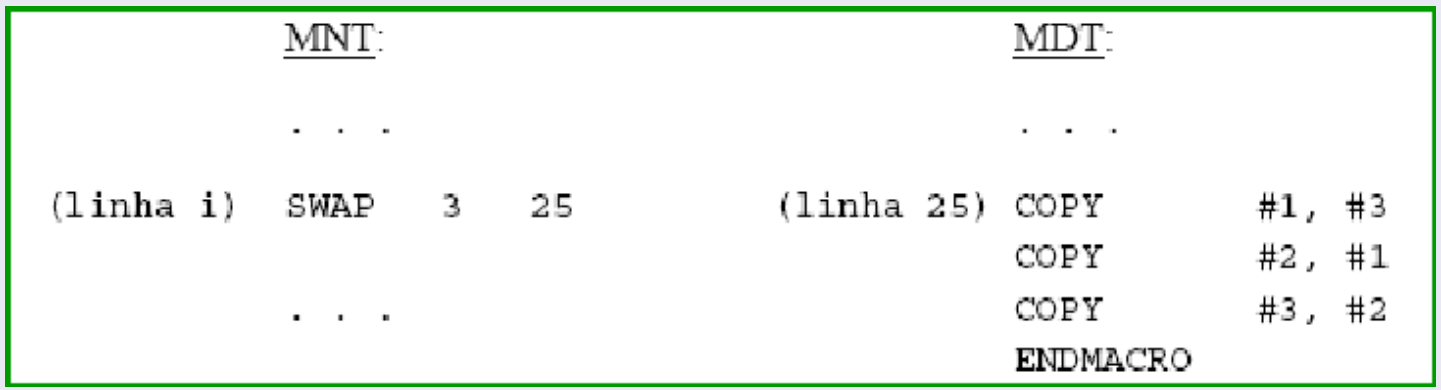
- Tipicamente, os processadores de macros exigem que uma macro seja definida antes de ser chamada.
- Essa restrição permite que o processador de macros realize seu trabalho com apenas uma passagem sobre o programa fonte.
- A macro TROCA, como definida antes, sempre troca os valores das posições A e B, 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 $\#i$.



Passagem 0 do Montador - Macros

1. Ler próxima linha do código fonte
2. Se é MACRO, ler a definição completa da Macro e armazena na MDT. Ir para 1.
3. Se é uma outra diretiva de pré-processamento, executar e ir para 1.
4. 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.
5. Em qualquer outro caso, escreve a linha no novo código fonte. Ir para 1.
6. Se eof, fim da passagem 0.

Fluxograma - Passagem 0 do Montador

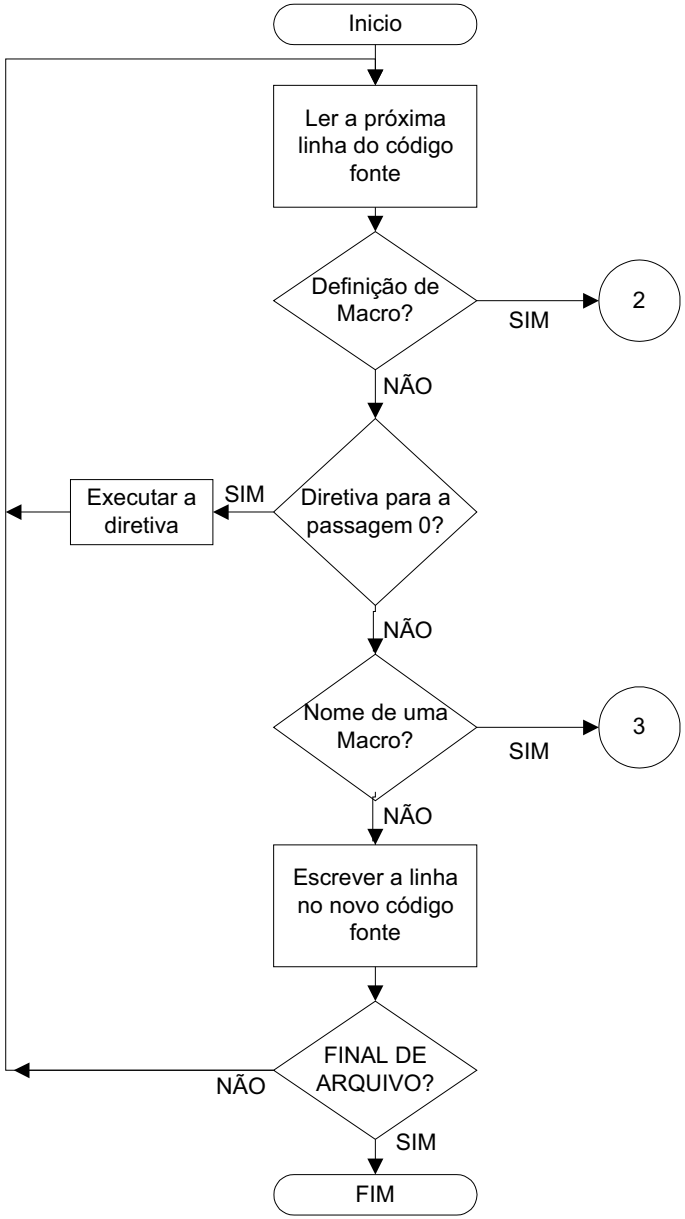


Figura: Passagem 0

Fluxograma - Passagem 0 do Montador

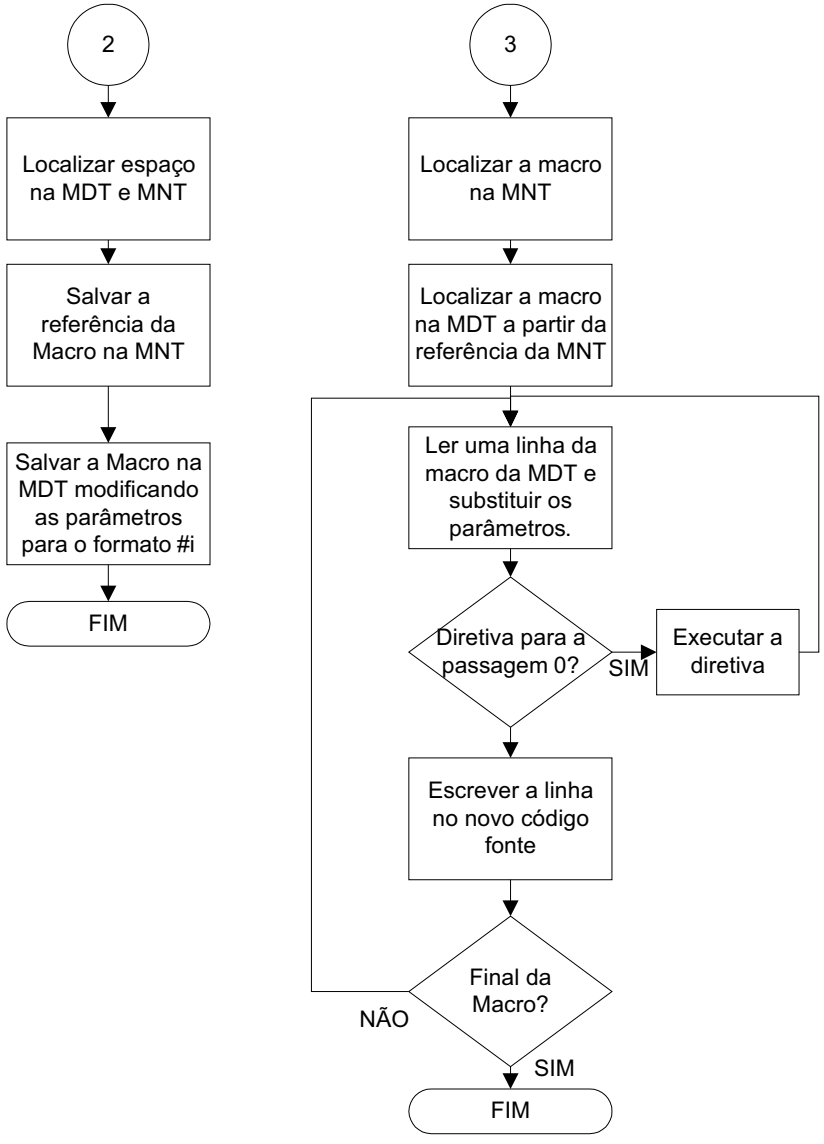


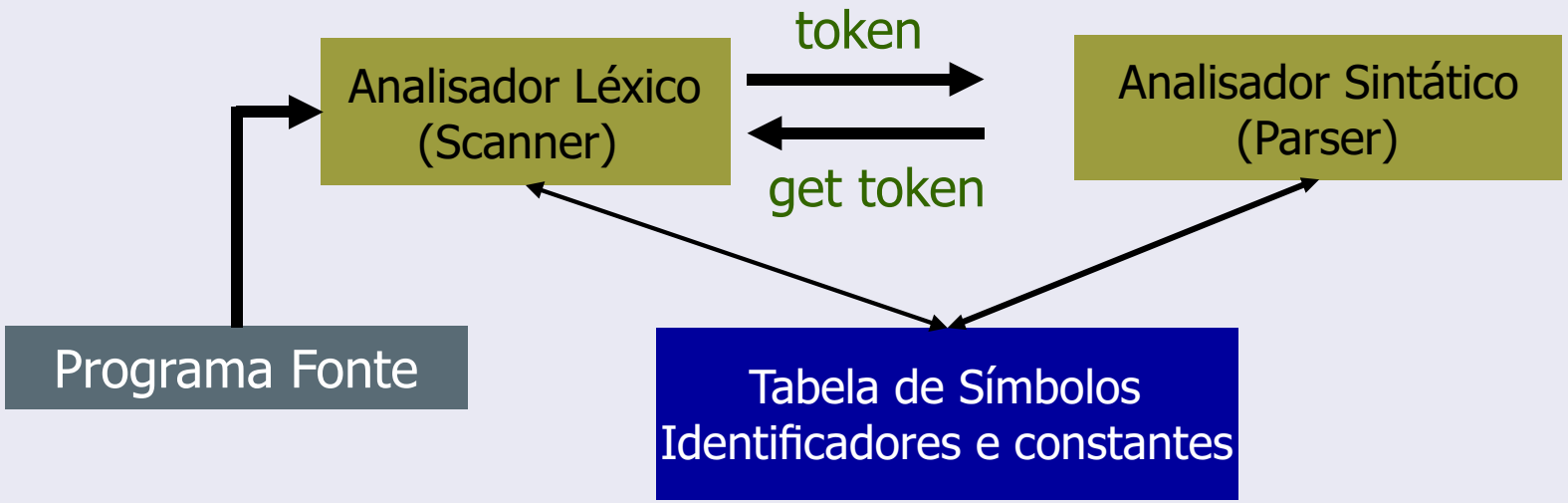
Figura: Passagem 0

Construção de um Montador

- Como já vimos, os tradutores (incluídos ai, 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.

Construção de um Montador

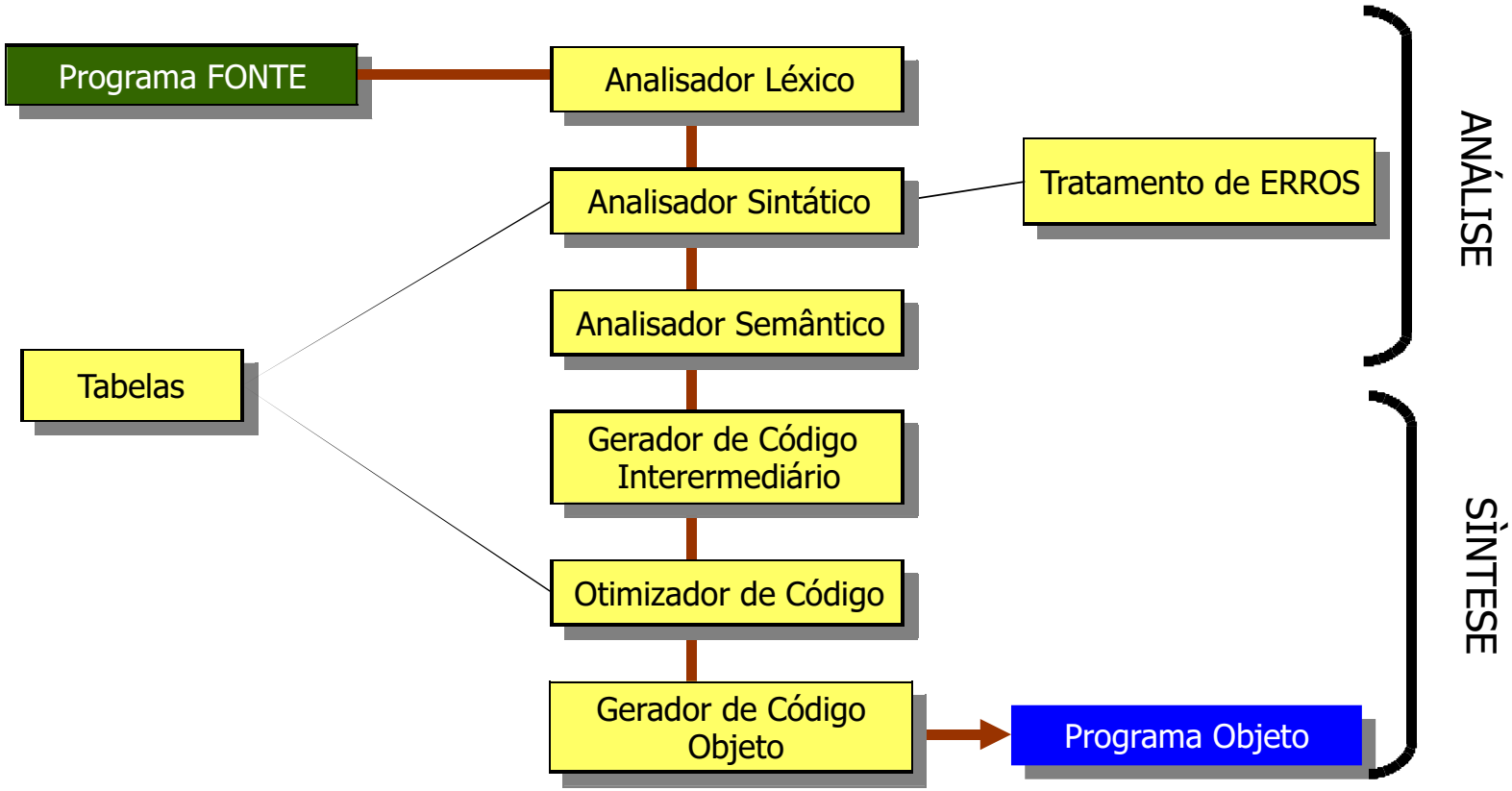
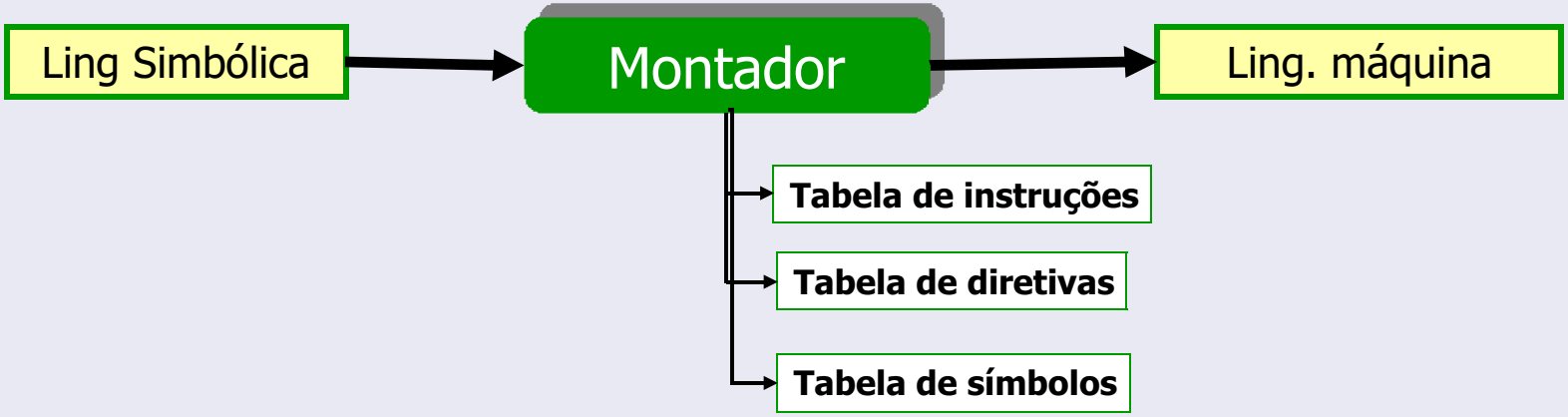


Figura: Estrutura de um Tradutor

Construção de um Montador

- 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

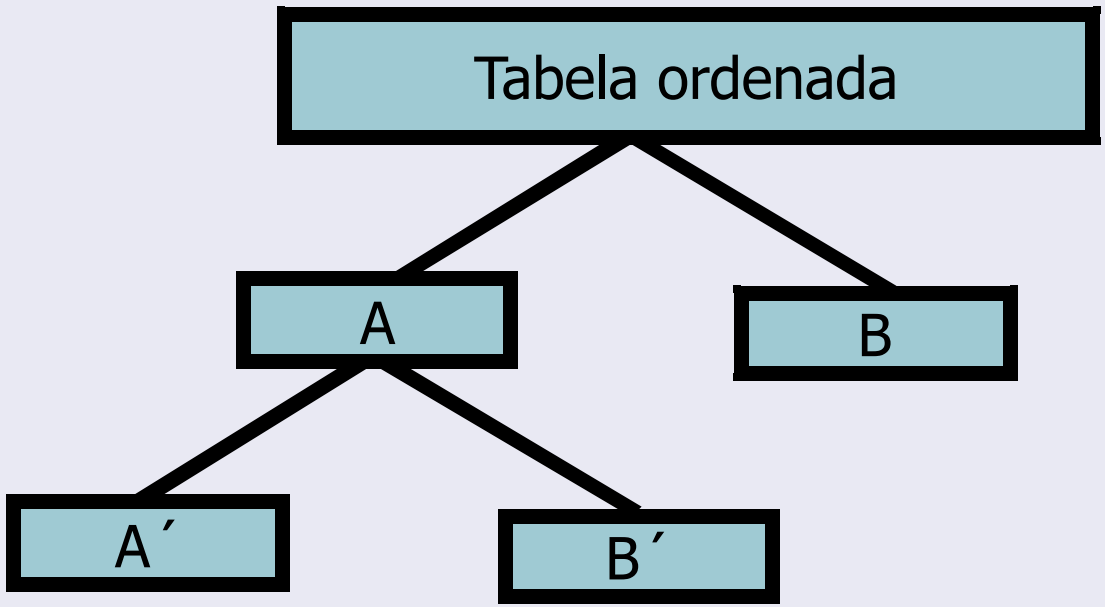
- O processo de busca normalmente recebe como entrada código (nome), e tenta encontrar na tabela,
- Retorna os valores associados ao código de busca quando encontrado ou informa que o mesmo não foi encontrado.

1 - Busca Sequêncial

- Consiste em ler todos os ítems 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

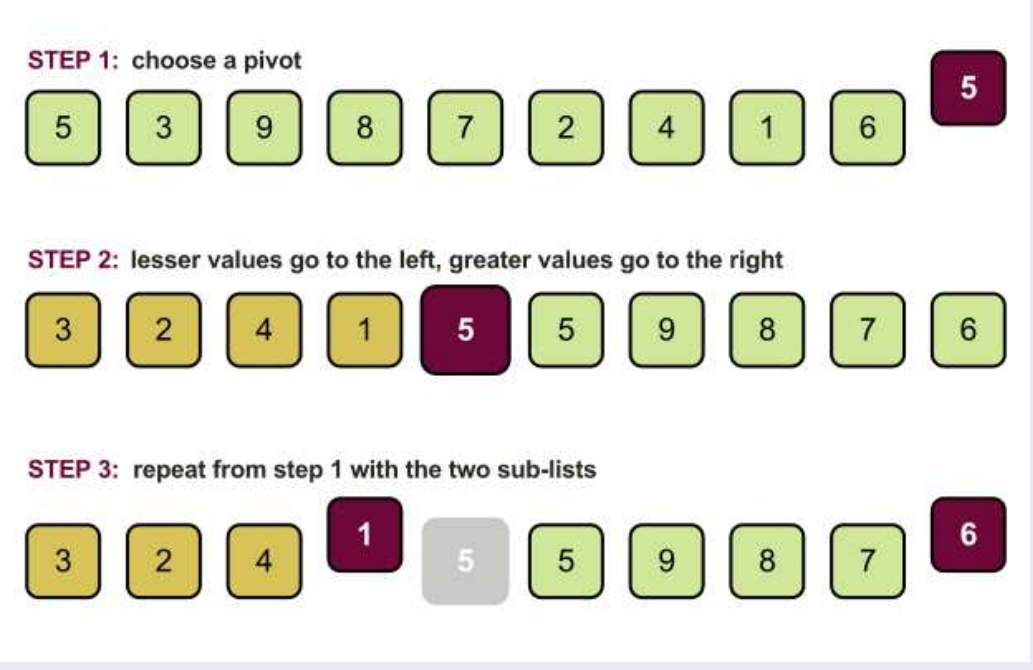
for $k \leftarrow 0$ to $N - 2$ **do**
  $posMenor \leftarrow k$
  for $i \leftarrow k + 1$ to $N - 1$ **do** {Percorre todo o vetor}
    if $numero[i] < numero[posMenor]$ **then**
      $posMenor \leftarrow i$
    end if
  end for
  if $posMenor \neq k$ **then**
    $aux \leftarrow numero[posMenor]$
    $numero[posMenor] \leftarrow numero[k]$
    $numero[k] \leftarrow aux$
  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 $O(n^2)$.

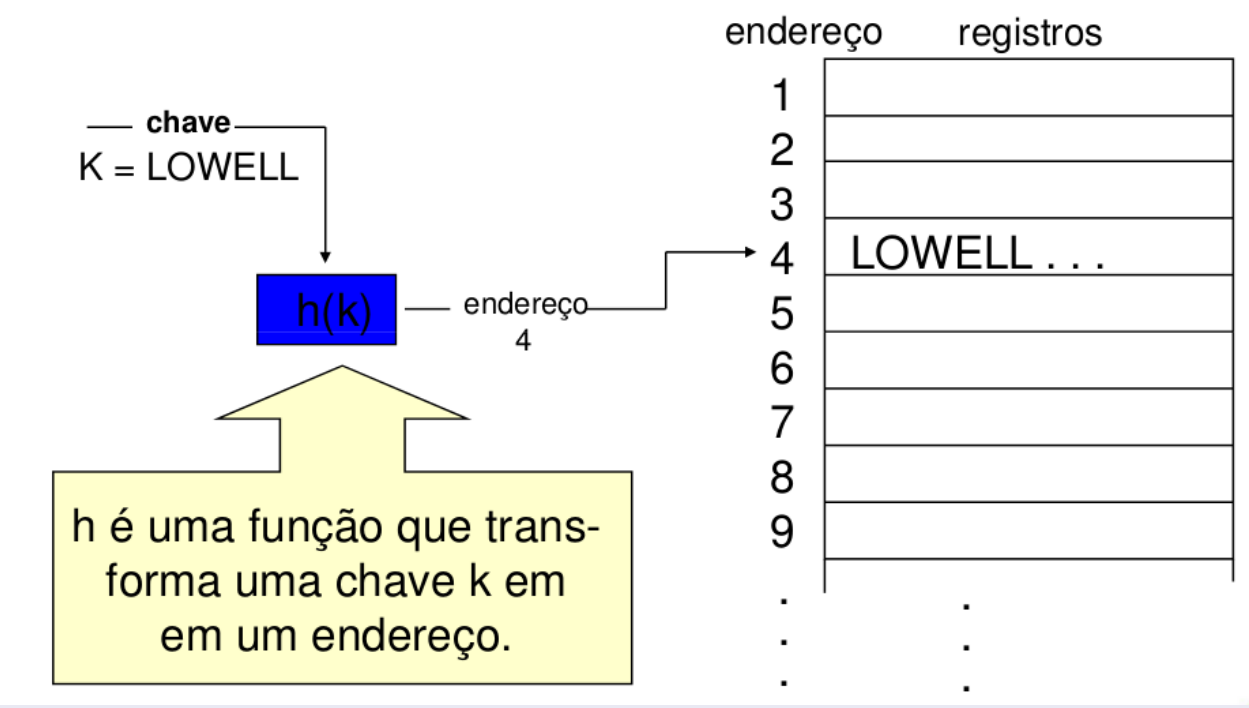
Quicksort

- 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: $O(n \log_2 n)$



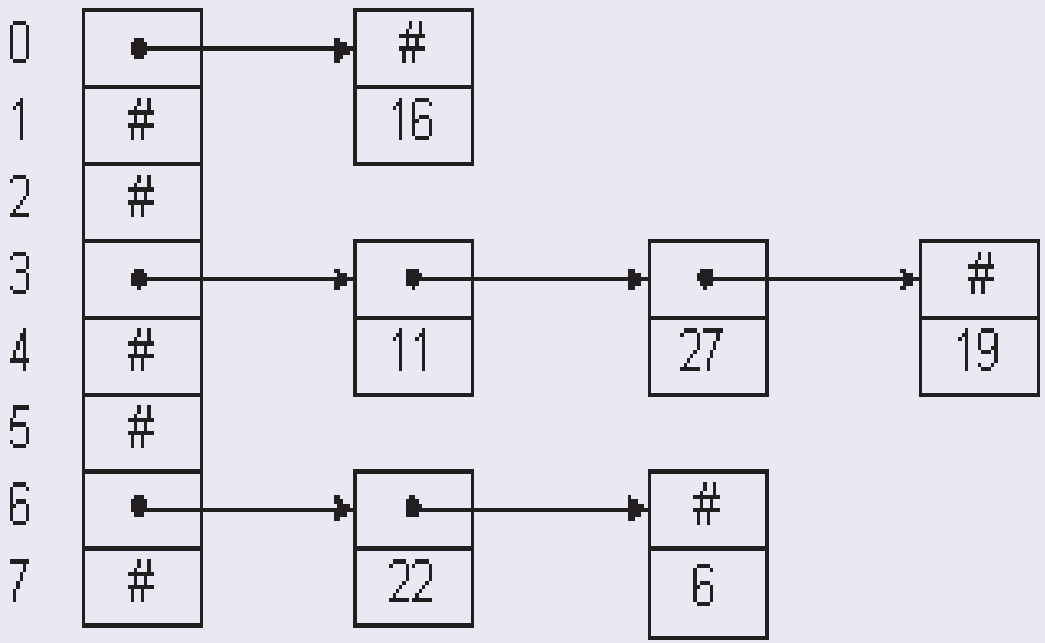
Hashing

- Porém, seria melhor se nossa tabela estivese 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 dificil achar uma função hash que gere um hashing sem colições.
  - 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

```
            STOP
VETOR:      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