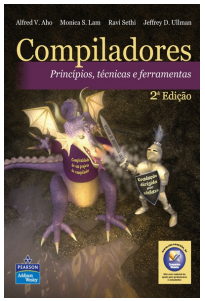
CTC-41

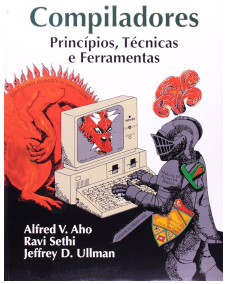
Compiladores

Carlos Alberto Alonso Sanches

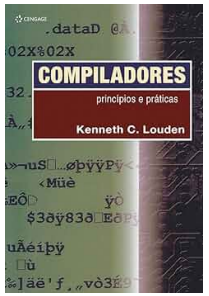
Bibliografia

- A.V. Aho, M.S. Lam, R. Sethi e J.D. Ullman
  Compiladores: princípios, técnicas e ferramentas





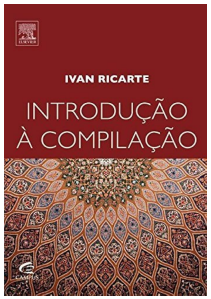
- K.C. Louden
  Compiladores: princípios e práticas

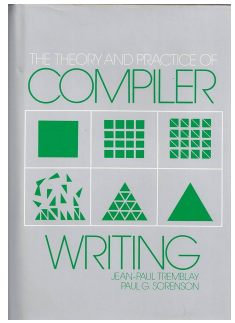


- P.R. Santos e T. Langlois
  Compiladores da teoria à prática

Bibliografia complementar

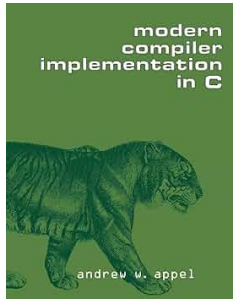
- I. Ricarte
  Introdução à compilação





- J.P. Tremblay and P.G. Sorenson
  The theory and practice of compiler writing

- A.W. Appel
  Modern compiler implementation in C



- T.Æ. Mogensen
  Basics of compiler design
  http://www.diku.dk/~torbenm/Basics

Organização do curso

- Primeiro bimestre
  - Introdução (Cap. 1)
  - Análise léxica (Cap. 2)
  - Análise sintática (Cap. 3 e 4)

Projeto 1: analisador léxico (20%)

Prova 1: todo o bimestre (80%)

- Segundo bimestre
  - Análise sintática (Cap. 5)
  - Análise semântica (Cap. 6)
  - Geração de código (Cap. 7)

Projeto 2: analisador sintático (30%)

Projeto 3: analisador semântico (20%)

Prova 2: todo o bimestre (50%)

- Exame

Projeto final: compilador completo (100%)

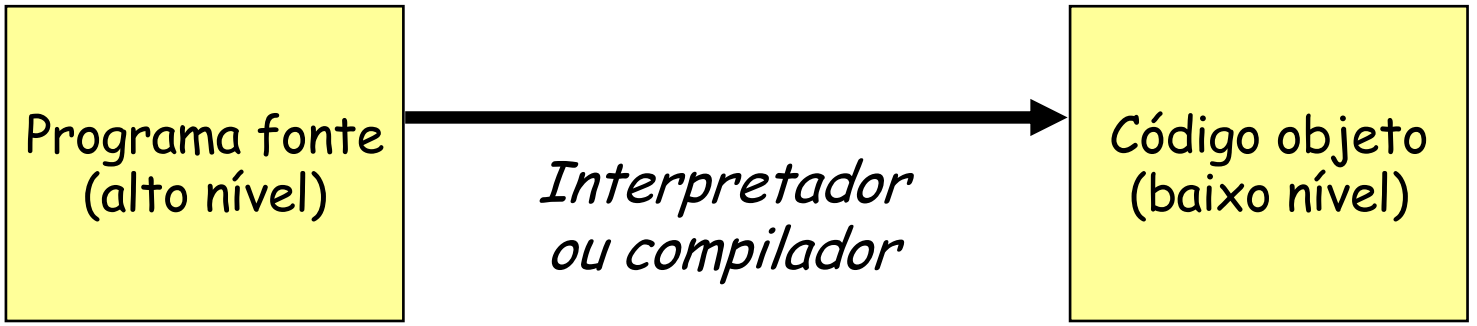
CTC-41

1) Introdução

Processo de compilação

Linguagens

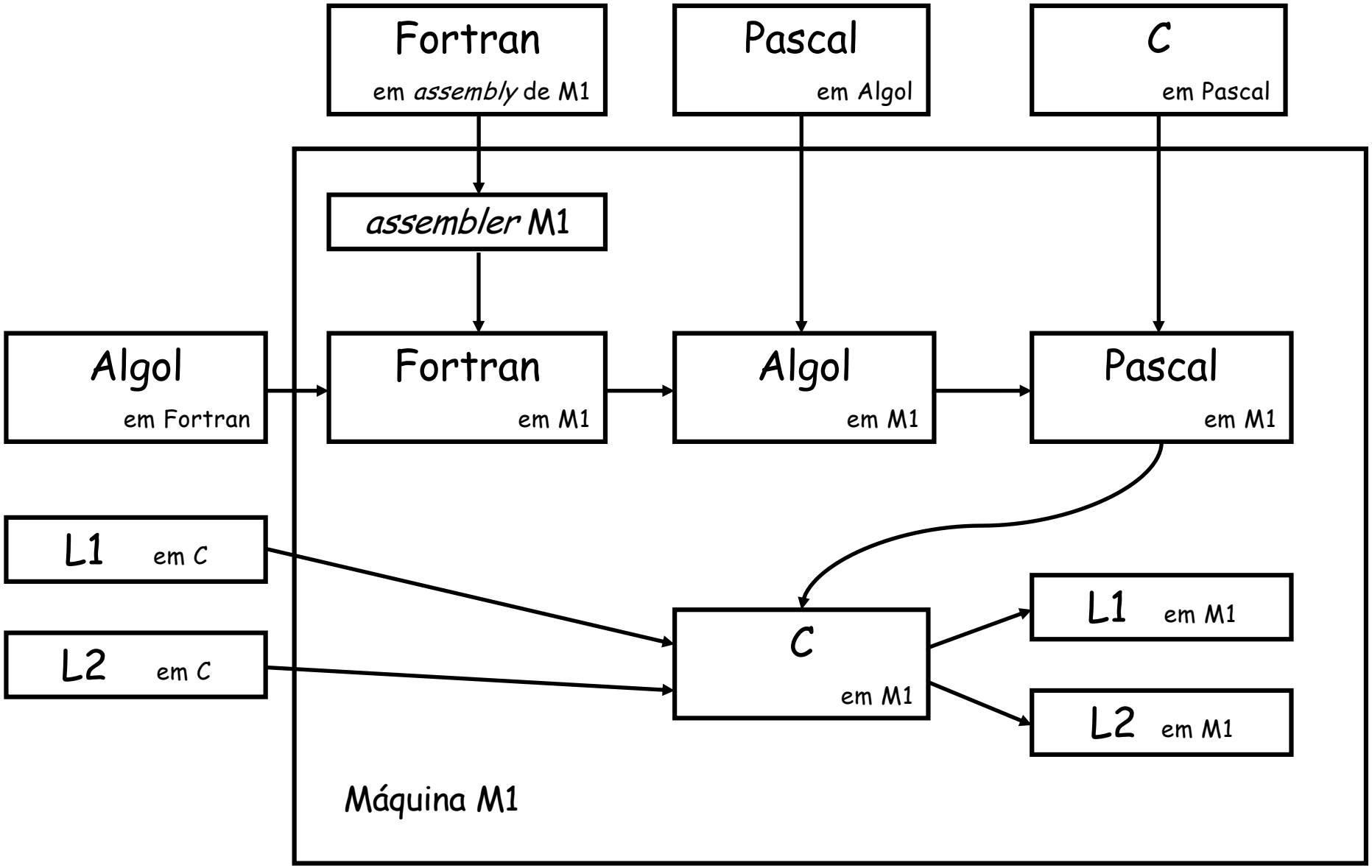
- Os seres humanos se comunicam através de linguagens naturais.
  - Há muitas variações (idiomas) e formatos (alfabetos, gestos, notações matemáticas, partituras musicais, etc.)
- No entanto, a transmissão de comandos aos computadores precisa ser em níveis muito elementares: bits, uso de memória e registradores, instruções de máquina, etc.
- Programar deste modo é extremamente cansativo, tornando praticamente inviável a elaboração de grandes sistemas.
- A melhor solução encontrada para a programação de computadores foi o desenvolvimento de linguagens de alto nível:



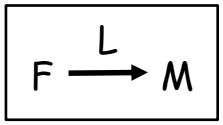
Interpretador *versus* compilador

- Interpretador
  - Programa fonte é executado pelo interpretador, sem geração de código objeto
  - Essa execução ocorre necessariamente no computador hospedeiro
  - Erros no programa fonte geralmente levam a um término abrupto
  - Exemplos: Basic, SmallTalk, HTML, LISP, Python, MatLab, R
- Compilador
  - É mais elaborado e se recupera após detecção de erros no programa fonte
  - Gera código objeto executável específico para cada máquina
  - Este código geralmente é otimizado, e será executado por *hardware*
  - Compilação é mais lenta que interpretação, mas produz códigos mais rápidos
  - Um compilador pode se autocompilar para rodar em outra máquina
  - Exemplos: C, Pascal, Fortran, Cobol
- Sistema híbrido
  - Produz um único código intermediário para ser executado em várias máquinas
  - Exemplo: a compilação de programas Java gera *bytecodes*, que posteriormente são traduzidos pelas JVM, produzindo um código objeto específico

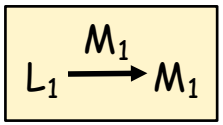
Primeiros compiladores

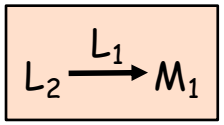


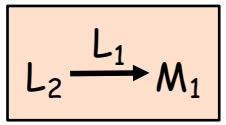
Método de *bootstrapping*

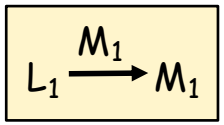


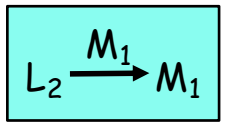
Linguagem fonte F, compilador escrito em L, código para máquina M

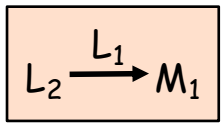
Disponível inicialmente 

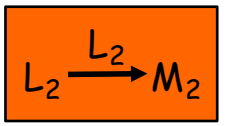
Desenvolve-se 

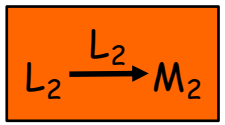


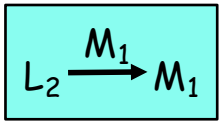
é compilado por 

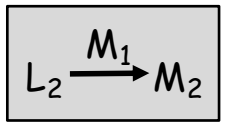
gerando 

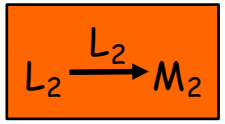
Adapta-se 

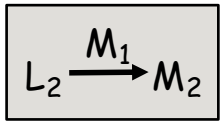
para se obter 

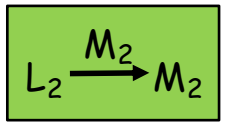


é compilado por 

gerando 



é compilado por 

gerando 

Aspectos de uma linguagem

- Cada linguagem, seja natural ou de programação, possui três aspectos fundamentais:
  - Léxico ou ortográfico: corresponde ao conjunto de termos permitidos. Na linguagem natural, são as palavras de um dicionário. No processo de compilação, são chamados de átomos ou *tokens.*
  - Sintático ou gramatical: corresponde às regras de formação das expressões válidas. Exemplos na linguagem natural: compatibilidades entre tempos verbais e sujeito (número e gênero), uso correto de conjunções, pontuações, etc.
  - Semântico: corresponde ao significada de cada expressão, que precisa fazer sentido. Exemplos de erros semânticos em frases da linguagem natural:
    - Eu chovo toda quarta-feira à tarde.
    - Um ônibus comprou laranja no açougue.

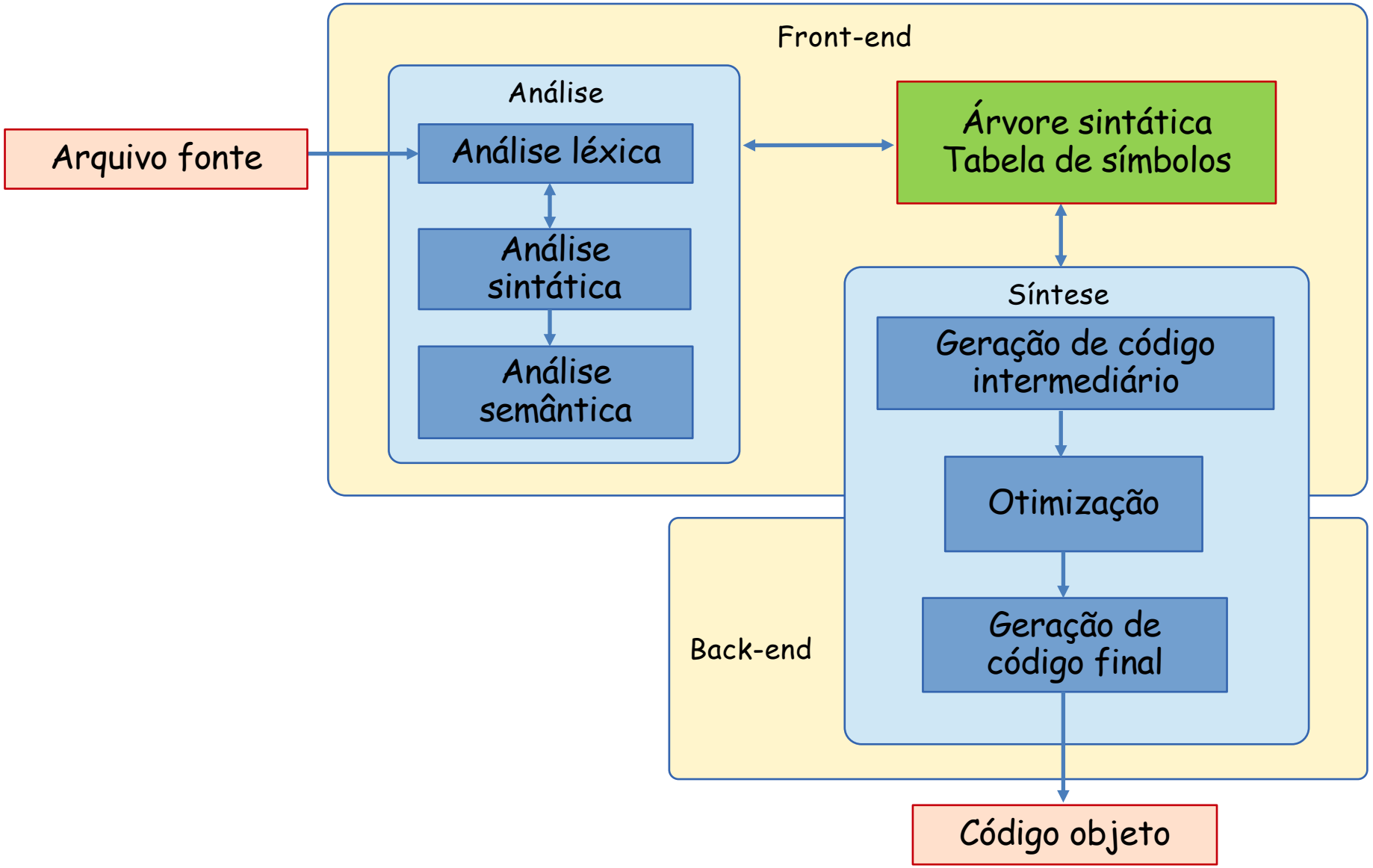
Tratamento de erros

- Exemplos de erros na linguagem C:
  - Léxicos
    - `Int x;`
    - `char 1a;`
  - Sintáticos
    - `a = 6`
    - `X == 5;`
  - Semânticos
    - `int x = 10.7;`
    - `int f1(int a, float b) { return a%b; }`
    - ```
      void f2(int j, int k)
      { if (j == k) break; else continue; }
      ```
- Diante de erros como esses, o compilador deve ser capaz de emitir alguma mensagem, e em seguida retomar a compilação.
      ```

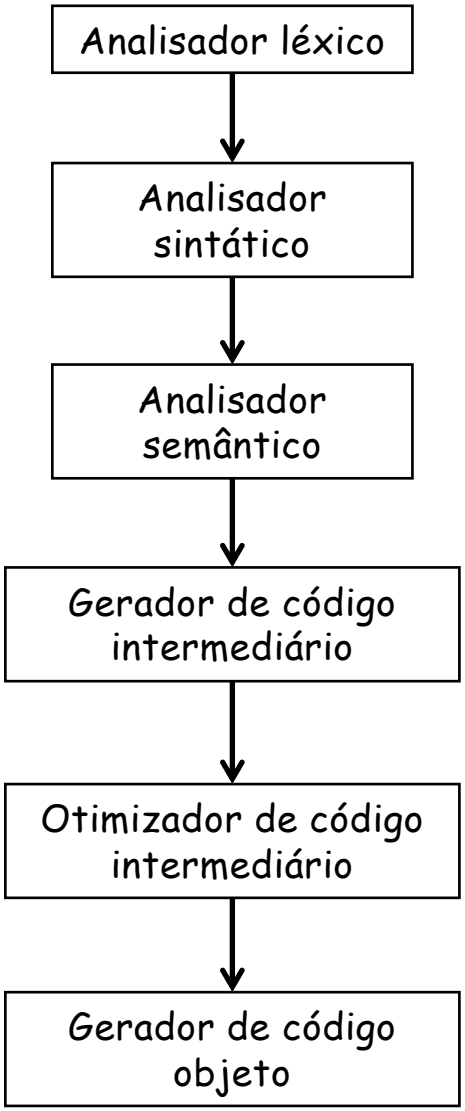
Processo de compilação

- A compilação de um programa possui duas grande partes:
  - Análise: é formada por 3 fases (léxica, sintática e semântica)
    - Durante a análise, o programa fonte é dividido em elementos constituintes, construindo-se uma estrutura hierárquica intermediária para a sua representação (chamada de *árvore sintática*).
    - Uma função essencial da análise é registrar os identificadores usados e coletar informações sobre seus atributos (nome, tipo, argumentos, escopo, etc.), criando a chamada *tabela de símbolos*.
  - Síntese: percorre-se a árvore sintática e, com o uso da tabela de símbolos, gera-se o código objeto desejado
    - Primeiramente, é produzido um código intermediário, que depois será otimizado e transformado no código objeto.
    - Na fase final, leva-se em conta as propriedades da máquina alvo (instruções, modos de endereçamento e de representação de dados, registradores, etc.)
    - Deste modo, um mesmo compilador pode gerar códigos para diversas máquinas.
- Costuma-se chamar de interface *front-end* (vanguarda) as fases que dependem apenas da linguagem fonte, e de interface *back-end* (retaguarda) as que dependem somente da máquina alvo.

Processo de compilação



Exemplo



Arquivo fonte

```
while (i < nn) i = i + j;
```

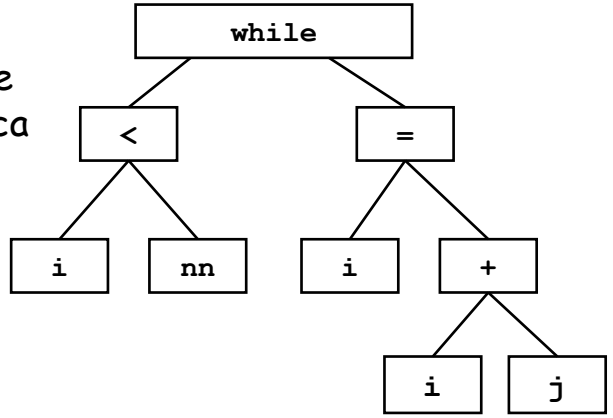
Sequência de *tokens*

while	(	i	<	nn	)
i	=	i	+	j	;

Tabela de símbolos

i	int	---
nn	int	---
j	int	---

Árvore sintática



Código intermediário

```
R1: T1 = i < nn
    JF T1 R2
    T2 = i + j
    i = T2
    JUMP R1
R2: - - - -
```

```
R1: T1 = i < nn
    JF T1 R2
    i = i + j
    JUMP R1
R2: - - - -
```

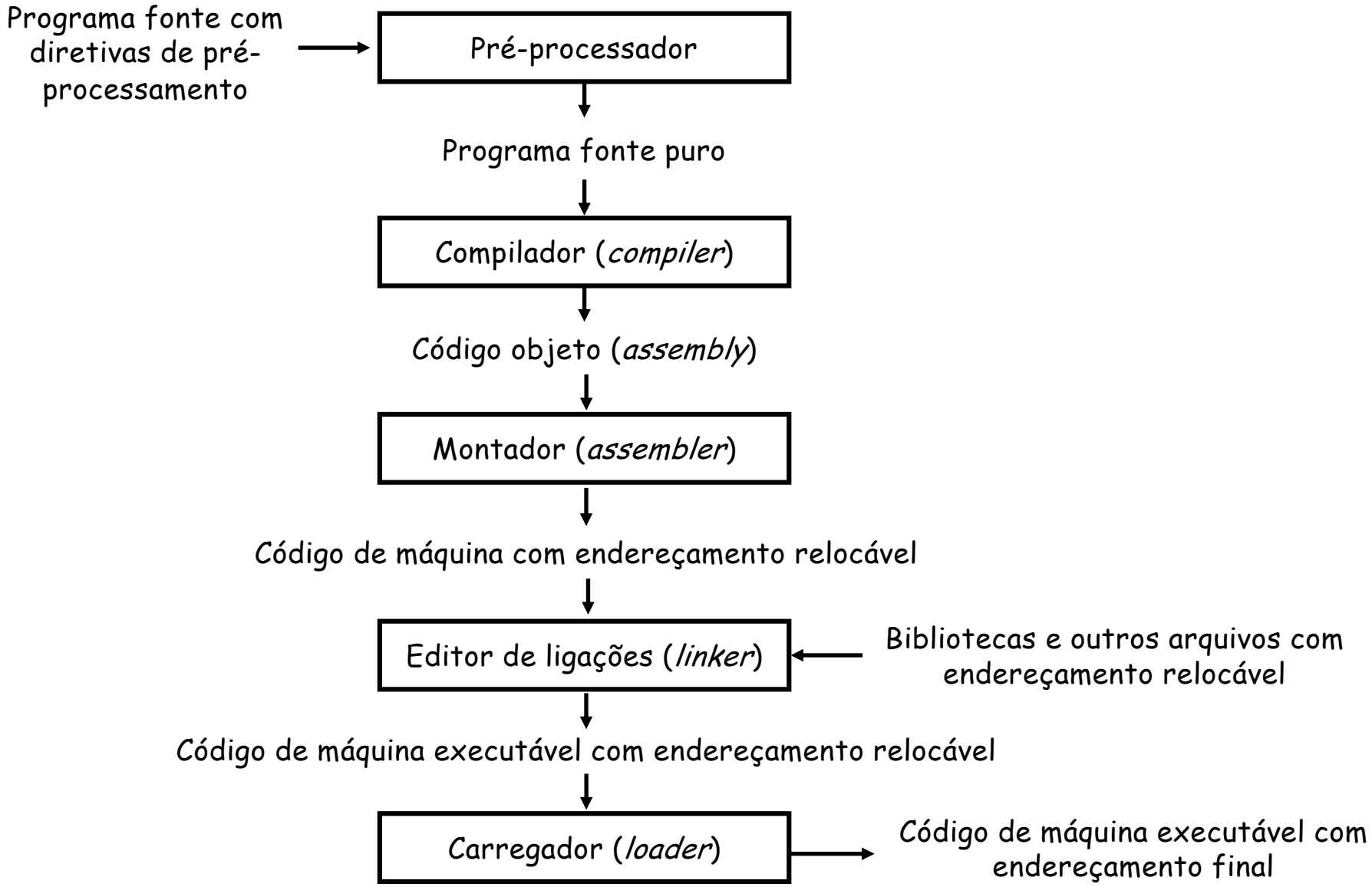
Código objeto

```
    load i
R1: sub nn
    JZ R2
    JP R2
    load i
    add j
    st i
    J R1
R2: - - - -
```

Pré e pós processamentos

- A entrada para o compilador pode ser produzida por um ou mais pré-processadores:
  - Pré-processamento de macros (exemplo: **`#define`**)
  - Inclusão de arquivos (exemplo: **`#include`**)
  - Compilação condicional, diretivas de pré-processamento
- A saída do compilador pode necessitar de pós-processamentos:
  - Montador (*assembler*) em uma passagem (a partir da versão binária) ou em duas passagens (a partir da versão mnemônica): gera código de máquina relocável
  - Editor de ligações (*linker*): cria um único arquivo a partir de diversos códigos relocáveis
  - Carregador (*loader*): altera e define os endereços relocáveis

Esquema geral



Exemplo de código *assembly*

```
#include <stdio.h>
void main ( )
{
    int n, fat, i;
    scanf ("%d", &n);
    fat = 1;
    i = 2;
    while (i <= n) {
        fat = fat * i;
        i = i + 1;
    }
    printf ("%d", fat);
}
```

`inic` é o rótulo da primeira instrução executável

```
C1:     CONST   1
C2:     CONST   2
n:      CONST   0
fat:    CONST   0
i:      CONST   0
inic:   READ    n
        LD      C1
        ST      fat
        LD      C2
        ST      i
loop:   SUB     n
        JP      escr
        LD      fat
        MULT    i
        ST      fat
        LD      i
        ADD     C1
        ST      i
        JUMP    loop
escr:   WRITE   fat
        STOP
        END     inic
```

Exemplo de código relocável

```
C1:     CONST   1
C2:     CONST   2
n:      CONST   0
fat:    CONST   0
i:      CONST   0
inic:   READ    n
        LD      C1
        ST      fat
        LD      C2
        ST      i
loop:   SUB     n
        JP      escr
        LD      fat
        MULT    i
        ST      fat
        LD      i
        ADD     C1
        ST      i
        JUMP    loop
escr:   WRITE   fat
        STOP
        END     inic
```

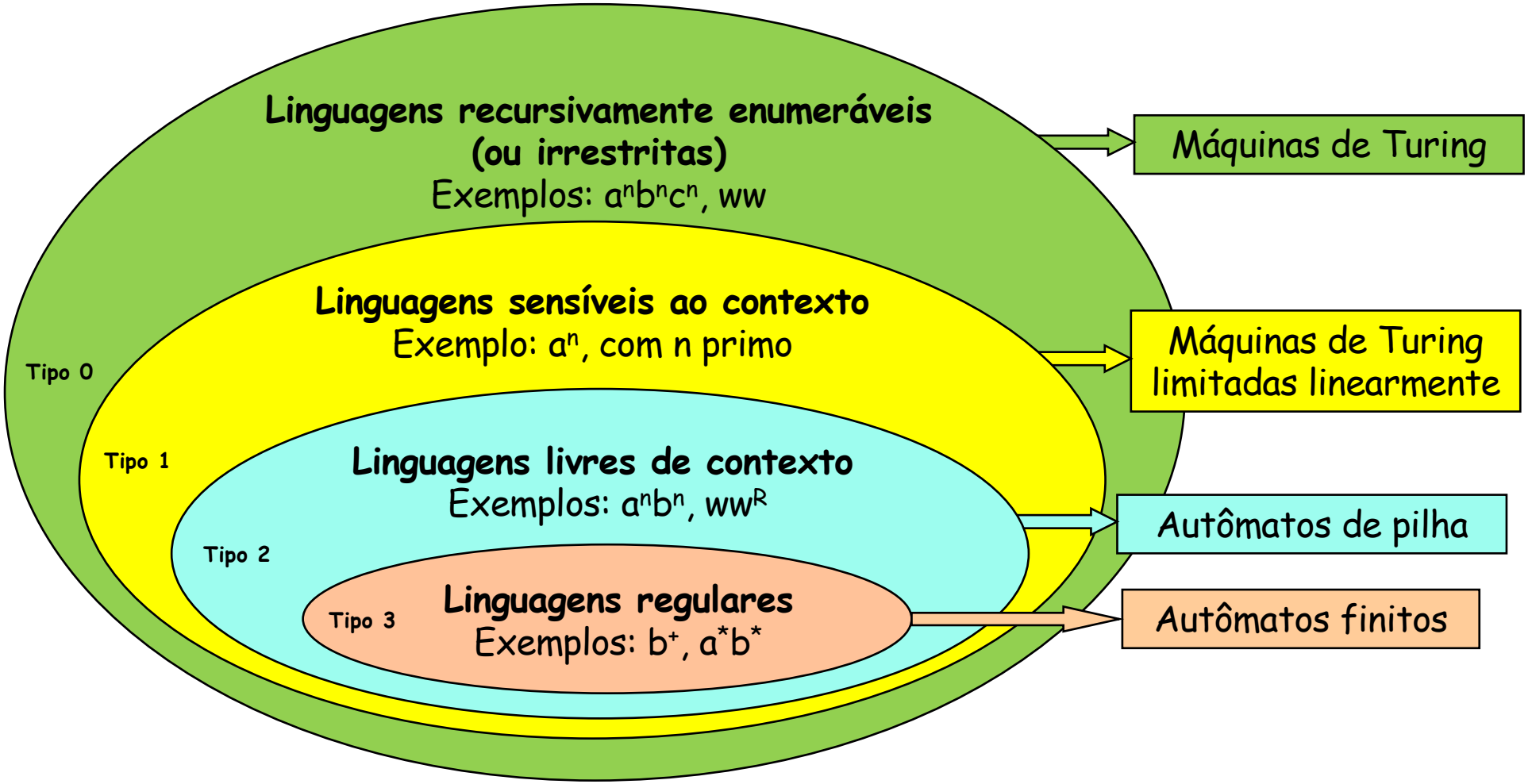
Mnemônico	Código
LD	1
ST	2
ADD	4
SUB	5
MULT	6
JUMP	11
JP	14
READ	15
WRITE	16
STOP	17

Rótulo	Endereço
C1	0
C2	1
n	2
fat	3
i	4
inic	5
loop	10
escrever	19

Memória	Código	Endereço
0	1	
1	2	
2	0	
3	0	
4	0	
5	15	2
6	1	0
7	2	3
8	1	1
9	2	4
10	5	2
11	14	19
12	1	3
13	6	4
14	2	3
15	1	4
16	4	0
17	2	4
18	11	10
19	16	3
20	17	0

Hierarquia de Chomsky

- Noam Chomsky estabeleceu em 1956 uma hierarquia entre as linguagens formais, com seus reconhecedores.

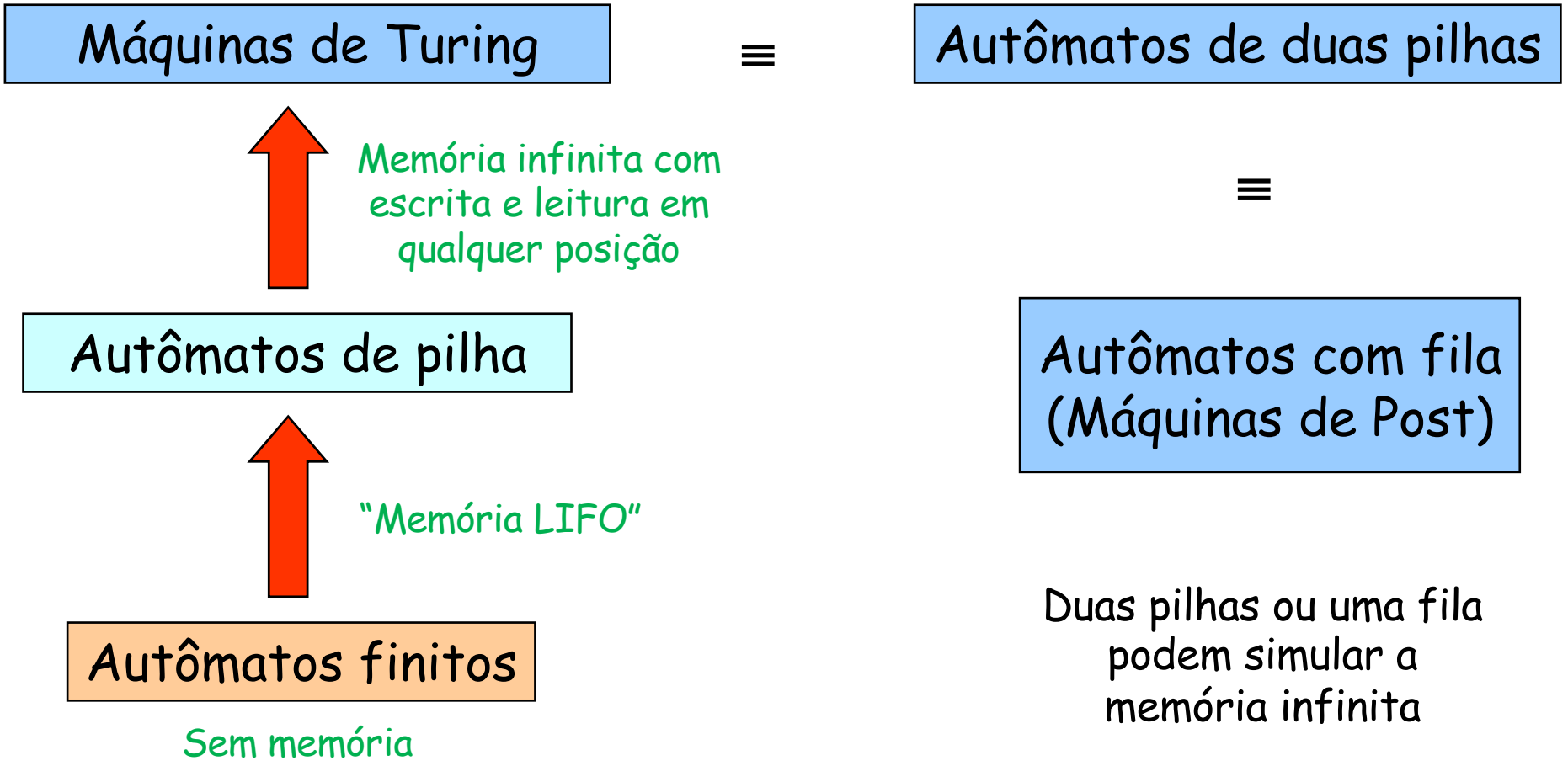


Gramáticas e reconhecedores

- As gramáticas são *formalismos axiomáticos*: através de regras de produção, definem como são *geradas* as palavras de uma determinada linguagem formal.
- Os reconhecedores são *formalismos operacionais*: são dispositivos computacionais que *reconhecem* se uma palavra pertence ou não a uma determinada linguagem formal.

Poder computacional

- Qual é o fator diferencial que realmente aumenta o poder computacional desses dispositivos reconhecedores?



Linguagens formais na compilação

- Geralmente, as linguagens de programação são especificadas através de gramáticas livres de contexto, e seus *tokens* são definidos com o uso de expressões regulares.
- Portanto, o analisador sintático (*parser*) deve ser capaz de reconhecer uma linguagem do tipo 2, e o analisador léxico (*scanner*), uma linguagem do tipo 3. Por isso, precisaremos fazer uma breve revisão sobre alguns aspectos da teoria de linguagens formais.
- Além disso, é importante destacar que alguns problemas da compilação, por terem maior complexidade, são tratados posteriormente na análise semântica. Exemplos:
  - Declaração de um identificador para seu uso posterior. É semelhante a verificar múltiplas ocorrências de um trecho da linguagem.
  - Idem em relação aos parâmetros de uma função.
  - Validação de expressões com variáveis de tipos diferentes.
  - etc.

Ferramentas automatizadas

- Na década de 50, o desenvolvimento dos primeiros compiladores consumia milhares de horas de trabalho...
- Muita evolução ocorreu desde então, principalmente a hierarquia de Chomsky e os algoritmos de reconhecimento.
- Deste modo, várias técnicas foram automatizadas na realização do processo de compilação:
  - Geradores de analisadores léxicos: produzem *scanners* a partir de expressões regulares que definem os *tokens*. Geralmente são autômatos finitos.
  - Geradores de analisadores sintáticos: produzem *parsers* a partir de gramáticas livres de contexto. Geralmente, são autômatos de pilha.
  - Dispositivos de tradução dirigida pela sintaxe: produzem rotinas que percorrem a árvore sintática, gerando código intermediário de acordo com cada nó.
  - Geradores automáticos de código: produzem código de máquina a partir de regras que definem a tradução de cada operação do código intermediário.