



Compiladores

Aula 1

Horário de aula

- segunda-feira: 16h00-17h40
- terça-feira: 14h00-15h40
- local: Lab. 5b Central

Celso Olivete Júnior

olivete@fct.unesp.br

Tópicos da disciplina

- Introdução à compilação
- Analisador léxico
- Analisador sintático descendente
- Analisador sintático ascendente
- Análise semântica
- Geração de código intermediário
- Ambientes de execução
- Geração de código objeto

Metodologia

- Aulas expositivas teórico-práticas
- Exercícios práticos
- Projetos em duplas (análise individual)

Avaliação

- Avaliação 1: 23-24/09 Avaliação 2: 18-19/11 Exame: 02-03/12
- As notas de todas as atividades serão entre 0 (zero) e 10,0 (dez) e atribuídas individualmente, mesmo em atividades em grupo;
- O desempenho do aluno no primeiro bimestre será avaliado por uma prova (NP1) e notas de trabalhos/projetos(NTP1), sendo que o projeto tem o peso de 70% e os trabalhos de 30%;
- O desempenho do aluno no segundo bimestre será avaliado por uma prova (NP2) e notas de trabalhos/projetos(NTP2), sendo que o projeto tem o peso de 70% e os trabalhos de 30%;
- A média semestral (MS) será calculada da seguinte maneira:
 - Média das notas das provas bimestrais $\rightarrow M_{\text{Provas}} = (NP1 + NP2)/2$
 - Média das notas dos trabalhos/projetos bimestrais $\rightarrow M_{\text{TrabProj}} = (NTP1 + NTP2)/2$
 - $MS = (7 * M_{\text{Provas}} + 3 * M_{\text{TrabProj}})/10$ SE E SOMENTE SE ($M_{\text{Provas}} \geq 5$ E $M_{\text{TrabProj}} \geq 5$)
 - Caso contrário ($M_{\text{Provas}} < 5$ OU $M_{\text{TrabProj}} < 5$)
 - $MB = \text{Menor Nota } (M_{\text{Provas}} \text{ ou } M_{\text{TrabProj}})$

Projeto

- Desenvolvido em duplas
- Avaliação do projeto será na forma de apresentação
 - Notas individuais

Pré-requisitos

❑ Disciplinas:

- Programação
- Teoria da computação
- Linguagens formais e autômatos
- Teoria dos grafos

Bibliografia básica

- ❑ AHO, A. V., ULLMAN, J.D. e SETHI, R., **Compiladores: Princípios, Técnicas e Ferramentas**, LTC, 2008.
- ❑ PRINCE, A. M. A. e TOSCANI, S. S., **Implementação de Linguagens de Programação: Compiladores**, Editora Sagra-Luzzatto, 2001.
- ❑ CRESPO, R. G. **Processadores de Linguagens: Da Concepção à Implementação**, 2ª ed., IST Press, 2001.
- ❑ MENEZES, P. F. B., **Linguagens Formais e Autômatos**, Editora Sagra-Luzzatto, 2001.

Motivação

- ❑ **Compiladores:** uma das principais ferramentas do cientista/engenheiro da computação
- ❑ Técnicas de compilação se aplicam a projetos gerais de programas
 - Editores de texto, sistemas de recuperação de informação, reconhecimento de padrões, ...

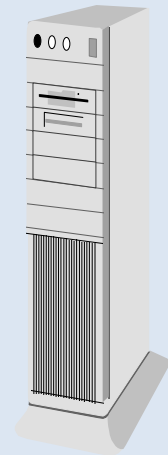
Motivação

❑ Utilização de conceitos e métodos de diversas disciplinas

- Algoritmos
- Linguagens de programação
- Grafos
- Engenharia de software
- Arquitetura de computadores

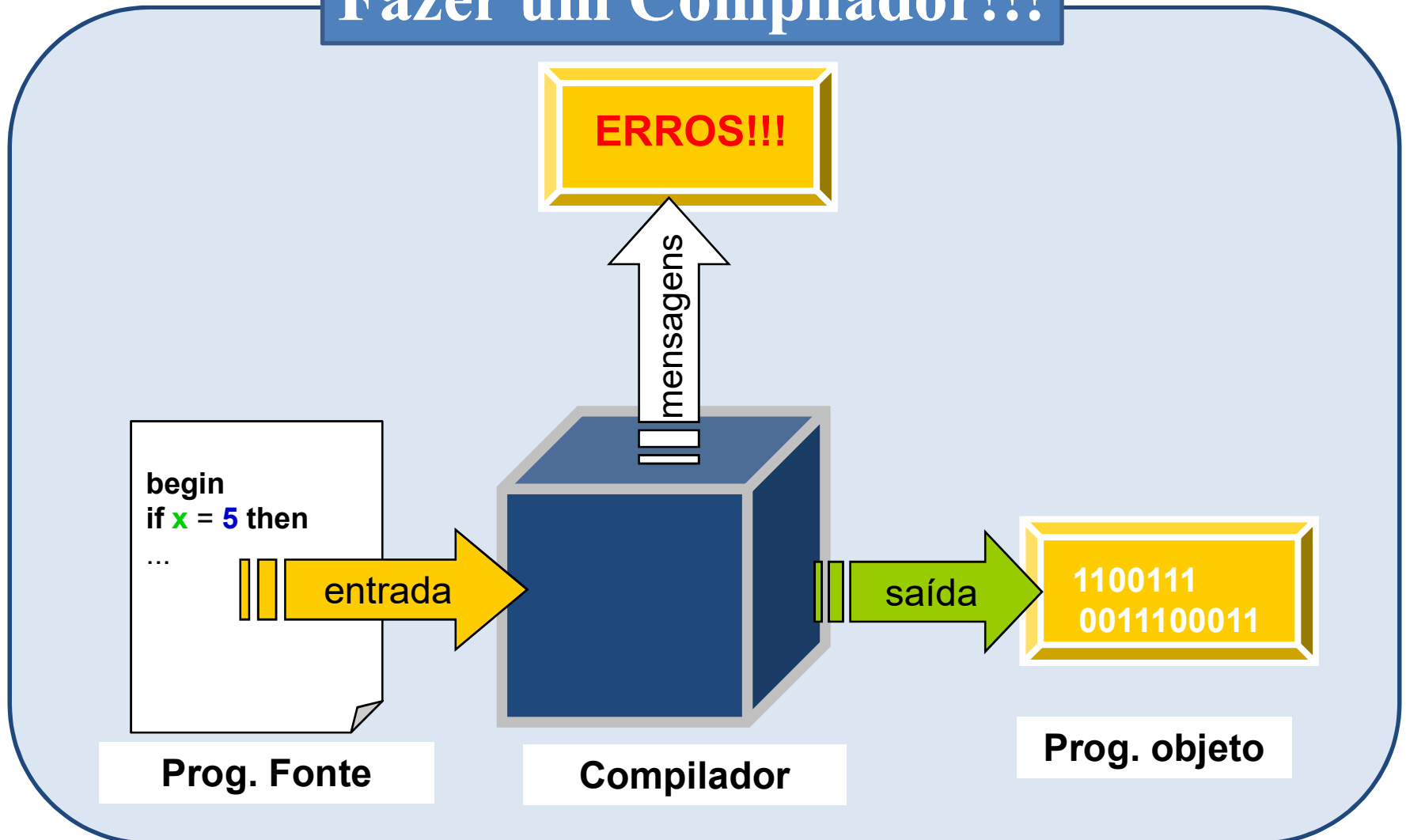


**Linguagens de
Baixo Nível**



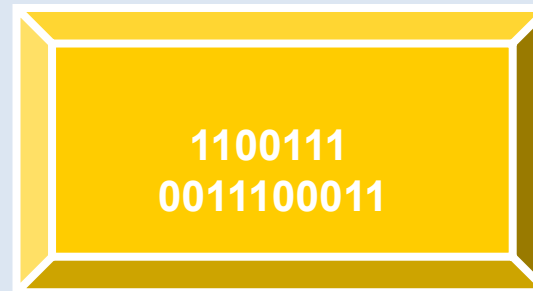
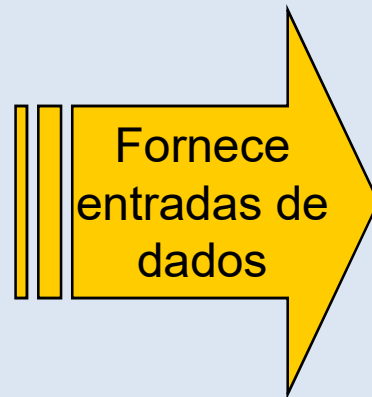
Objetivo

Fazer um Compilador!!!



Objetivo

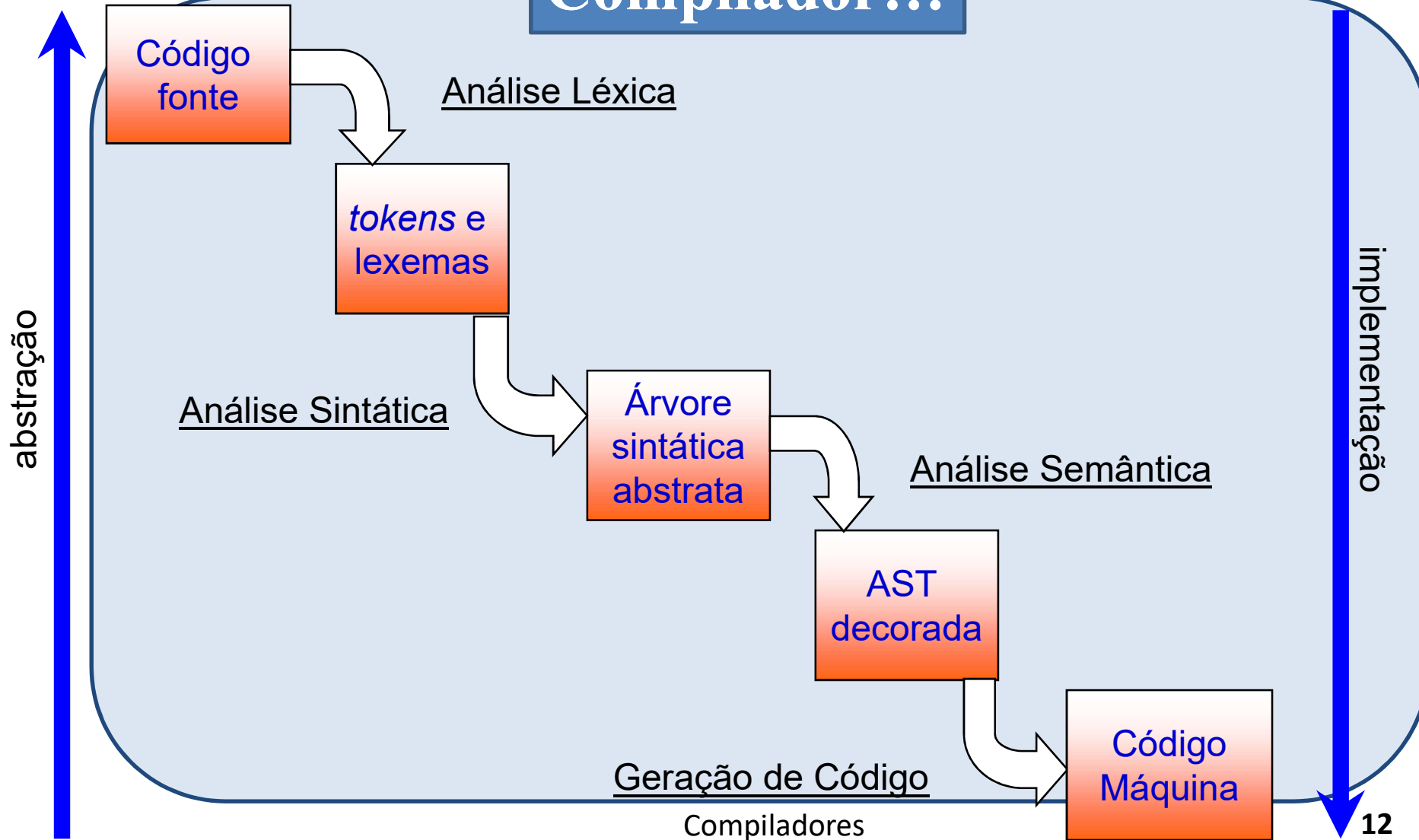
Executando o programa objeto



Prog. objeto

Objetivo

Compilador!!!



Na aula de hoje...

□ **Compilador:** o que é, para que serve e estrutura geral

Roteiro

- Introdução
- Compilação
- Fases da compilação
- Estrutura geral de um compilador
- Definição de linguagens de programação
- Classificação de compiladores

Introdução

- ❑ **Definição:** lê um programa em uma linguagem fonte e o traduz em um programa em uma linguagem-alvo (objeto)
 - ❑ **Linguagem-fonte:** Pascal, C
 - ❑ **Linguagem-alvo:** linguagem de montagem (*assembly*), código de máquina

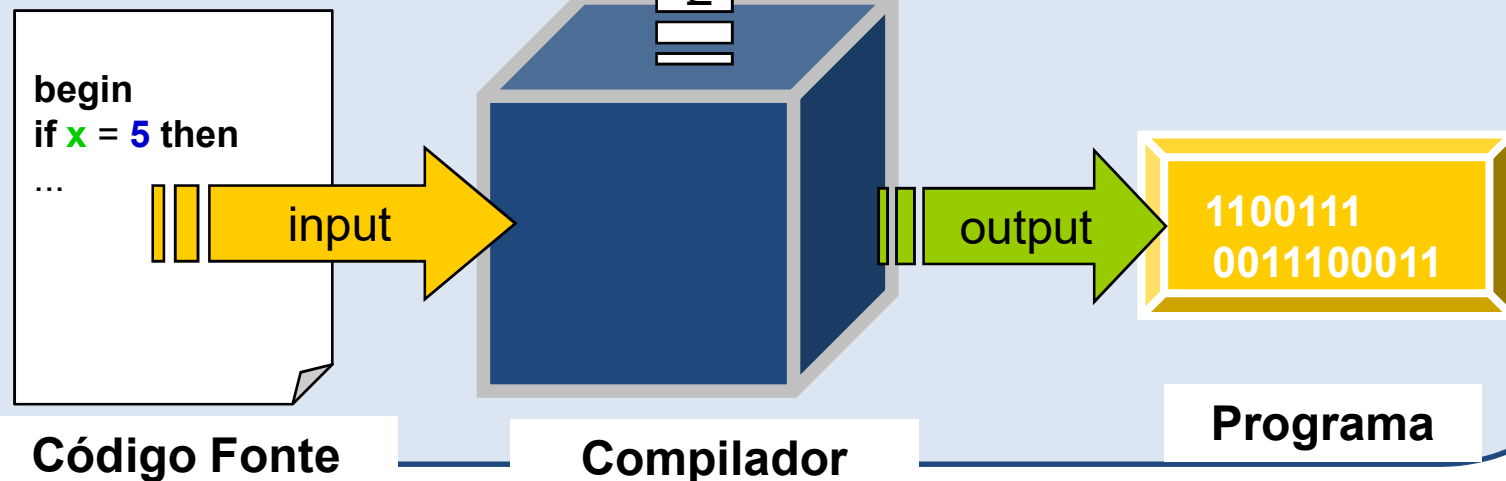
Introdução

- ❑ **Definição:** lê um programa em uma linguagem fonte e o traduz em um programa em uma linguagem-alvo (objeto)

- ❑ **Linguagem-fonte:**

ERROS!!!

- ❑ **Linguagem-alvo:** linguagem de montagem (*assembly*), código de máquina

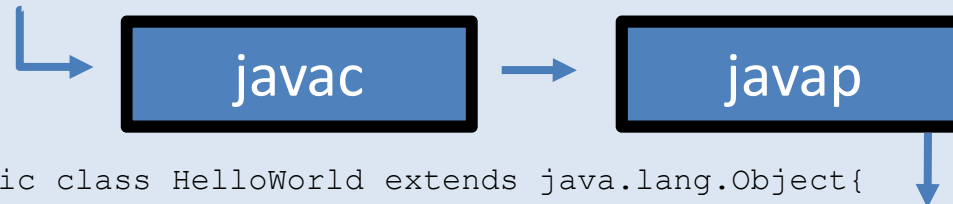


Introdução

- ❑ Tradutor de uma linguagem mais abstrata (**origem**) para uma mais concreta (**destino**).

Exemplo Java:

```
public class HelloWorld {
    public static void main(String[] args) {
        System.out.println("Hello"); } }
```

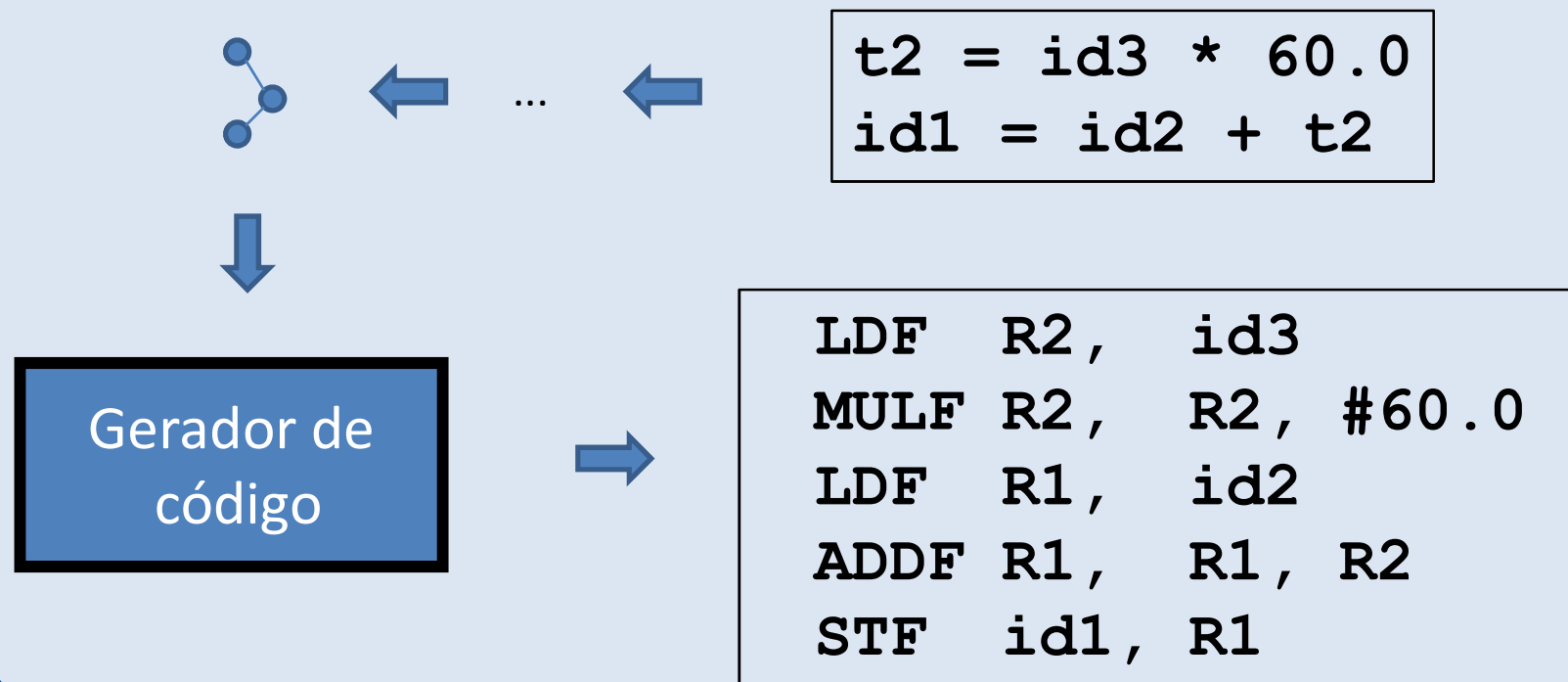


```
public class HelloWorld extends java.lang.Object{
public HelloWorld();
Code:
  0:  aload_0
  1:  invokespecial  #1; //Method java/lang/Object."<init>":()V
  4:  return
public static void main(java.lang.String[]);
Code:
  0:  getstatic      #2; //Field
java/lang/System.out:Ljava/io/PrintStream;
  3:  ldc           #3; //String Hello
  5:  invokevirtual #4; //Method
java/io/PrintStream.println:(Ljava/lang/String;)V
  8:  return
}
```

Código
interpretado
pela JVM

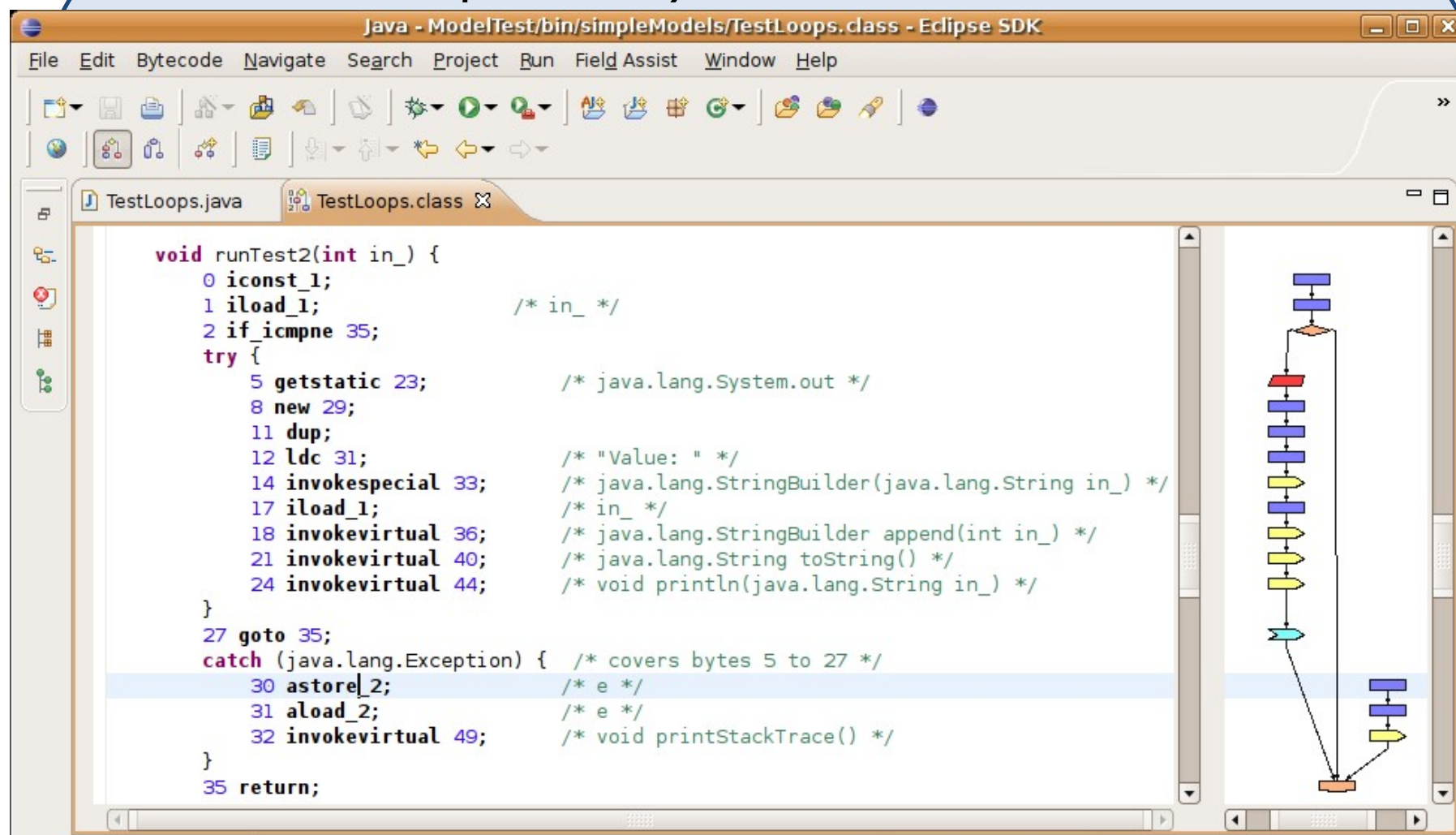
Introdução

- ❑ Tradutor de uma linguagem mais **abstrata** (**origem**) para uma mais concreta (**destino**).
Exemplo: *assembly*



Introdução

❑ Outro exemplo de *bytecodes*



The screenshot displays the Eclipse IDE interface for the file `TestLoops.class`. The left pane shows the source code for the `runTest2` method, which includes a try-catch block. The right pane shows the corresponding bytecode instructions, with a vertical stack of nodes representing the control flow graph. The catch block is highlighted in blue in both panes.

```
void runTest2(int in_) {  
    0 iconst_1;  
    1 iload_1;           /* in_ */  
    2 if_icmpne 35;  
    try {  
        5 getstatic 23;    /* java.lang.System.out */  
        8 new 29;  
        11 dup;  
        12 ldc 31;        /* "Value: " */  
        14 invokespecial 33; /* java.lang.StringBuilder(java.lang.String in_) */  
        17 iload_1;        /* in_ */  
        18 invokevirtual 36; /* java.lang.StringBuilder append(int in_) */  
        21 invokevirtual 40; /* java.lang.String toString() */  
        24 invokevirtual 44; /* void println(java.lang.String in_) */  
    }  
    27 goto 35;  
    catch (java.lang.Exception) { /* covers bytes 5 to 27 */  
        30 astore_2;      /* e */  
        31 aload_2;  
        32 invokevirtual 49; /* void printStackTrace() */  
    }  
    35 return;  
}
```

Compiladores

Funcionalidades

- ❑ ... o programador poderia escrever direto na linguagem destino (código objeto)?

Fácil?
Rápido?

Compiladores

Funcionalidades

- Facilitar programação (**abstração**)
- Checar certos tipos de erros e vulnerabilidades
- Gerar código portátil
- Otimizar código
- Velocidade, tamanho, etc.

Compiladores

Exemplos de Erros

erro de sintaxe

variável não declarada

código inalcançável

variável não inicializada

erro léxico

número ou tipo de argumentos inválidos em chamada de função

Compiladores

História

- ❑ Antigamente a programação era feita em **código de máquina**
- ❑ Programação em **linguagem máquina**
 - Rapidez execução *versus* desenvolvimento complicado
 - Necessidade de um montador
 - ✓ **Não há magia!**
- ❑ Finalmente, linguagens de mais alto nível

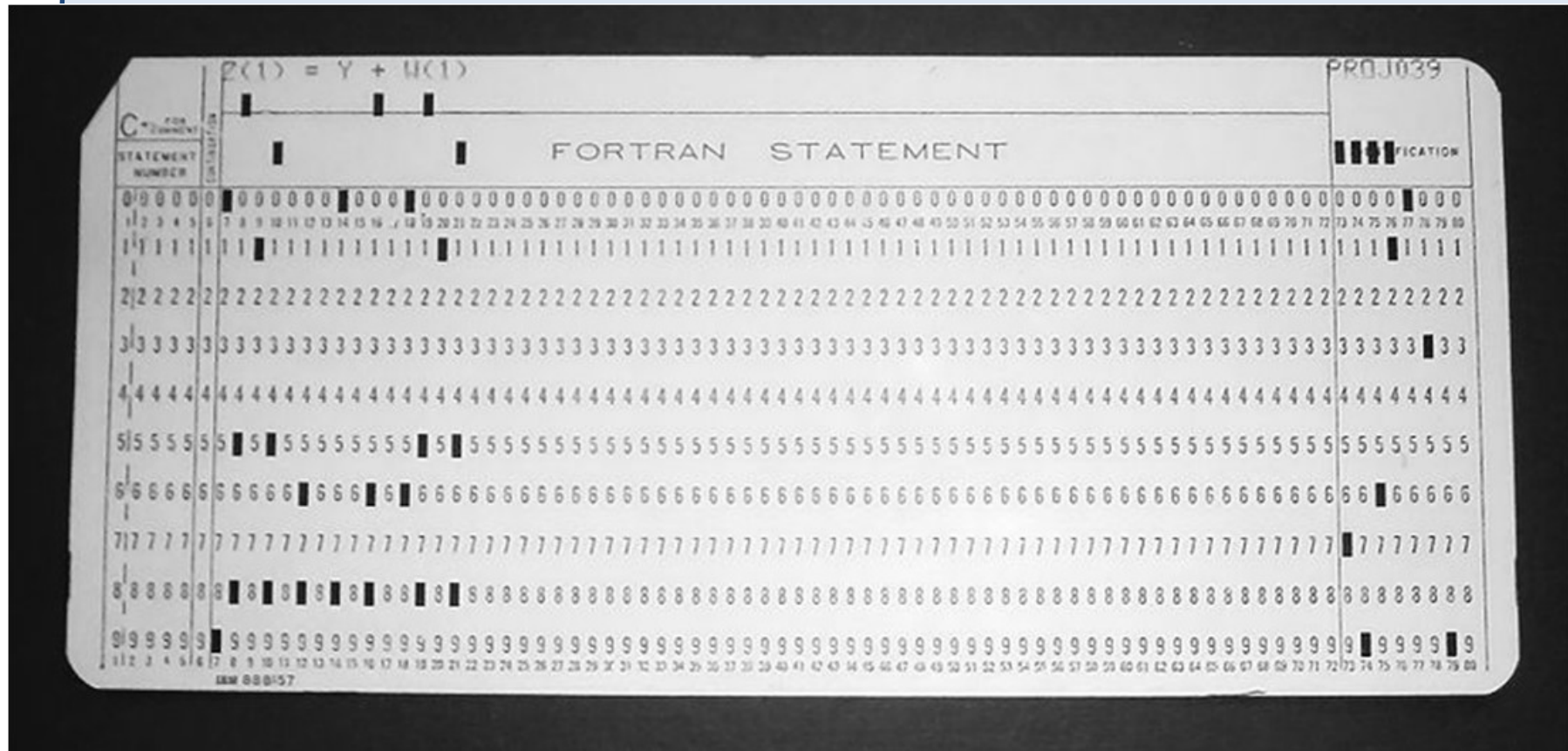
Compiladores

História

- ❑ **Primeiros compiladores** começaram a surgir no início dos **anos 50**
- ❑ **Trabalhos iniciais:** tradução de **fórmulas aritméticas** em código de máquina
- ❑ Compiladores eram considerados programas muito difíceis de construir
- ❑ Primeiro compilador → **Fortran** (permitia a declaração de **identificadores** com até **6 caracteres**)
 - **FOR**mula **TRAN**slation System

Compiladores História

➤ **FOR**mula **TRAN**slation System: cartão perfurado



Compiladores

História

- ❑ Desde então, **técnicas sistemáticas** para construção de compiladores foram identificadas
 - Reconhecimento de cadeias, gramáticas, geração de linguagem
- ❑ Desenvolvimento de **boas linguagens e ambientes de programação**
 - C, C++, linguagens visuais

Compiladores

História

- ❑ Desenvolvimento de programas para produção automática de compiladores

- **lex, flex**

- ✓ Lex → Gerador de analisadores léxicos (UNIX)

- ✓ Flex → Gerador de analisadores léxicos (LINUX / Windows)

- ✓ **Entrada:** Arquivo de descrição do analisador léxico

- ✓ **Saída:** Programa na linguagem "C" que realiza a análise léxica

- Outros geradores de analisadores:

- ✓ TPly - TP Lex / Yacc → Gera um programa em PASCAL

- ✓ JavaCC → Para linguagem Java

- ✓ Flex++ ou Flexx => Para linguagem C++ (orientado a objetos)

Compiladores

História

- ❑ Atualmente, um aluno de graduação pode construir um compilador rapidamente
 - Ainda assim, programa bastante complexo
 - ✓ Estimativa de código acima de 10.000 linhas

Compiladores

História

❑ Com isso, tornou-se uma **área de grande importância**

- 1957: **Fortran** – primeiros compiladores para processamento de expressões aritméticas e fórmulas
- 1960: **Algol** – primeira definição formal de linguagem, com gramática na forma normal de *Backus* (BNF), estruturas de blocos, recursão, etc.

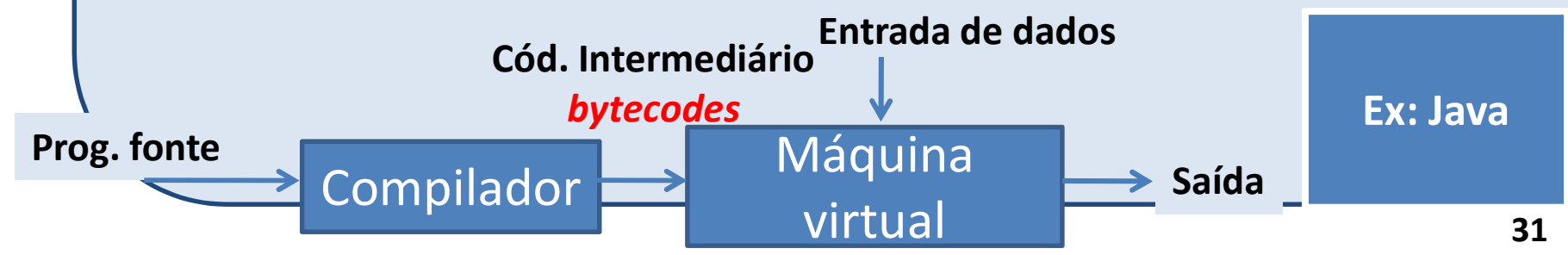
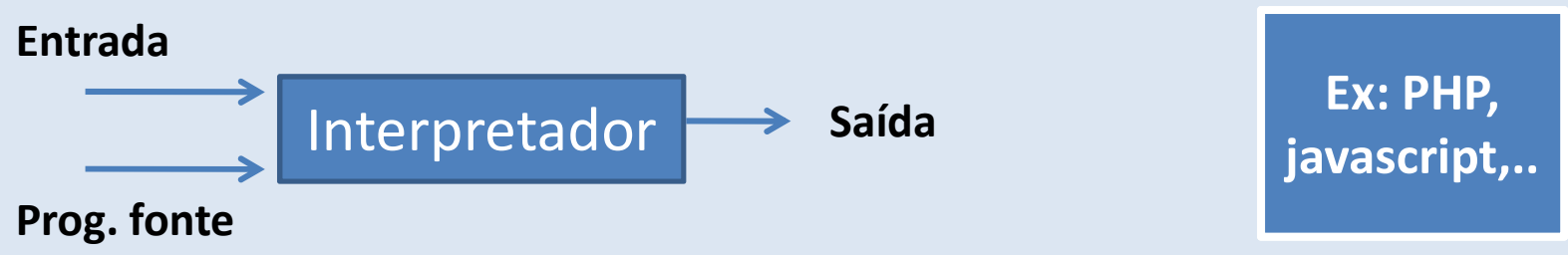
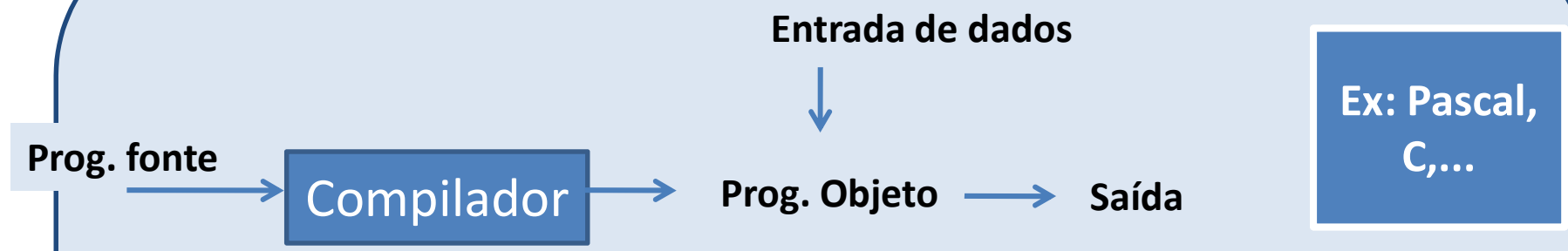
Compiladores

História

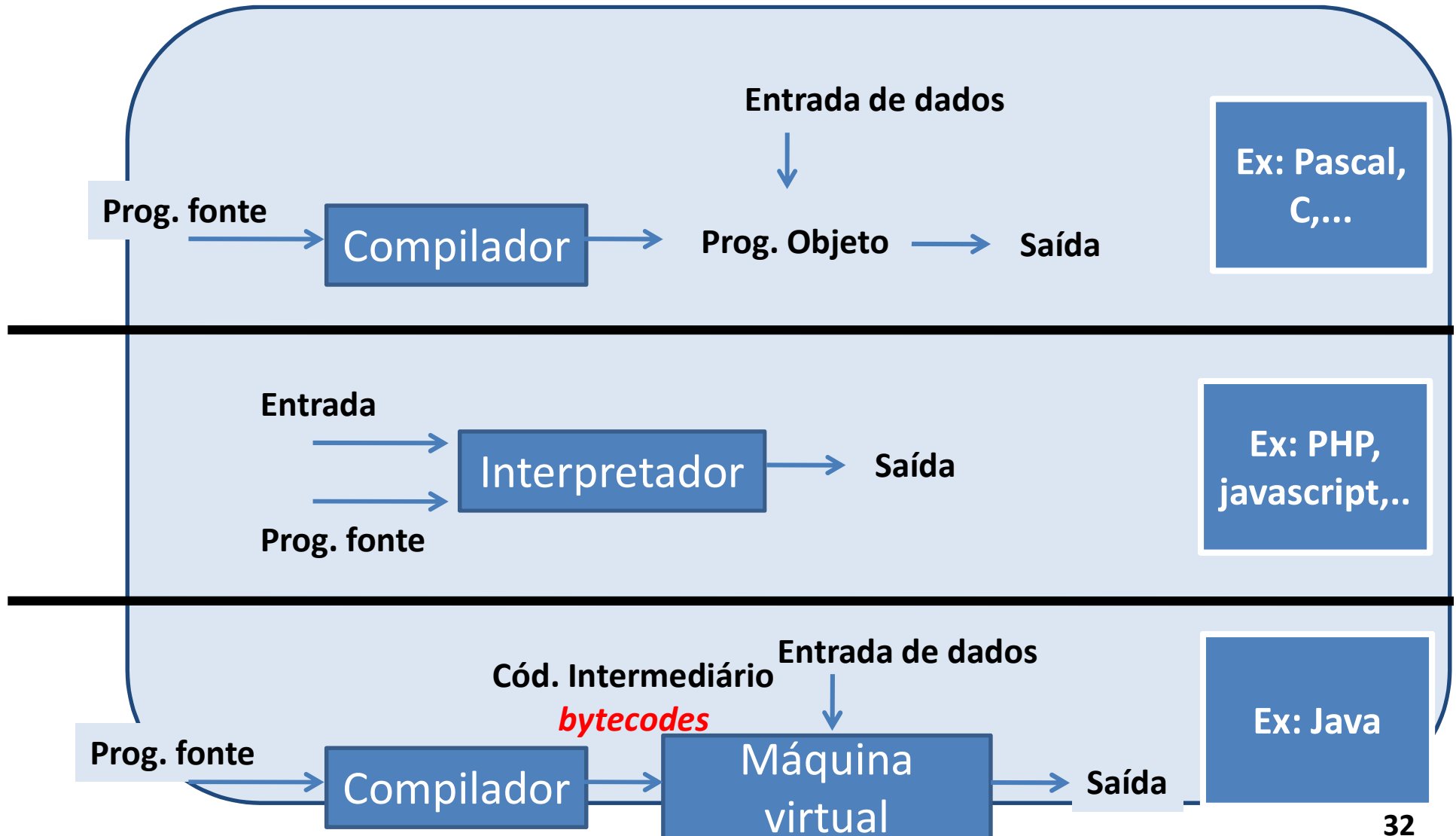
- ❑ Com isso, tornou-se uma **área de grande importância**
 - ❑ 1970: **Pascal** – tipos definidos pelos usuários
 - ❑ 1985: **C++** – orientação a objetos, exceções
 - ❑ 1995: **Java** – compilação instantânea (traduz *bytecodes* para código de máquina e executa), melhorando o tempo e execução do programa. Portabilidade

Organização de um compilador

Compilador, interpretador e máquina virtual

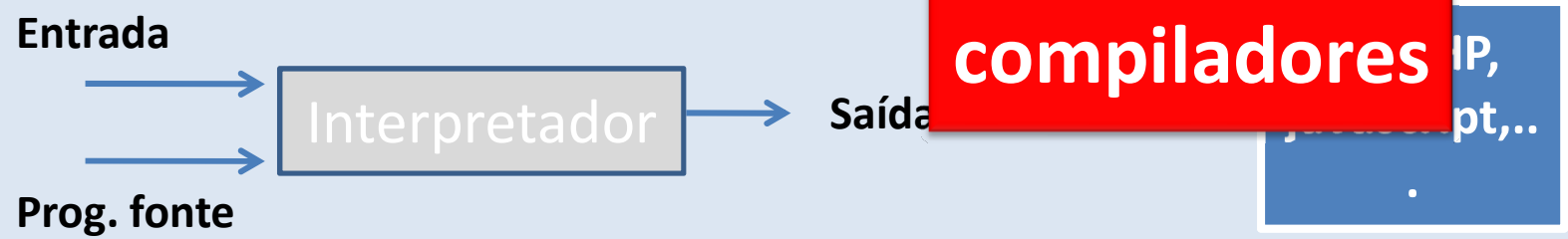
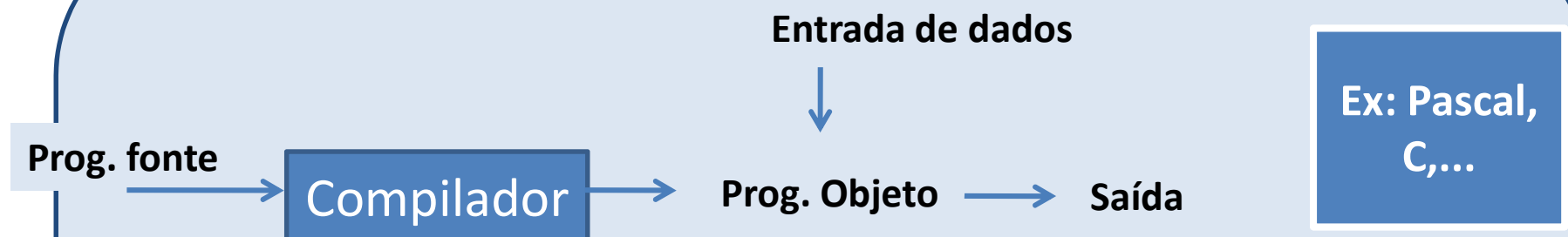


Compilador, interpretador e máquina virtual – Vantagens versus desvantagens

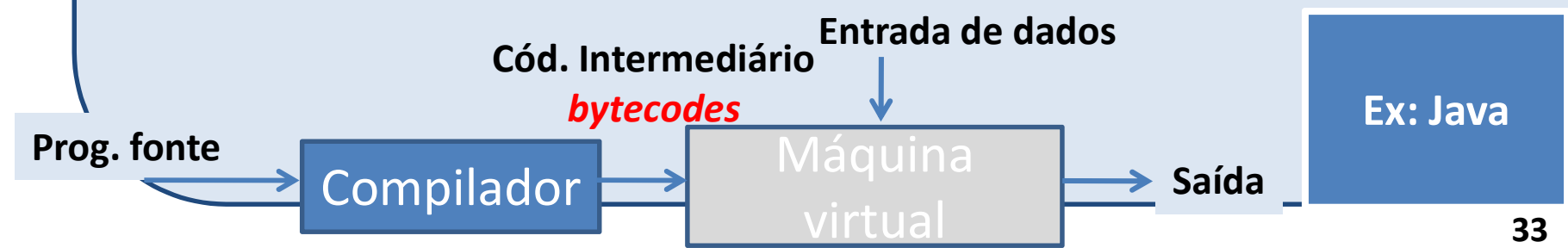


Organização de um compilador

Compilador, interpretador e máquina virtual

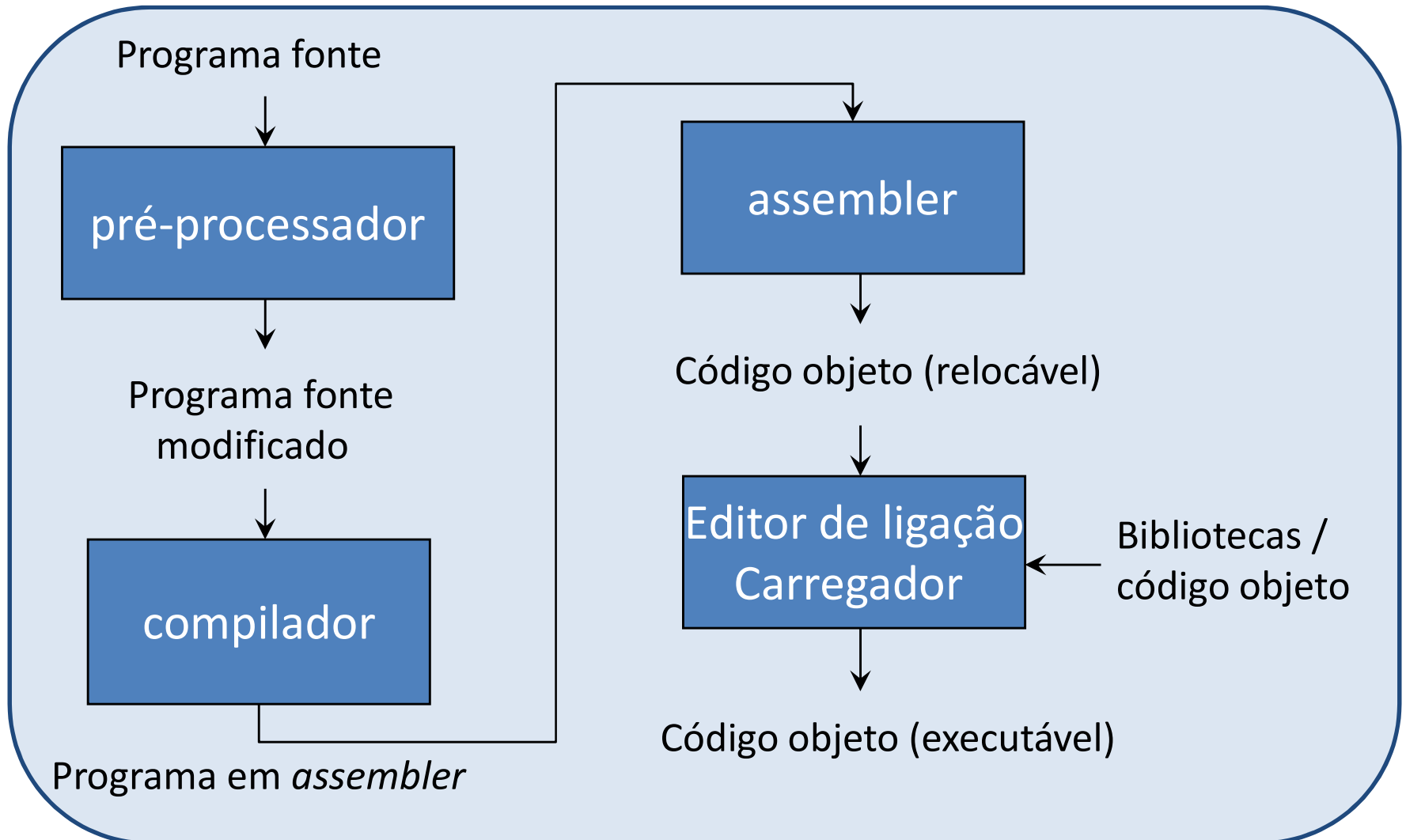


Foco em compiladores

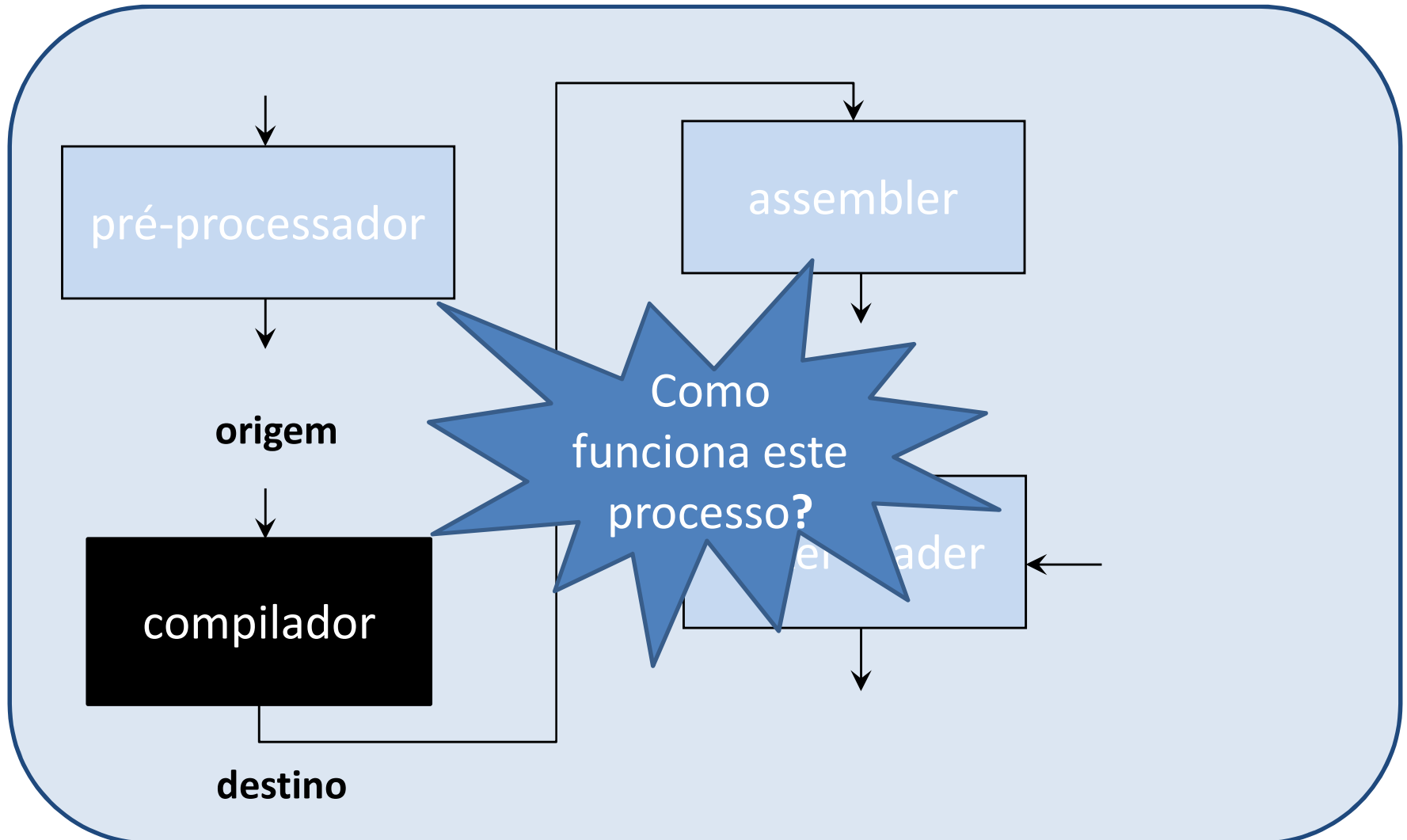


Processo de compilação

subdivisão de um programa fonte



Processo de compilação

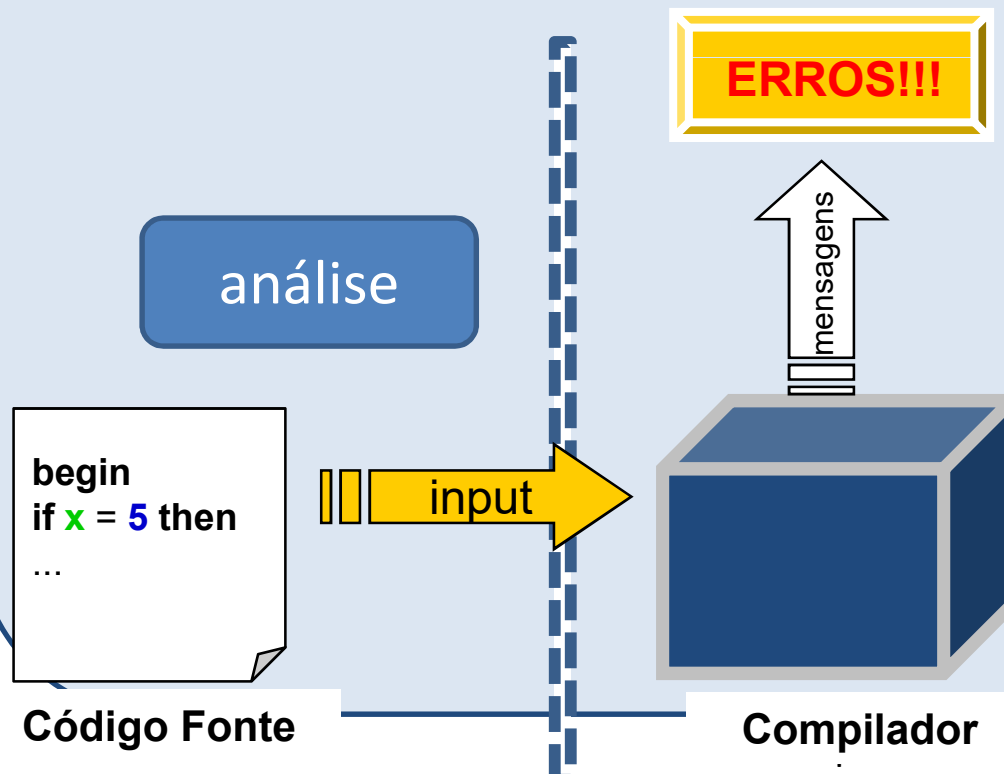


Processo de compilação

□ **Compilação**: duas fases → análise e síntese

1. análise (*front-end*):

- ✓ Cria representações intermediárias do programa (subdivisões)
- ✓ Verifica presença de certos tipos de erro

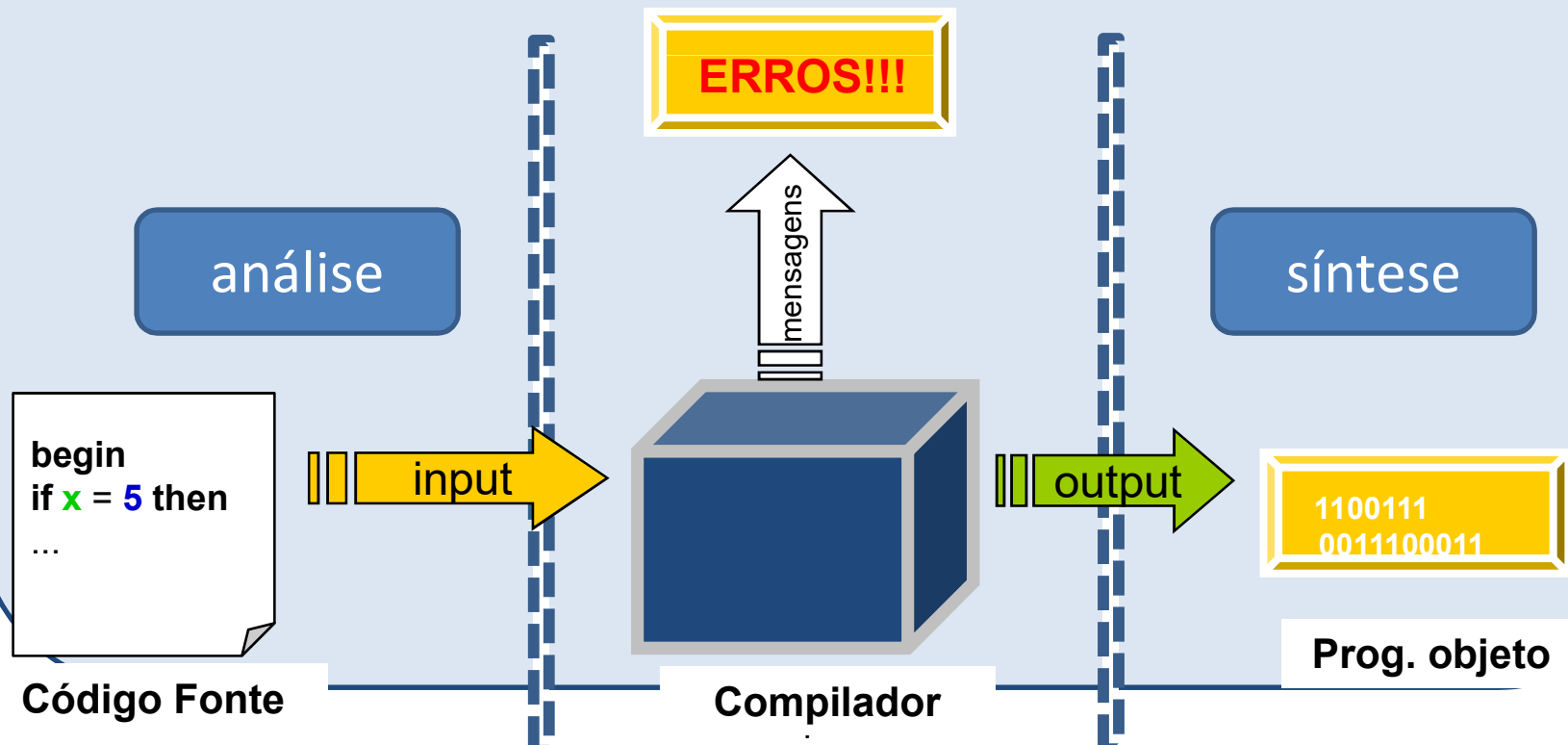


Processo de compilação

❑ **Compilação**: duas fases → análise e síntese

2. síntese (*back-end*):

- ✓ Constrói o programa destino a partir de representações intermediárias





unesp

Processo de compilação

Analisador Léxico

Arquivo

```
1] programa ;
2]
3] var i : integer;
4]   aux char;
5]
6] inicio
7]   escrevaln('teste');
8]   leialn();
9]
10]   se aux <> teste
11]     entao leialn(aux)
12]     senao escrevaln('TEste');
13]
14] fim.
15]
16]
17]
18]
19]
20]
21]
22]
23]
24]
25]
26]
27]
28]
29]
30]
31]
32]
33]
34]
35]
36]
37]
38]
39]
40]
41]
42]
43]
44]
45]
46]
47]
48]
49]
50]
```

Token	Lexema	Linha
<PROGRAMA>	PROGRAMA	1
<PONTOVIRG>	:	1
<VAR>	VAR	3
Id1	I	3
<DOISPONTO>	:	3
Tipo1	INTEGER	3
<PONTOVIRG>	:	3
Id2	AUX	4
Tipo2	CHAR	4
<PONTOVIRG>	:	4
<INICIO>	INICIO	6
<ESCREVALN>	ESCREVALN	7
<ABREPARENTESES>	{	7
Texto	'TESTE'	7
<FECHADARENTESES>	}	7

Erros Léxicos

NENHUM_ERRO_ENCONTRADO

Erros Sintáticos

Esperando " IDENTIFICADOR " antes de ; na linha 1
Esperando " : " antes de <TIPO> na linha 4
Esperando " IDENTIFICADOR " antes de) na linha 8

Analisar

Limpar Dados

Processo de compilação

□ Compilação: duas fases

1. análise (*front-end*):

- ✓ Cria representações intermediárias do programa (subdivisões)
- ✓ Verifica presença de certos tipos de erro

2. síntese (*back-end*):

- ✓ Constrói o programa destino a partir de representações intermediárias

Processo de compilação

1) Análise

1.1) Análise léxica

- ✓ Organiza caracteres de entrada em grupos, chamados *tokens*
- ✓ **Erros:** tamanho máximo da variável excedido, caracteres inválidos..

1.2) Análise sintática

- ✓ Organiza *tokens* em uma estrutura hierárquica
- ✓ **Erros:** falta de (,), =, **identificador inválido...**

Processo de compilação

□ Análise

1.3) Análise semântica

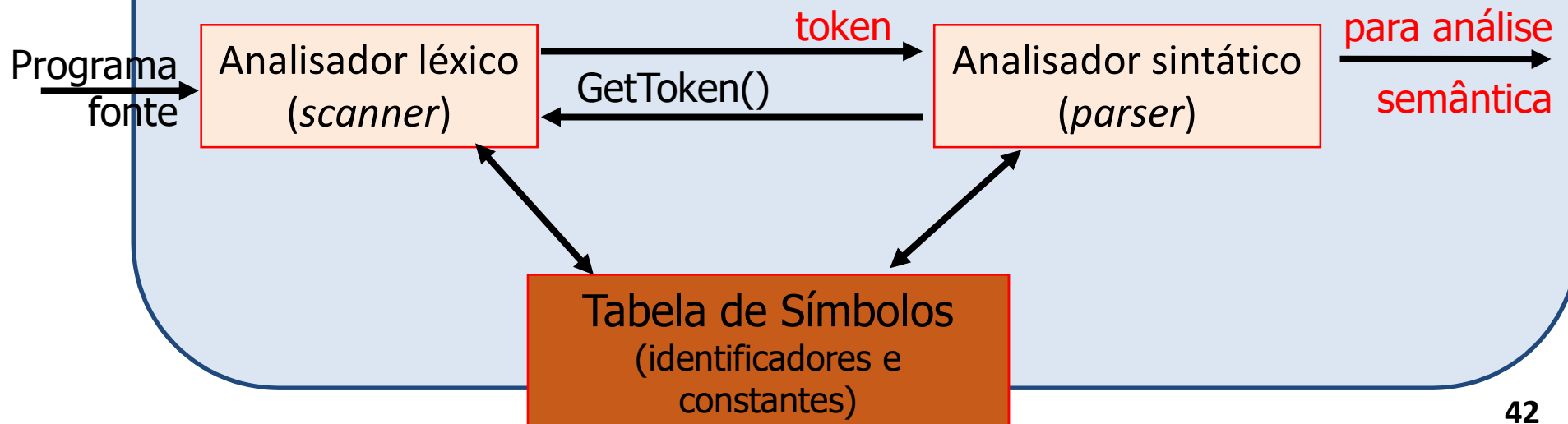
- ✓ Checa se o programa respeita regras básicas de consistência
- ✓ **Erro:** tipos inconsistentes → atribuir uma *string* em uma variável inteira

Processo de compilação

1) análise

1.1) Análise léxica

- ❑ Lê os caracteres de entrada (*scanner*) e os agrupa em sequências chamadas **lexemas** (*tokens*)
- ❑ Os **tokens** são consumidos na fase seguinte (**análise sintática**)



Processo de compilação

1) análise

1.1) Análise léxica

- ❑ A **tabela de símbolos** é utilizada para diferenciar palavras e símbolos reservados da linguagem (while, for, :=, >, <) **de identificadores definidos pelo usuário**

Tabela de símbolos

Lexema (lido)	Token (tipo)
;	<PONTO_VIRG>
aux	<IDENTIFICADOR>
while	<PALAVRA RESERVADA>



Arquivo

```

1] programa ;
2]
3] var i : integer;
4]   aux char;
5]
6] inicio
7]   escrevaln('teste');
8]   leialn();
9]
10]   se aux <> teste
11]     entao leialn(aux)
12]     senao escrevaln('TEste');
13]
14] fim.
15]
16]
17]
18]
19]
20]

```

Token	Lexema	Linha
<PROGRAMA>	PROGRAMA	1
<PONTOVIRG>	:	1
<VAR>	VAR	3
Id1	I	3
<DOISPONTO>	:	3
Tipo1	INTEGER	3
<PONTOVIRG>	:	3
Id2	AUX	4
Tipo2	CHAR	4
<PONTOVIRG>	:	4
<INICIO>	INICIO	6
<ESCREVALN>	ESCREVALN	7
<ABREPARENTESES>	{	7
Texto	'TESTE'	7
<FECHADARENTESES>	}	7

Erros Léxicos

NENHUM_ERRO_ENCONTRADO

Erros Sintáticos

Esperando " IDENTIFICADOR " antes de ; na linha 1
 Esperando " : " antes de <TIPO> na linha 4
 Esperando " IDENTIFICADOR " antes de) na linha 8

Analisar

Limpar Dados

Finalizar

Processo de compilação

1) análise

1.1) Análise léxica

- ❑ A **tabela de símbolos** também é utilizada para armazenar o tipo e o valor das variáveis e o seu escopo

Tabela de símbolos

Lexema (lido)	Token (tipo)	valor
;	<PONTO_VIRG> ou ;	-
aux	<IDENTIFICADOR>	10
while	<PALAVRA RESERVADA>	-

Processo de compilação

1) análise

1.1) Análise léxica. Exemplo:

`expr = a + b * 60`



Analizador
Léxico

Tabela de
Símbolos

<identificador, 1>, <=>,
<identificador, 2>, <+>,
<identificador, 3>, <*>,
<numero, 60>

	nome	tipo	
1	expr	-	...
2	=	-	
3	a	-	
...			

Processo de compilação

1) análise

1.1) Análise léxica – Reconhecimento e classificação dos *tokens*

- ❑ Pode ser feito com o uso de **expressões regulares e autômatos finitos**

Processo de compilação

1) análise

1.1) Análise léxica – Reconhecimento e classificação dos *tokens*

□ Exemplos de expressões regulares

- letra → [A-Z] | [a-z]
- dígito → [0-9]
- dígitos → dígito dígito*
- identificador → letra[letra | dígito]*

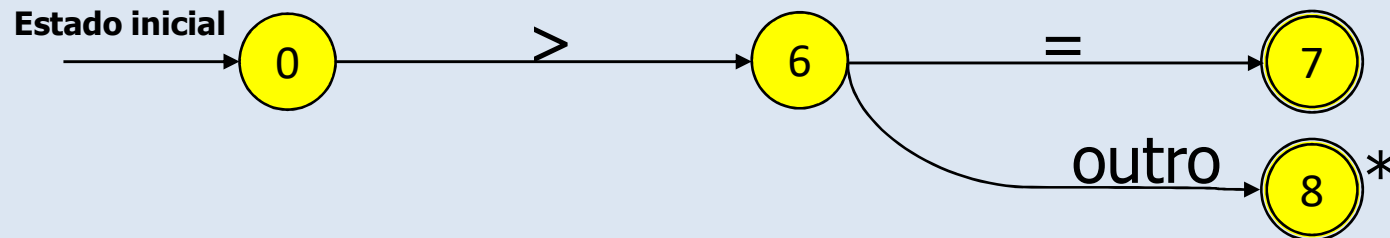
Processo de compilação

1) análise

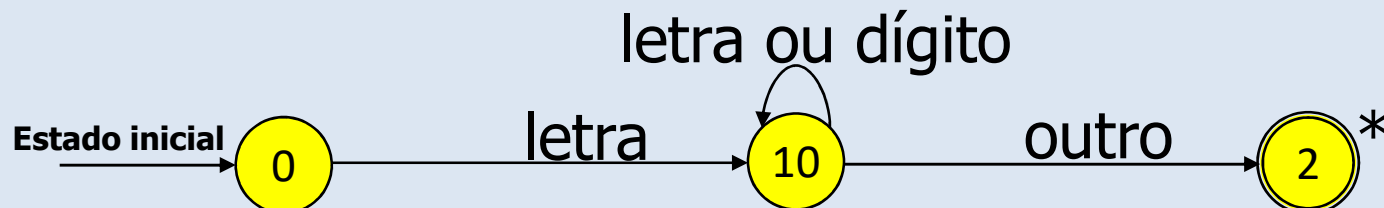
1.1) Análise léxica – Reconhecimento e classificação dos *tokens*

Exemplos de autômatos finitos

Exemplo para > e >=



Exemplo para identificadores



Processo de compilação

1) análise

1.2) Análise sintática

- ❑ Utiliza os **tokens** produzidos pela análise léxica e verifica a formação do programa com o uso de **GLC** (gramáticas livres de contexto)
 - A partir dos **tokens** cria uma representação intermediária tipo árvore (árvore sintática) → mostra a estrutura gramatical da sequência de **tokens**

Processo de compilação

1) análise

1.2) Análise sintática

`expr = a + b * 60`

`<identificador, 1>, <=>`,

`<identificador, 2>, <+>`,

`<identificador, 3>, <*>`,

`<numero, 60>`

Analizador
Sintático

`<id,1>`

=

+

`<id,2>`

*

`<id,3>`

60

Árvore sintática:

Percorrendo a árvore e consultando a GLC é possível verificar se a expressão pertence à linguagem

Nó interior representa uma operação

Nó filho representa um argumento

Processo de compilação

1) análise

1.3) Análise semântica

- ❑ Utiliza a árvore sintática e a tabela de símbolos para:
 - ❑ Criar a consistência semântica (**significado**) do prog. fonte em relação à linguagem.
 - ❑ Exemplo: verificação de tipos
 - A expressão

`x = x + 3.0;`

está sintaticamente correta, mas pode estar semanticamente errada, dependendo do tipo de x.

Processo de compilação

❑ Compilação: duas fases

1. análise (*front-end*):

- ✓ Cria representações intermediárias do programa
- ✓ Verifica presença de certos tipos de erro

2. síntese (*back-end*):

- ✓ Constrói o programa destino a partir de representações intermediárias

Processo de compilação

2) síntese

□ Geração de código:

- Recebe como entrada uma representação intermediária (fases da análise **léxica e sintática**) e transforma em uma **linguagem objeto**
 - ✓ Alocação de memória, uso de registradores

Processo de compilação

2) síntese

❑ Geração de código:

➤ Código intermediário

➤ Exemplos de representações:

- ✓ **3 endereços:** cada instrução usa não mais que três operandos
- ✓ **Pilha:** operandos são acessíveis apenas a partir da pilha

Processo de compilação

2) síntese

□ Geração de código:

- **Código intermediário**
- Exemplo: 3 endereços

```
id1 = id2 + id3 * inttfloat(60)
```



```
t1 = inttfloat(60)
t2 = id3 * t1
t3 = id2 + t2
id1 = t3
```


Processo de compilação

2) síntese

Otimizador de código

Realiza transformações no código com objetivo de melhorar algum aspecto relevante

- tempo de execução, consumo de memória, tamanho do código executável, etc.

Processo de compilação

2) síntese

❑ Otimizador de código

```
t1 = inttofloat(60)
t2 = id3 * t1
t3 = id2 + t2
id1 = t3
```



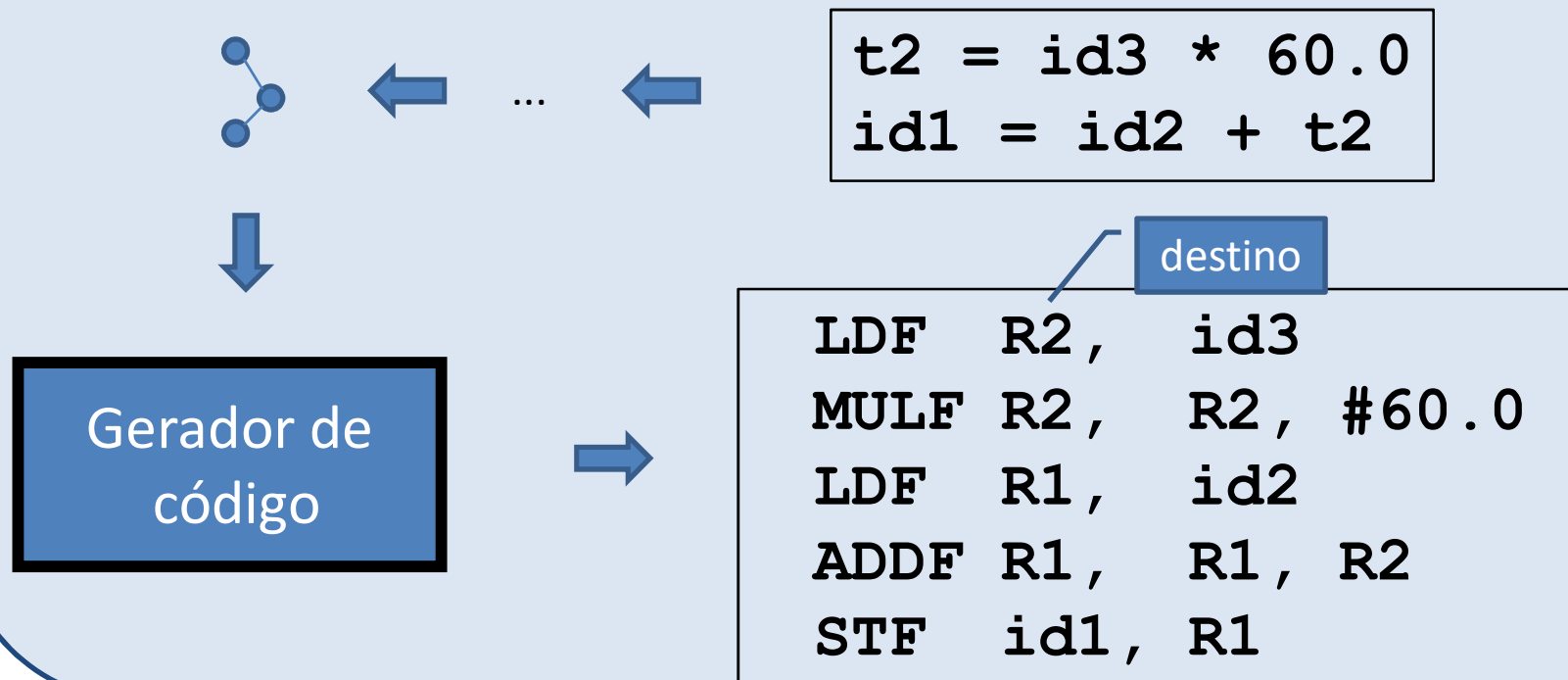
```
t2 = id3 * 60.0
id1 = id2 + t2
```

Processo de compilação

2) síntese

❑ Geração de código

- ❑ Consiste em traduzir o código intermediário para a linguagem-destino (p.ex, *assembly*)





Projeto de um analisador léxico

❑ **EXERCÍCIO** – Projetar um analisador léxico para uma calculadora simples com números naturais e reais e operações básicas (soma, subtração, multiplicação e divisão)

- ❑ Obs: 1. deverá ser feito em dupla – a mesma que irá desenvolver o projeto da disciplina
- 2. data de entrega: 07/08



Projeto de um analisador léxico

❑ Questões a considerar:

1. Que símbolo usar como separador de casa decimais?
2. A calculadora usa representação monetária?
3. A calculadora aceita espaços entre os operandos e operadores?
4. O projetista é quem decide sobre as características desejáveis do compilador ou interpretador. Para a maioria das linguagens de programação existem algumas **convenções** que devem ser respeitadas



Projeto de um analisador léxico

□ **Exemplo** - seja a cadeia $3.2 + (2 * 12.01)$, o analisador léxico teria como saída:

3.2 => número real

+ => operador de soma

(=> abre parênteses

2 => número natural

* => operador de multiplicação

12.01 => número real

) => fecha parênteses



Projeto de um analisador léxico

1. Definição do Alfabeto

$$\Sigma = \{0,1,2,3,4,5,6,7,8,9,.,(,),+,-,*,/, \backslash b\}$$

- **OBS.**: projetista deve considerar TODOS os símbolos que são necessários para formar os padrões



Projeto de um analisador léxico

2. Listagem dos *tokens*

- OPSOMA: operador de soma
 - OPSUB: operador de subtração
 - OPMUL: operador de multiplicação
 - OPDIV: operador de divisão
 - AP: abre parênteses
 - FP: fecha parênteses
 - NUM: número natural/real
- **OBS.:** projetista deve considerar *tokens* especiais e cuidar para que cada *token* seja uma unidade significativa para o problema

Projeto de um analisador léxico

3. Especificação dos *tokens* com definições regulares

- OPSOMA $\rightarrow +$
- OPSUB $\rightarrow -$
- OPMUL $\rightarrow *$
- OPDIV $\rightarrow /$
- AP $\rightarrow ($
- FP $\rightarrow)$
- NUM $\rightarrow [0-9]^+.[?][0-9]^*$

❑ **OBS.:** cuidar para que as definições regulares reconheçam padrões claros, bem formados e definidos



Projeto de um analisador léxico

4. Montar os autômatos para reconhecer cada ***token***

- **OBS.:** os autômatos reconhecem ***tokens*** individuais, mas é o conjunto dos autômatos em um único autômato não-determinístico que determina o analisador léxico de um compilador, por isto, deve ser utilizada uma numeração crescente para os estados.



Projeto de um analisador léxico

5. Implementar o analisador léxico

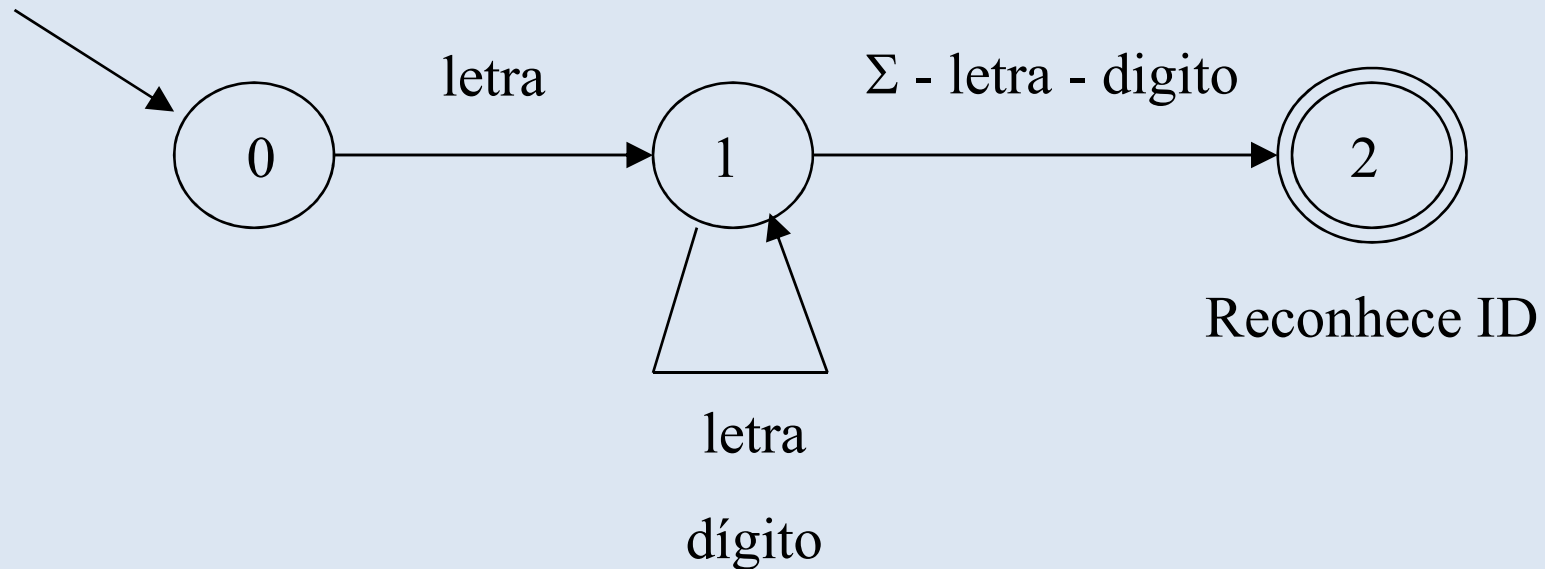
- Existem duas formas básicas para implementar os autômatos: usando a **tabela de transição** ou **diretamente em código**

Estilo de implementação

- ❑ Cada **token** listado é codificado em um número natural
- ❑ Deve haver uma variável para controlar o **estado** corrente do autômato e outro para indicar o estado de **partida** do autômato em uso
- ❑ Uma função **falhar** é usada para desviar o estado corrente para um outro autômato no caso de um estado não reconhecer uma letra

Estilo de implementação

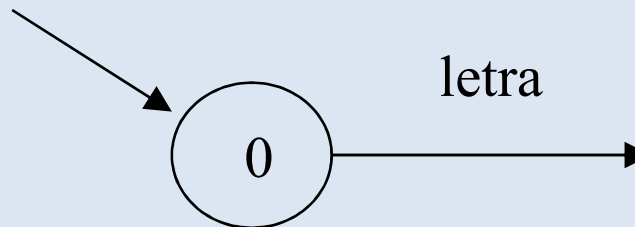
- ❑ Cada estado é analisado individualmente em uma estrutura do tipo **switch...case**



Estilo de implementação

- ❑ Cada estado é analisado individualmente em uma estrutura do tipo **switch...case**

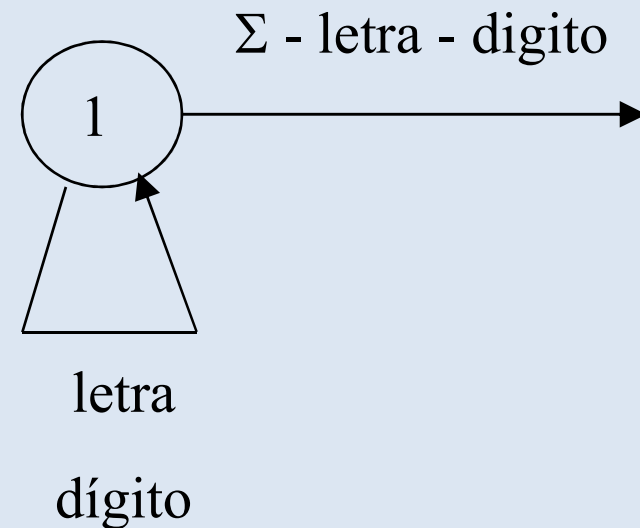
```
int lexico()
{
    while (1)
    {
        switch (estado)
        {
            case 0: c= proximo_caracter();
                    if (isalpha(c))
                    {
                        estado= 1;
                        adiante++;
                    }
                    else
                    {
                        falhar();
                    }
                    break;
            ...
        }
    }
}
```



Estilo de implementação

- ❑ Cada estado é analisado individualmente em uma estrutura do tipo **switch...case**

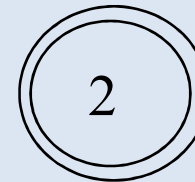
```
...
case 1: c= proximo_caracter();
        if (isalpha(c) || isdigit(c))
        {
            estado= 1;
            adiante++;
        }
        else
        {
            if ((c == '\n') || (c == '\t') || (c == '\b')) estado= 2;
            else falhar();
        }
        break;
...
}
```



Estilo de implementação

- ❑ Cada estado é analisado individualmente em uma estrutura do tipo **switch...case**

```
...  
    case 2: estado= 0;  
            partida= 0;  
            return ID;  
            break;  
    }  
}
```



Reconhece ID

www.inf.ufes.br/~tavares/labcomp2000/analex.html

Na próxima aula...

- Revisão de linguagens formais.