

# *Paradigmas de Programação*

## *Aula 3*

*Linguagens de Programação*

*Desenvolvimento e*

*execução de programas*

*Características de linguagens*

*Execução de programas*

*Prof.: Rodrigo D Malara (adaptado de  
Edilberto M. Silva)*

# Domínios de Programação

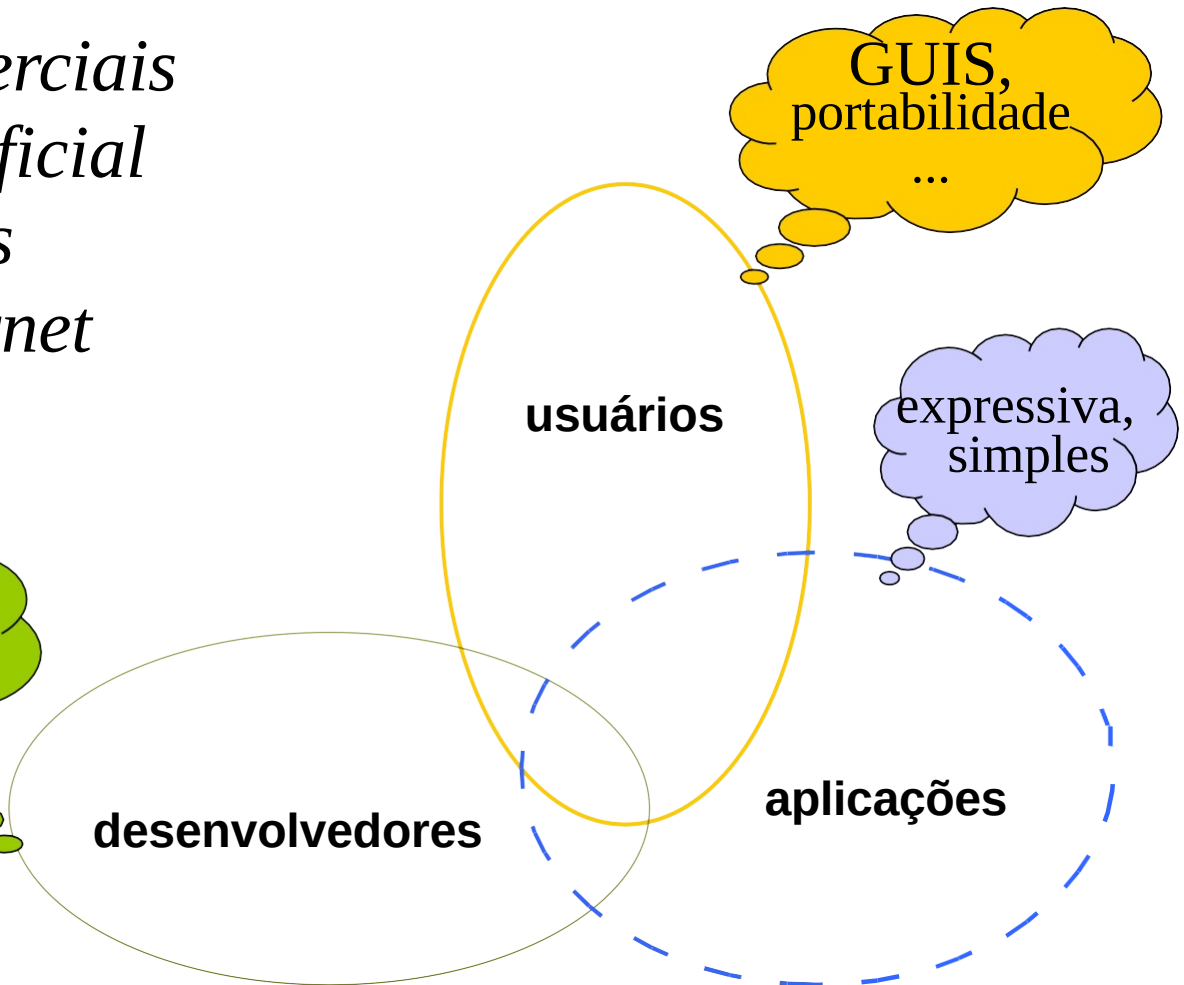
☁️ Aplicações

☁️ Aplicações comerciais

☁️ Inteligência artificial

☁️ Sistemas básicos

☁️ Aplicações Internet




# *Modelos de Linguagens de Programação*

## *Linguagens imperativas*

 *programação estruturada modular*

 *programação orientadas a objetos*

## *Linguagens declarativas*

 *programação funcional*

 *programação em lógica*

## *Linguagens concorrentes*


 *programação paralela*


# Característica: *legibilidade*

 *Facilidade de ler e escrever programas*

 *Legibilidade influi:*

 *desenvolvimento e depuração de programas*

 *manutenção de programas*

 *desempenho de equipes de programação*

 *Fatores que melhoram a legibilidade:*

 *Aumentar o nível de abstração*

 *Comandos de controle – variedade de opções*

 *Modularização de programas*

 *Documentação*

 *Convenções léxicas, sintaxe e semântica*

- *Exemplo em Java: nomes de classes iniciam por letra maiúscula, nomes de atributos usam letras minúsculas*
- *Uso de ferramentas de padronização: ex: Checkstyle*

# Característica: *simplicidade*

- ☁️ *Representação de cada conceito seja simples de aprender e dominar*
  - ✂️ *Simplicidade **sintática** exige que a representação seja feita de modo preciso, sem ambigüidades*
    - *contra-exemplo:  $A++$ ;  $A=A+1$ ;  $A+=1$ ;  $++A$ .*
  - ✂️ *Simplicidade **semântica** exige que a representação possua um significado independente de contexto*
- ☁️ *Simplicidade não significa concisão*
  - ✂️ *A linguagem pode ser concisa mas usar muitos símbolos especiais*
  - ✂️ *Exemplo: linguagens funcionais*

# Característica: *expressividade*

☁️ Representação clara e simples de dados e procedimentos a serem executados pelo programa

✂️ Exemplo: tipos de dados em Pascal

☁️ Expressividade x concisão

✂️ muito concisa: falta expressividade?

✂️ muito extensa: falta simplicidade?

☁️ Linguagens mais modernas:

✂️ incorporam apenas um conjunto básico de representações de tipos de dados e comandos

✂️ aumentam o poder de expressividade com bibliotecas de componentes

✂️ Exemplos: Pascal, C++ e Java

# Característica: *ortogonalidade*

☁ Possibilidade de combinar entre si, sem restrições, os componentes básicos da LP

✂ Exemplo: permitir combinações de estruturas de dados, como arrays de registros

✂ Contra exemplo: não permitir que um array seja usado como parâmetro de um procedimento

☁ *Componente de primeira ordem*: pode ser livremente usado em expressões, atribuições, como argumento e retorno de procedimentos




☁ Influenciada pelo modelo de LP

✂ Modelo de Objetos: objeto


✂ Modelo funcional: funções

# Característica: *portabilidade*

## Multiplataforma:

-  capacidade de um software rodar em diferentes plataformas sem a necessidade de maiores adaptações
-  Sem exigências especiais de hardware/software
-  Exemplo: aplicação compatível com sistemas Unix e Windows

## Longevidade:

-  ciclo de vida útil do software e o do hardware não precisam ser síncronos; ou seja, é possível usar o mesmo software após uma mudança de hardware



# Característica: *confiabilidade*

 *Mecanismos que facilitem a produção de programas que atendam às sua especificações*

 *Tipagem forte: o processador da linguagem deve*

- *assegurar que a utilização dos diferentes tipos de dados seja compatível com a sua definição*
- *evitar que operações perigosas, tal como aritmética de ponteiros, seja permitida*

 *Tratamento de exceções: sistemas de tratamento de exceções permitem construir programas que*

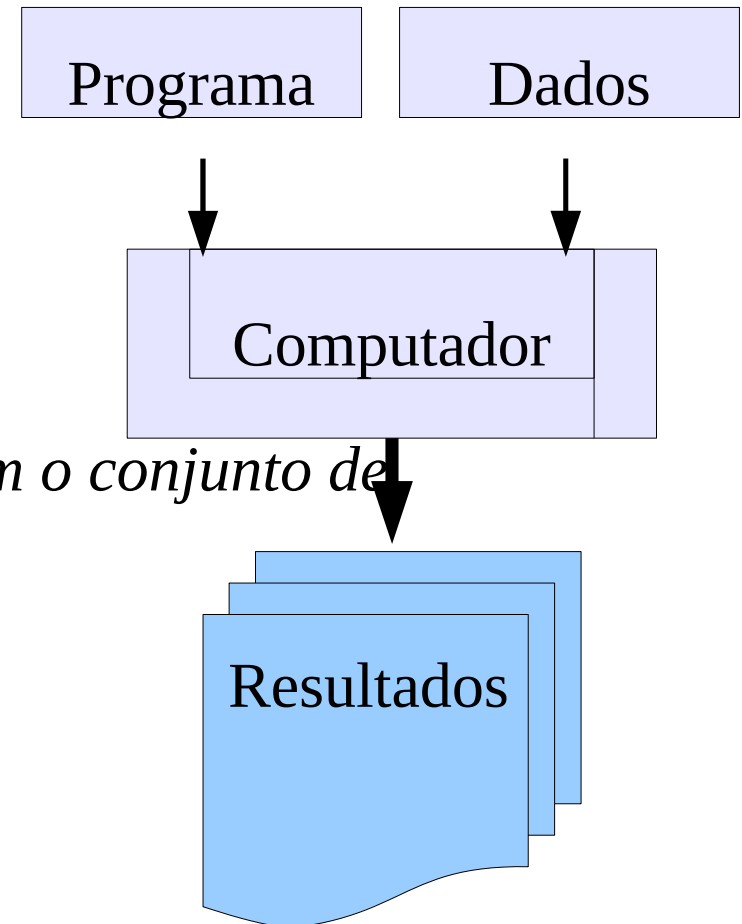
- *possuam definições de como proceder em caso de comportamento não usual*
- *possibilitem tanto o diagnóstico quanto o tratamento de erros em tempo de execução*

# Introdução: *execução de programas*




📁 **Componentes:** programa, dados, computador

📁 **Objetivo:** viabilizar a execução de um programa fonte, juntamente com seus dados, em um computador para a obtenção dos resultados

📁 **Problema:** notação usada no programa pode ser incompatível com o conjunto de instruções executáveis



# Computador e linguagens

-  Um computador pode ser representado por:
-  uma **máquina virtual**, capaz de executar operações mais abstratas (representadas através de uma linguagem de programação)
  -  uma **máquina real**, capaz de executar um determinado conjunto de operações concretas (expressas em linguagem de máquina)

$L_1$ : máquina  
assembler

$L_2$  máquina C

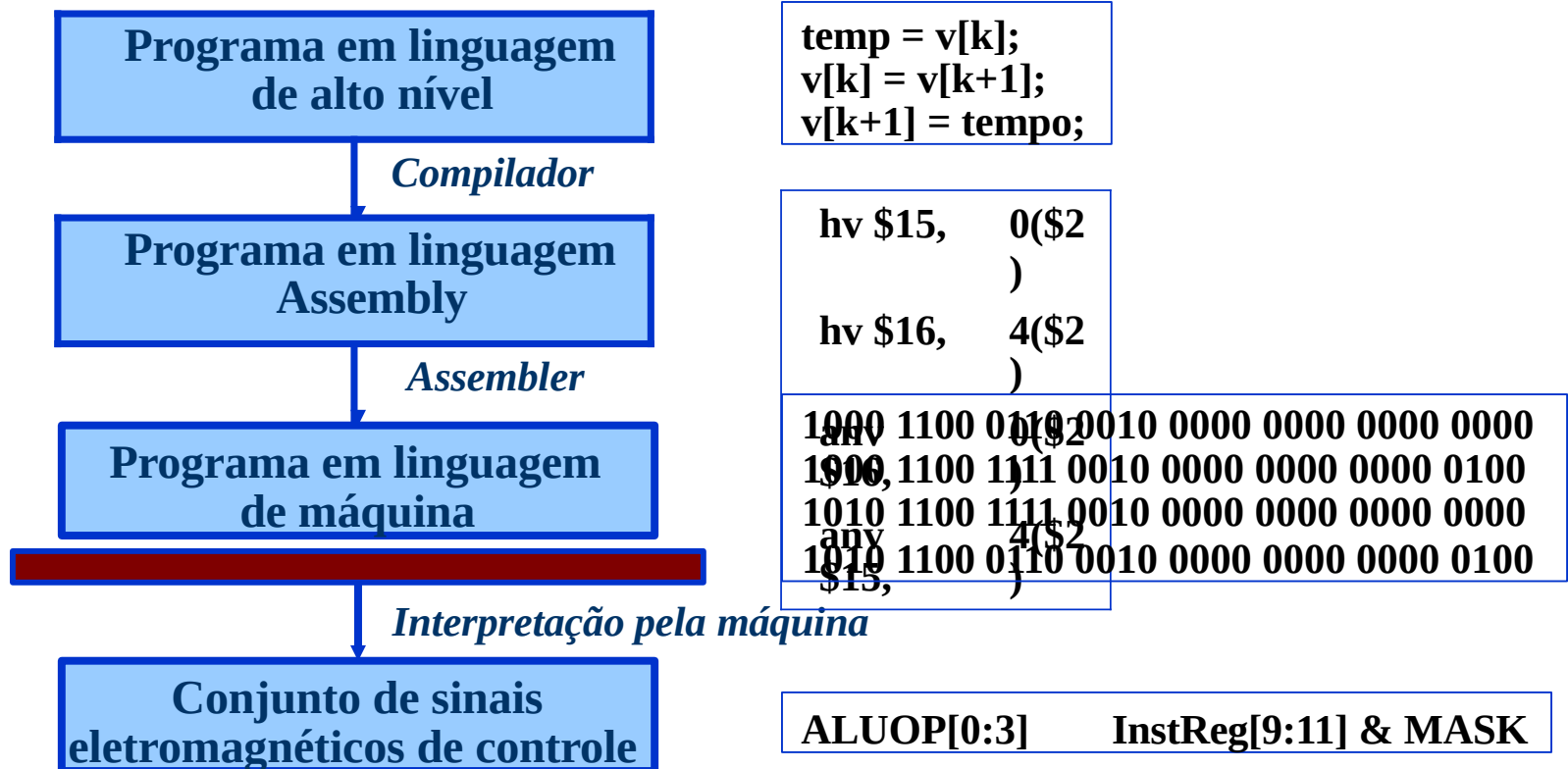
$L_3$  máquina Pascal




$L_0$ : máquina real

# Linguagens e Máquinas


- ☰ Cada máquina representa um conjunto integrado de estruturas de dados e algoritmos
- ☰ Cada máquina é capaz de armazenar e executar programas em uma linguagem de programação  $L_1, L_2, L_3, \dots, L_n$ .



# *Máquina reais e máquinas virtuais*

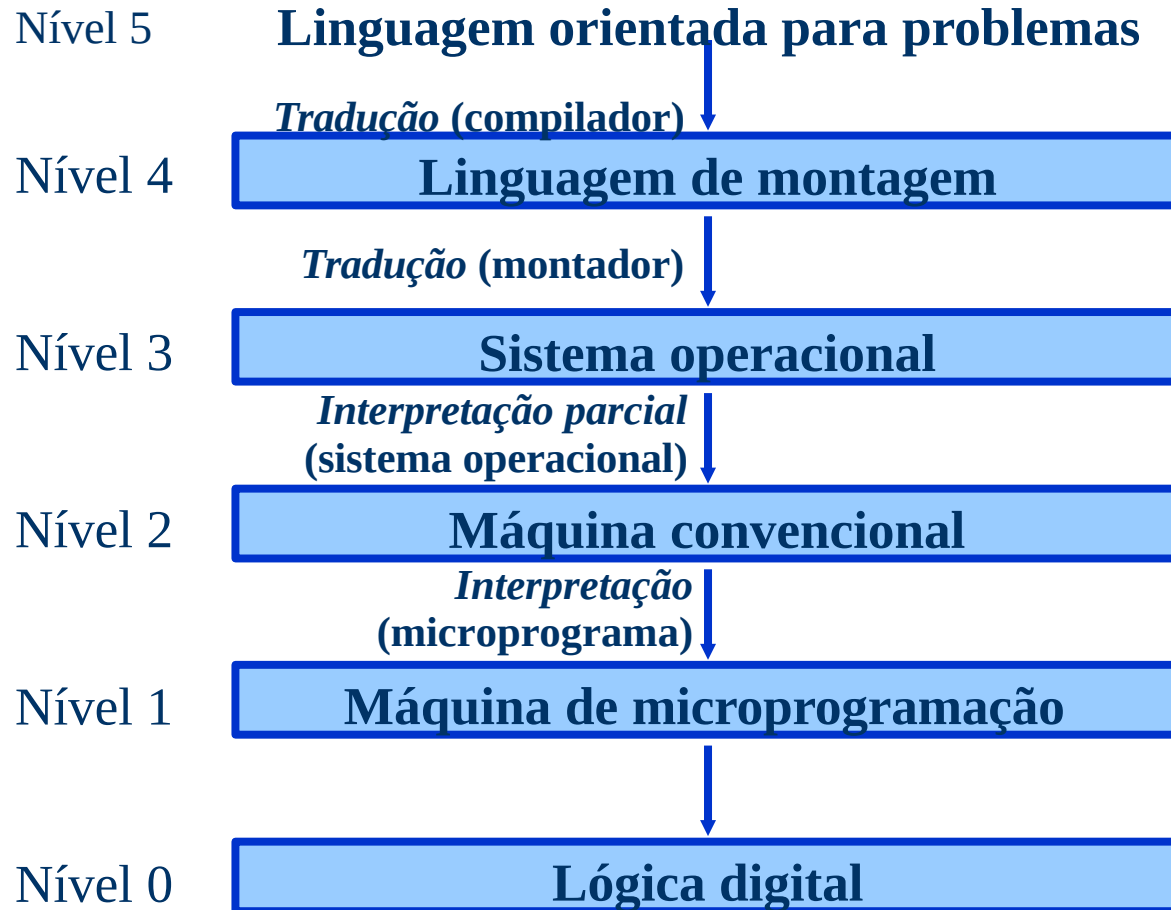
 **Máquina real:** conjunto de hardware e sistema operacional capaz de executar um conjunto próprio de instruções (plataforma de execução)

 O elo de ligação entre a máquina abstrata e a máquina real é o processador da linguagem

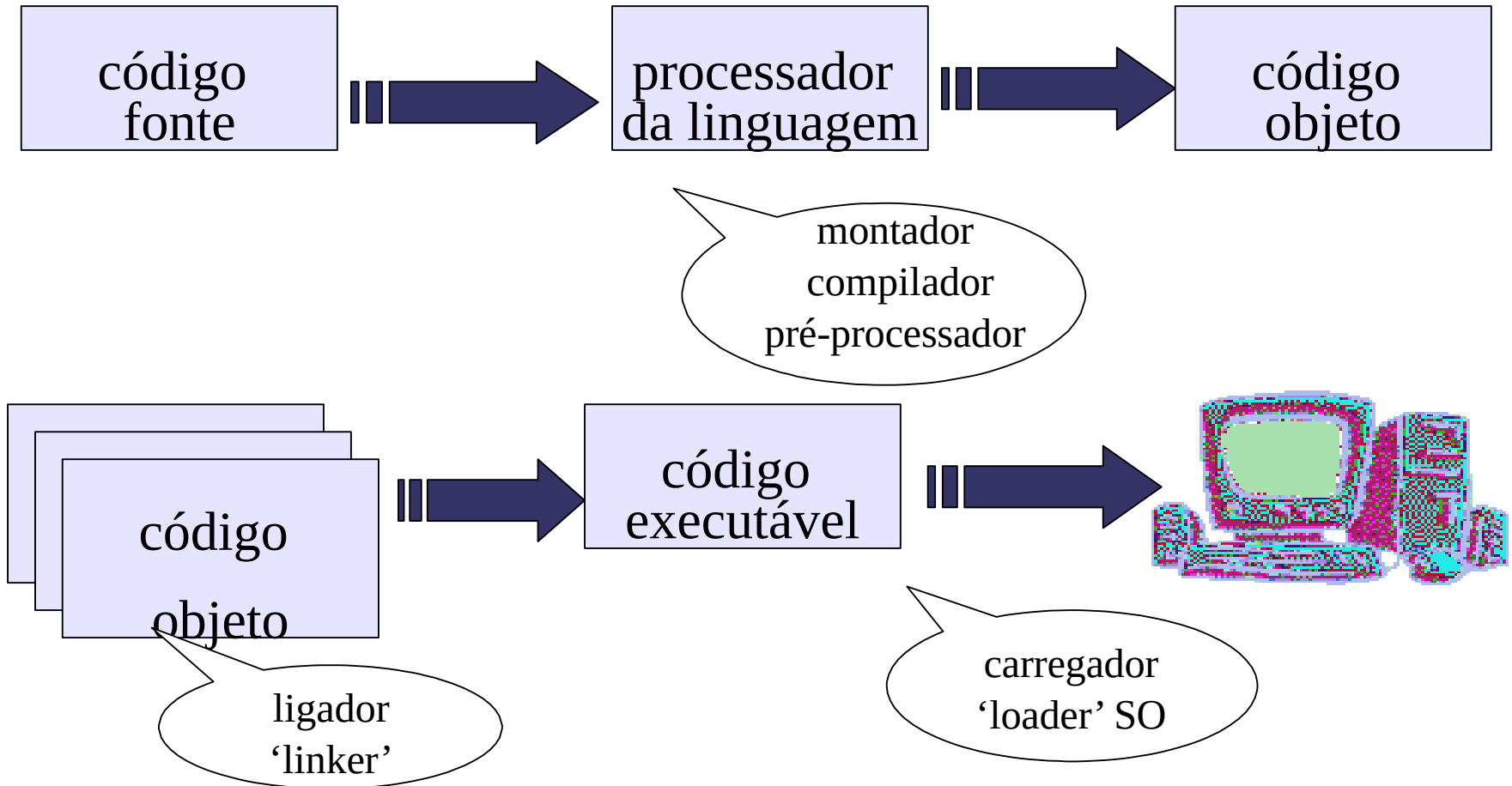
 **Processador da linguagem:** programa que traduz as ações especificadas pelo programador usando a notação própria da linguagem em uma forma executável em um computador

 **Máquina abstrata (virtual):** combinação de um computador e um processador de linguagem

# *Máquina reais e máquinas virtuais*



# *Preparação de programas*

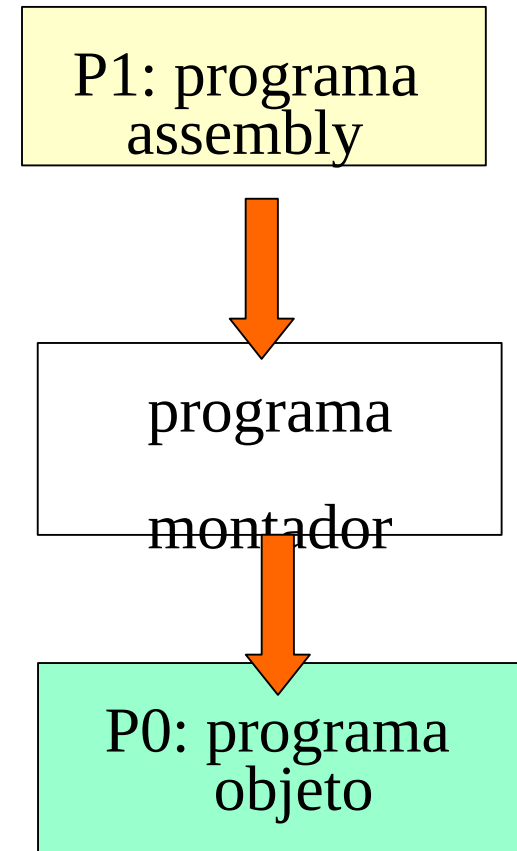


# Programa Montador (assembler)

- ☁ Montadores aceitam como entrada um programa escrito em linguagem de montagem (assembly) e produzem código de máquina correspondente a cada instrução
- ☁ Sendo P1 o programa assembler e P0 o programa em linguagem de máquina, P1 e P0 devem ser funcionalmente equivalentes

Linguagem é 'assembly'

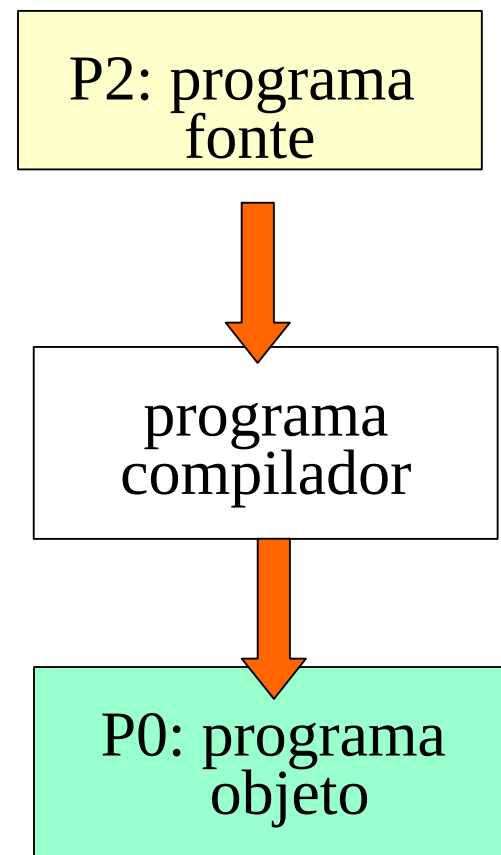
O montador do 'assembly' se chama assembler



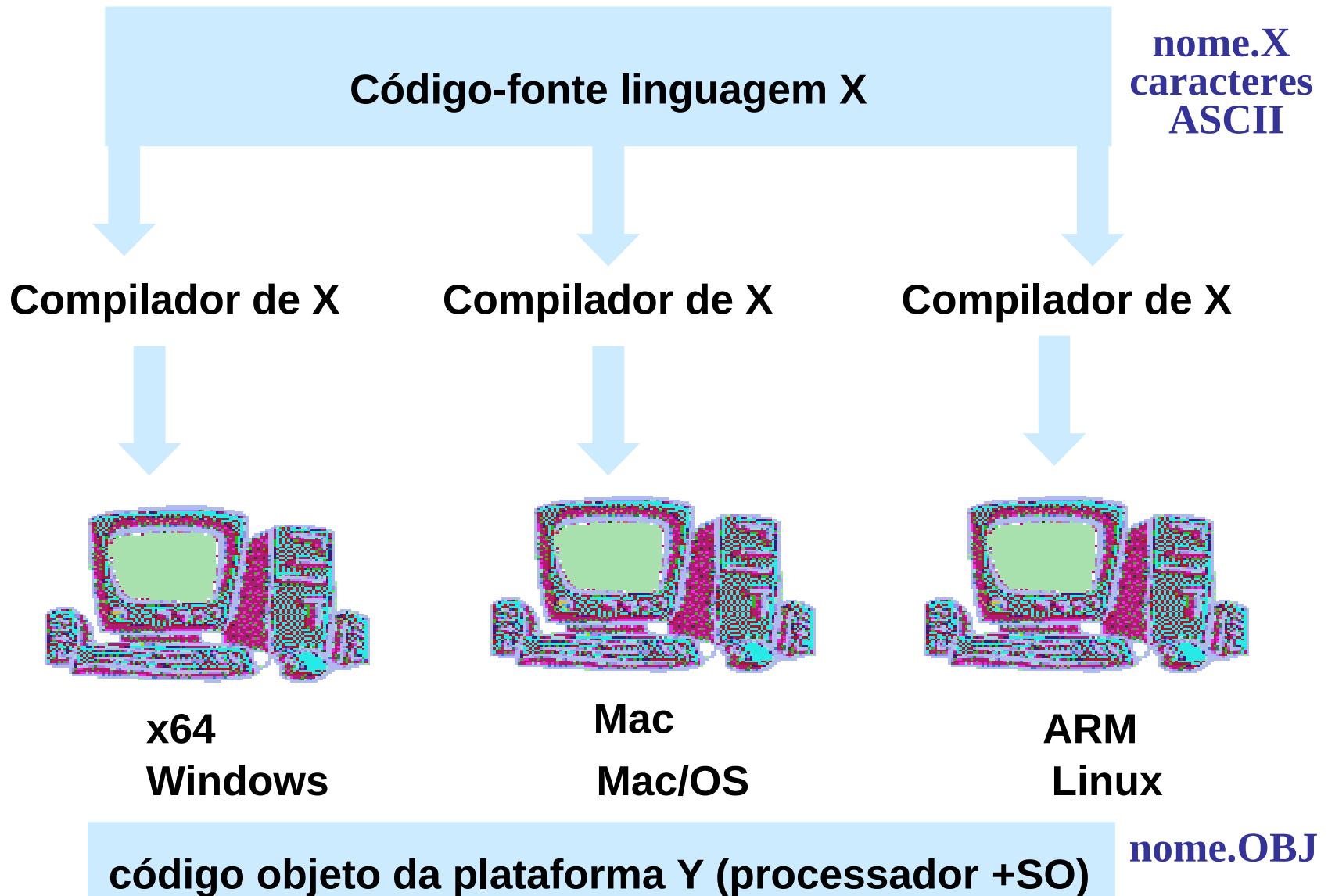


# Programa Compilador

- ☁ *Compiladores aceitam como entrada um programa escrito em linguagem de programação e produzem um programa equivalente em outro código*
- ☁ *Programa objeto: linguagem de máquina, linguagem assembler ou linguagem intermediária*
- ☁ *Sendo P2 o programa fonte e P0 o programa objeto, P2 e P0 devem ser funcionalmente equivalentes*



# Compiladores x plataformas



# *Compiladores diferentes para plataformas diferentes*

- *Diferentes formatos de executáveis*
- *Convenções de chamadas ao SO são diferentes*
- *Uso da pilha do processo difere entre SOs*
- *Diferentes tamanhos dos mesmos tipos de dados*
- *Bibliotecas padrão chamam o SO de maneira diferente*
- *Programas compilados por compiladores diferentes na mesma plataforma não são compatíveis*
- *Diferenças nas estruturas dos sistemas de arquivos*
  - *Uso da "\" no Windows e "/" nos Unix.*

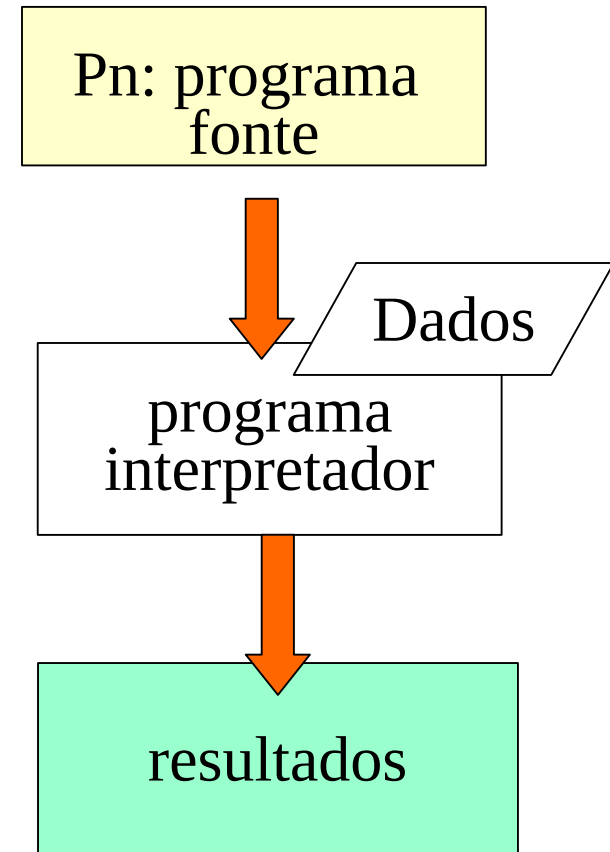
# Interpretador

☁ **Interpretador**: a partir de um programa fonte escrito em uma linguagem de programação ou **código intermediário** e mais o conjunto de dados de entrada exigidos pelo programa realiza o processo de execução







☁ Pode produzir código executável, mas não produz programa objeto persistente

☁ Exemplos: Basic, LISP, Smalltalk e Java\*

\*Java possui uma abordagem híbrida com a compilação e geração de um Bytecode que é executado pela Java Virtual Machine






# *Interpretadores: características*



-  *Interpretadores se baseiam na noção de código intermediário, não diretamente executável na plataforma de destino*
-  *+Simplicidade: programas menores e menos complexos que compiladores*
-  *+Portabilidade: o mesmo código pode ser aceito como entrada em qualquer plataforma que possua um interpretador*
-  *-Propriedade Intelectual: Desprotegida pois o código-fonte deve ser disponibilizado para o cliente.*
-  *-Performance: Prejudicada pois o programa tem que ser compilado sempre antes de ser executado.*
-  *-Segurança: Qualquer pessoa com acesso ao equipamento onde o software está instalado, pode mudar o seu comportamento*

# Compilação e execução





## Compilação:

-  geração de código executável
-  depende da plataforma de execução
-  tradução lenta X execução rápida

## Interpretação pura

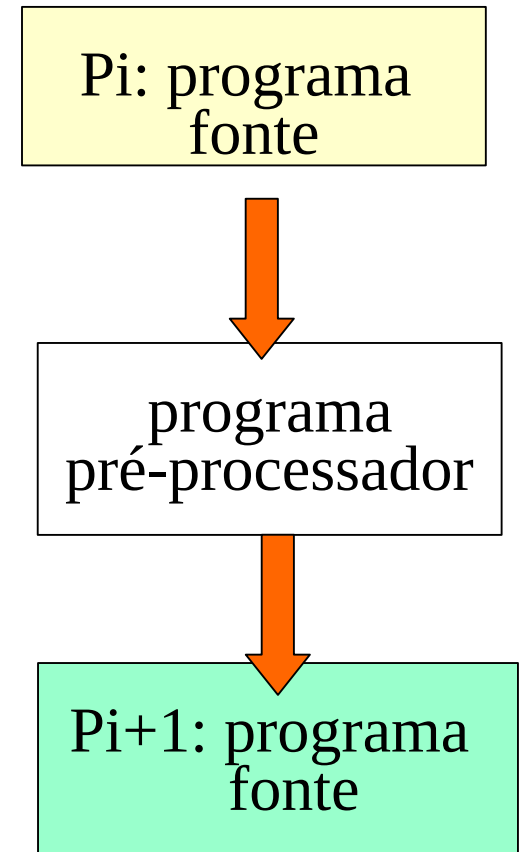
-  sem geração de código
-  execução lenta, independente de plataforma, segurança pode ser comprometida

## Híbrida

-  geração de código intermediário
-  independente de plataforma de execução
-  tradução lenta X execução não tão rápida
-  o 'executável' é sempre o mesmo e pode ser instalado em diferentes plataformas

# Pré-processador

- ☁ Programa que faz conversões entre linguagens de programação de alto nível similares ou para formas padronizadas de uma mesma linguagem de programação
- ☁ Possibilita a utilização de extensões da linguagem (macros ou mesmo novas construções sintáticas) utilizando os processadores da linguagem original. Ex: C ou C++



# Pré-processador: exemplo em C++

☁️ Programas fonte escritos em C++ são inicialmente submetidos a um pré-processador que gera uma unidade de tradução sobre a qual o compilador irá trabalhar

☁️ Finalidades:

- ✂️ inclusão de outros arquivos
- ✂️ definição de constantes simbólicas e macros
- ✂️ compilação condicional do código do programa
- ✂️ execução condicional de diretivas de pré-processamento

☁️ Diretivas para o pré-processador iniciam por #

☁️ Exemplo de macro

#define

CIRCLE\_AREA(x) (PI \* (x) \* (x) )

☁️ por exemplo o comando: `area = CIRCLE_AREA(4);`

☁️ será expandido para:

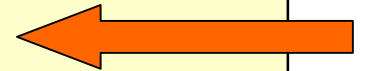
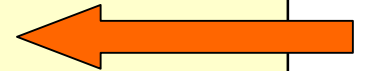
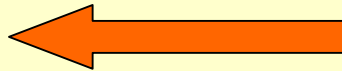
`area=(3.14159 * (4) * (4));`

☁️ antes da compilação

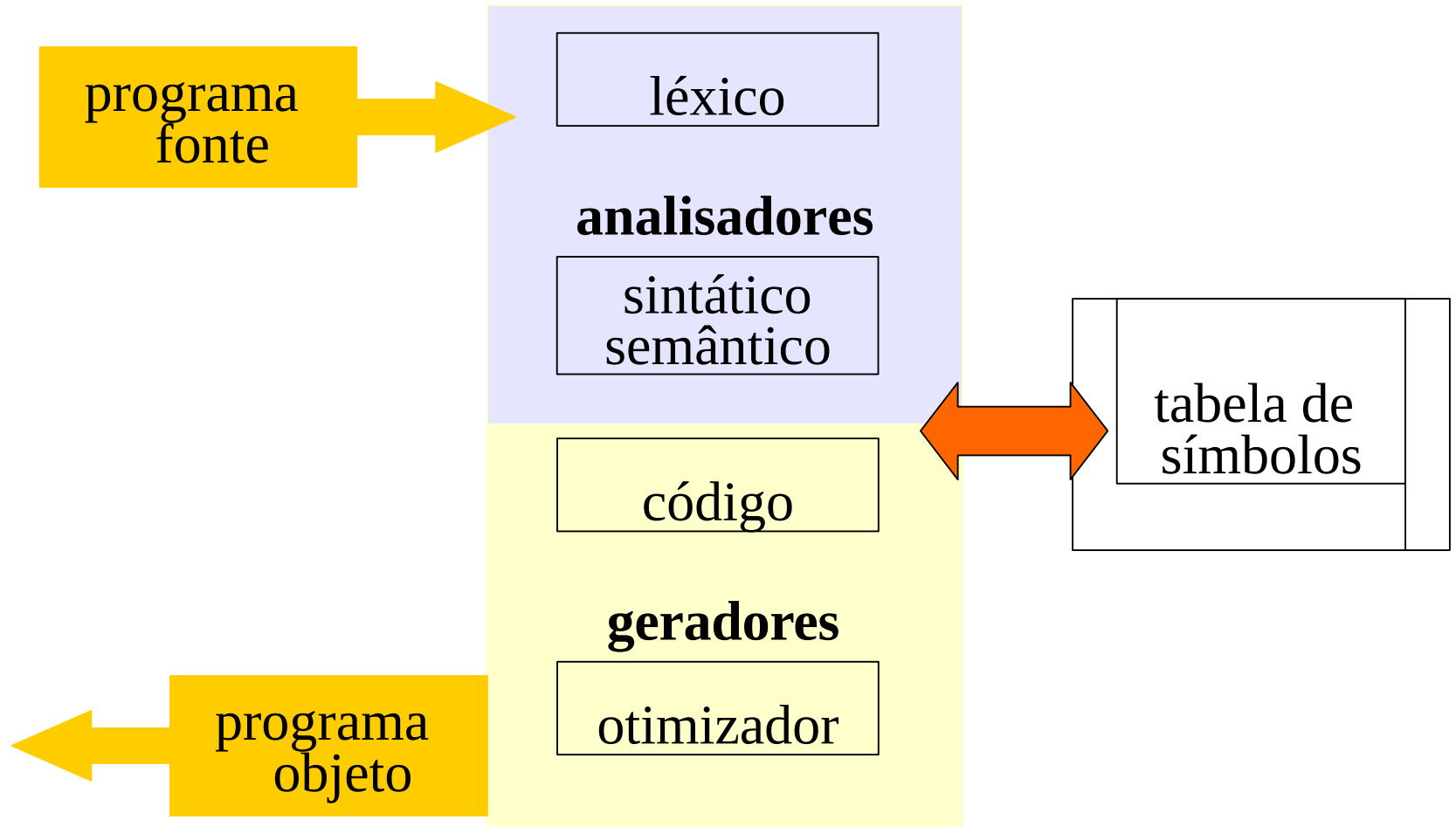


# *Preprocessador: exemplo em C#*


```
#define Dutch
using System;
public class Preprocessor {
    public static void main() {
#if Dutch
        Console.WriteLine("Halo Welt");
#else
        Console.WriteLine("Hello World");
#endif
    }
}
```



# *Componentes de compiladores*



# Analisadores

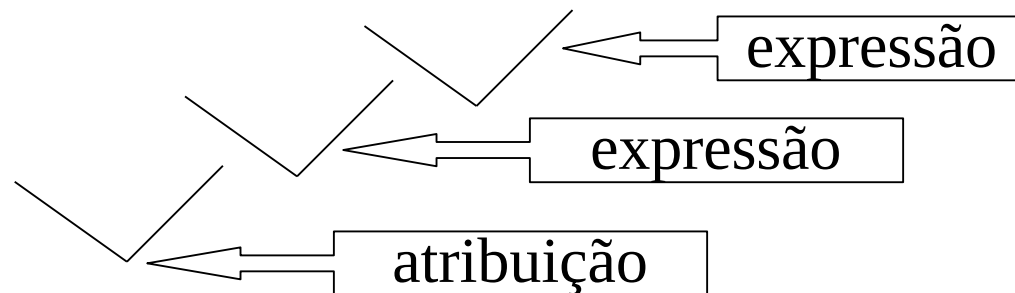
 **Analizador léxico (parser/scanner):** tem por objetivo separar os símbolos individuais da linguagem (tokens)

✂ identificadores, palavras-chave, operadores, etc

 *Analizador sintático: tem por objetivo descobrir a estrutura de cada construção do programa*

✂ *declaração, expressão, atribuição, if, while, ....*

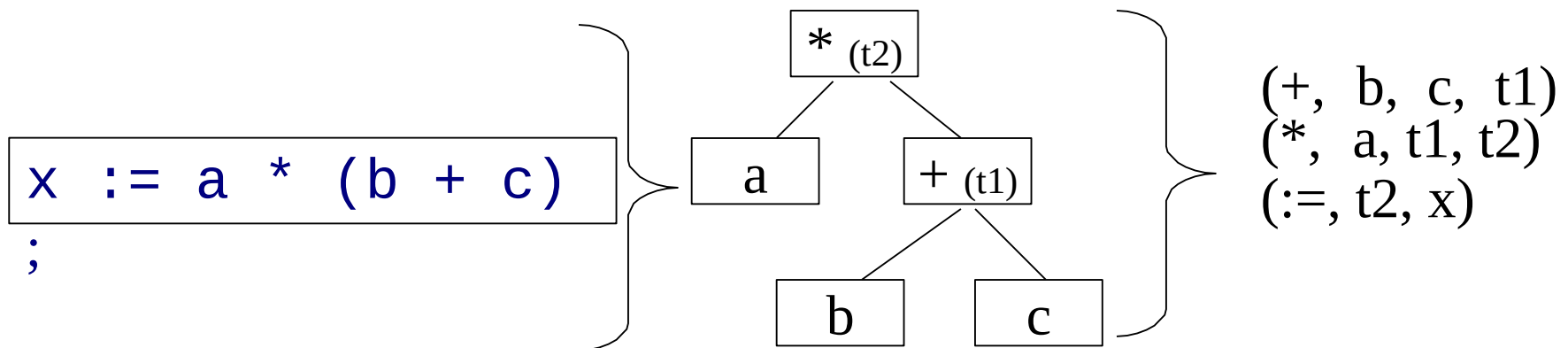
✂ *Exemplo:*  $x := a * (b + c) ;$



# Analizador semântico

- ☁️ *Análise sintática: processo de aplicar a gramática da linguagem para formar a árvore de derivação a partir da sequência de átomos (tokens)*
- ☁️ *Análise semântica: usa a árvore de derivação para gerar uma representação interna do programa, validar interação entre tipos de dados, escopo de variáveis, dentre outros.*

*Exemplo*




# *Tabela de símbolos*

☁️ Tem por objetivo manter um mapeamento entre os identificadores usados no programa (os nomes das variáveis que nós demos) e suas propriedades

☁️ Exemplo:  $x := a * (b + c) ;$   
 $(1) := (2) * ((3) + (4))$

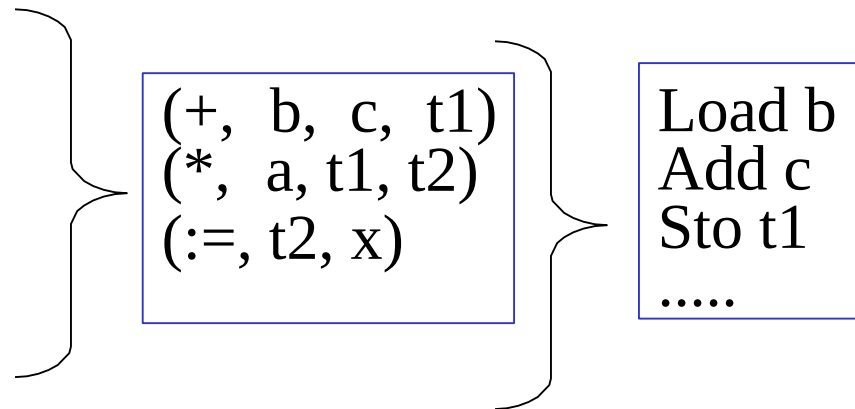
nome	tipo	escopo
x	real	...
a	real	...
b	int	...
c	int	...


# Geradores


 *Gerador de código: tem por objetivo produzir código funcionalmente equivalente de cada construção do programa*

 *Exemplo:*

$x := a * (b + c) ;$



 *Otimizador de código: tem por objetivo aplicar um conjunto de técnicas sobre o código objeto para torná-lo mais eficiente*

 *Exemplos: eliminação de expressões redundantes, substituição de funções “in-line”, etc*

# Compilador Pascal

- ☁️ Programas fonte de usuários podem ser traduzidos para ~~linguagem~~ para P-code
- ☁️ A versão em P-code deve ser interpretada
  - ✂️ maior tempo de execução
  - ✂️ melhores diagnósticos de erros
  - ✂️ favorece o processo de alterações e testes
  - ✂️ permite a reexecução completa ou incremental dentro do próprio ambiente de desenvolvimento

